Combining Bound-T and SWEET to Analyse Dynamic Control Flow in Machine-Code Programs

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

[1]Tidorum Ltd, Finland, niklas.holsti@tidorum.fi
niklas.holsti@tidorum.fi

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

Abstract

The first step in the static analysis of a machine-code subprogram is to construct the control-flow graph. The typical method is to start from the known entry-point address of the subprogram, retrieve and decode the instruction at that point, insert it in the control-flow graph, determine the address(es) of the successor instruction(s) from the known semantics of the instruction set, and repeat the process for the successor instructions until all reachable instructions and control flows are discovered and entered in the control-flow graph. This procedure is straight-forward as long as the successors of each instruction are statically defined. However, most instruction sets allow for dynamically determined successors, usually by allowing the target address of a branch to be set by the run-time, dynamically computed value of a register. We call such instructions *dynamic branches*. To construct the control-flow graph, a static analyser must somehow discover the possible values of the target address, in other words, it must perform a *value-analysis* of the program. This is problematic for two reasons. Firstly, the value-analysis must be applied to an incomplete control-flow graph, which means that the value-analysis will also be incomplete, and may be an under-estimate of the value-set for the complete subprogram. Second, value-analyses typically over-estimate the value-set, which means that the set of possible target addresses of the dynamic branch may be over-estimated, which leads to an over-estimate of the control-flow graph. The over-estimated graph may include instructions and control flows that do not really belong to the subprogram under analysis. This report describes how we connected two analysis tools, Bound-T from Tidorum Ltd and SWEET from Mälardalen University, so that the powerful "abstract execution" analysis in SWEET can be invoked from Bound-T to resolve dynamic branches that Bound-T finds in the machine-code program under analysis. The program-representation language ALF, defined by the SWEET group, is used as an interface language between Bound-T and SWEET. We evaluate the combined analysis on example programs, including both synthetic and real ones, and conclude that the approach is promising but not yet a great improvement. Bound-T contains several special-case analyses for dynamic branches, which currently perform slightly better than SWEET's more general analyses. However, planned improvements to SWEET may result in an analysis which is at least as powerful but more robust than the analyses in Bound-T alone.

1998 ACM Subject Classification F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages–*Program Analysis*

Key words and phrases Worst-case execution-time analysis, WCET, dynamic control flow, indexed branch

Table of Contents

1 Introduction and Overview 3

2 Dynamic Control Flow in Machine Code 3

2.1 Introduction to Static WCET Analysis 3
2.2 The Dynamic Branch Problem 4
2.2.1 Dynamic Branches 4
2.2.2 Switch-Case Statements 5
2.2.3 Calls Through Function Pointers 5
2.2.4 Virtual-Function Calls 6
2.3 Our New Combination 7

3 Coupling Bound-T and SWEET 7

3.1 Overview 7
3.2 Generating the ALF Program 9
3.2.1 Program Representation within Bound-T 9
3.2.2 Basic Translation from Bound-T Internal Form to ALF 9
3.2.3 Translation of Branching Control-Flow 10
3.2.4 Translation of Calls Between Subprograms 11
3.2.5 Translation of Unresolved Dynamic Branches 11
3.2.6 Generating Output-Annotation Specifications 11
3.2.7 Generating Annotations 12
3.3 SWEET Analysis of the Incomplete ALF Program 12
3.4 Using SWEET Outputs in Bound-T 13

4 Experimental Evaluation 14

4.1 Overview 14
4.2 A Loop Containing a Switch-Case using a Jump Table (tp_avr_21, asm) 15
4.2.1 Analysis by Bound-T Alone 16
4.2.2 Analysis by Bound-T and SWEET Combined 17
4.3 Some C Switch-Case Statements with Address Tables (tp_c_2, gcc) 17
4.3.1 Analysis by Bound-T Alone 18
4.3.2 Analysis by Bound-T and SWEET Combined 18
4.4 A Simple Switch-Table and Handler (tp_avr_6, asm) 19
4.4.1 Analysis by Bound-T Alone 19
4.4.2 Analysis by Bound-T and SWEET Combined 19
4.5 A Complex Switch-Table: Sparse Form (tp_c_2 / KuiSnd5Z, IAR) 20
4.5.1 Analysis by Bound-T Alone 20
4.5.2 Analysis by Bound-T and SWEET Combined 21
4.6 A Complex Switch-Table: Dense Form (tp_c_2 / KucDnd11Z, IAR) 23
4.6.1 Analysis by Bound-T Alone 23
4.6.2 Analysis by Bound-T and SWEET Combined 25
4.7 Complex Boolean Address Computation (tp_avr_7/8, asm) 28
4.7.1 Analysis by Bound-T Alone and Combined with SWEET 29

5 Summary and Conclusions 32

5.1 Goals and Methods 32
5.2 Overall Success Rates 32
5.3 Why Some Analyses Failed 33
5.4 Precision of Successful Analyses 34
5.5 Is the Combination More Powerful than its Component Tools? 35
5.6 Suggestions for Improvements and Future Work 36

Document Status and Change Log 38

1 Introduction and Overview

This report describes a procedure for analysing dynamic control flow in machine-code programs. The procedure combines the low-level machine-code analyser Bound-T from Tidorum Ltd [1] with the high- or intermediate-level data- and control-flow analyser SWEET from Mälardalen University [2], which analyses programs represented in the ALF language [3].

The report is organized as follows. Chapter 2 presents the problems posed by dynamic control flow for static analysis. It explains why machine-code programs have dynamic branches and how Bound-T tries to analyse or "resolve" such branches to discover the control-flow graphs, using various analysis methods in Bound-T itself, and why these analysis methods are limited and fail to work on some important classes of dynamic branches, in particular those where the machine-code program intentionally uses wrap-around or overflow in its computations. Chapter 3 then shows how we have connected Bound-T and SWEET so that the powerful analyses in SWEET can work on dynamic branches discovered by Bound-T.

Chapter 4 evaluates the tool combination on a set of example programs. Chapter 5 summarises the report, draws some conclusions, and sketches possible future work.

The pilot implementation described in Chapters 3 and 4 analyses machine-code programs for the Atmel AVR 8-bit processor architecture, a widely used microcontroller family [5]. The AVR can be considered a difficult subject for dynamic branch analysis because it implements mostly 8-bit operations and does not provide general base+index addressing. The AVR has a Harvard architecture and separate instructions for reading data from the code memory and from the data memory. Code addresses are 16 or 22 bits wide (depending on the AVR device) and so must be 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 for such register pairs is quite asymmetric and non-orthogonal. These limitations mean that the machine code generated for various kinds of dynamic branches in high-level languages is complex and not very idiomatic. This makes pattern-matching analysis methods unreliable, and semantically based analyses, such as ours, should be competitive.

This work was supported by the APARTS project (Advanced Program Analysis for Real-Time Systems). APARTS is a collaboration project (grant agreement no. 251413) funded by the European Commission's 7th Framework Programme under the IAPP activity (Industry-Academia Partnerships and Pathways) of the Marie Curie Mobility Actions.

2 Dynamic Control Flow in Machine Code

2.1 Introduction to Static WCET Analysis

Software programs in real-time systems are subject to real-time performance requirements, such as fixed periods for cyclic control loops and deadlines for responses to sporadic inputs. To verify that such real-time requirements are always met, it is useful (in principle even necessary) to have bounds on the worst-case execution time (WCET) of the relevant parts of the program. Static WCET analysis is a form of static program analysis that computes WCET bounds from the program itself, which is usually given in machine-code, executable form. The analysis, being static, covers all execution paths and all possible input data values, and therefore delivers safe bounds. The Bound-T tool from Tidorum Ltd [1] is such a WCET analysis tool. The SWEET tool from Mälardalen University [2] started as such a tool, but is now focused on the value- and control-flow analysis phases, to be described shortly, and is applied to an intermediate-level program representation [3] rather than machine language.

Static WCET analysis is usually applied to selected architectural components of a whole program, such as task-body subprograms or interrupt-handler subprograms. However, for simplicity we will use the term "program" to mean the part of the program that is analysed. In practice, this "program" is usually a selected subprogram and its callees.

Static WCET analysis of a machine-code program usually (and also in Bound-T) proceeds in several phases as follows [4]. First, the instructions in the program are decoded and organized into some kind of control-flow graph (CFG), or a set of such graphs, that shows the structurally possible execution paths (instruction sequences) in the program. The nodes in these graphs are usually basic blocks of instructions and the arcs represent control-

flow transfers that can be either normal flow from an instruction to the next instruction in address order, or branches in the flow caused by jump or call instructions. This first phase is called *CFG reconstruction*.

When the control-flow and call-graphs are available, a *processor-behaviour analysis* computes an upper bound for the execution time of each graph element (node, arc). For complex processors, the actual execution time of a graph element can depend greatly on the context (processor state) in which this element is executed, which means that the execution time depends on the execution path leading up to this graph element as well as on the current variable values when this graph element is executed. The processor-behaviour analysis produces a WCET bound that covers all possible contexts.

The existence of several paths to, and contexts for, a graph element can lead to an over-estimated WCET bound for the program, if the graph element is often executed in contexts where its real WCET is much less than its WCET in other contexts. To avoid this imprecision, the graph can be expanded by unpeeling loops and inlining called subprograms, which copies graph elements into separate execution paths. In the rarely used limit, this expansion can result in a single path to each graph element and therefore a single, context-specific, WCET for each element. In this limit, each graph element is executed at most once, in any execution of the program.

Usually, however, the graph is not expanded to such a limit, and therefore contains joining or looping paths which means that some graph elements can be executed several times in one execution of the whole program. The graph is then subjected to *flow analysis* to compute constraints (typically upper bounds) on the number of executions (the "execution frequency") of each graph element. Flow analysis usually depends on a *value analysis* that computes bounds on the values of program variables, especially variables that control program flow and therefore determine the number of iterations of loops and the depth of recursions. Several methods exist for computing loop and recursion bounds from value-analysis results, but they all depend on some concepts of control-flow patterns, such as loop induction variables ("loop counters") and their steps and limits.

When the analysis has found bounds on loops and recursions, and perhaps other bounds on the execution frequency of CFG elements, the overall WCET bound is typically computed using the Implicit Path Enumeration Technique (IPET) as follows: the execution time is expressed as the sum of the initially unknown execution frequency of each graph element, multiplied by the WCET bound of that element, and summing over all graph elements. This is a linear expression of the unknown execution frequencies. Its maximum value, under the known execution-frequency constraints, is the overall WCET bound, and can be found by an Integer Linear Programming solver. This last step is called the WCET *calculation* phase.

2.2 The Dynamic Branch Problem

2.2.1 Dynamic Branches

The above division of the analysis into phases, starting with CFG reconstruction and ending with WCET calculation, runs into difficulty when the program under analysis has *dynamic branches*, also known as *indirect* or *indexed* branches. In a dynamic branch, the target address is not given statically in the branch instruction, but is defined by the run-time, dynamically computed value held in a register or memory location. For example, the Atmel AVR processor has an instruction called `ijmp`, for "indirect jump", which jumps to the instruction at the address held in the `Z` register when the instruction is executed. The CFG reconstruction procedure must treat dynamic jumps differently from normal instructions in which the successor instructions are statically determined. Several methods for analysing dynamic branches have been described. We will summarise them based on a classification of the nature and origin of the dynamic branch.

Dynamic branches may be created (by compilers or assembly-language programmers) for a great many reasons, some more common than others. The common reasons are the following:

- *Switch-case statements*. Many compilers translate some forms of switch-case statements into code which uses dynamic jumps. This code can take a multitude of forms. A common, simple form uses the switch selector expression as an index to extract the address of the corresponding case from a table of such addresses and then performs a dynamic jump to the extracted address. The table is constant and so can be located in code space (EEPROM or flash memory).

- *Calls through function pointers*. A function pointer holds a dynamically determined address to a function (to its entry point). A call through a function pointer is typically compiled into a dynamic call instruction. However, some processors have no dynamic call instructions, so the same effect must be achieved in some

tricky way. For example, one can place the function pointer's value into a register and call a specific "trampoline" function which pushes this register onto the stack and then executes a return instruction. This return instruction transfers control to the referenced function by using the pushed address as the "return" address.

- *Virtual function calls*. This is a special case of a call through a function pointer, where the function pointer does not exist alone, but is one element of the "virtual function table" associated with the class of the object upon which the virtual function is called. Virtual function tables are usually constant throughout the execution, while other kinds of function pointers are often variable.

Next we briefly discuss the analysis methods that have been used for each of the above types of dynamic branch.

2.2.2 Switch-Case Statements

Dynamic branches from switch-case statements are often among the easier forms of dynamic flow to analyse. This is because the set of possible target addresses is constant and does not depend on the history of execution nor on assignment statements elsewhere in the program (as often happens for function pointers). The analysis can therefore be local to the subprogram that contains the switch-case statement. The code generated for a switch-case statement commonly has one of three forms:

1. A cascade of comparisons (*if-then-elsif ... end if*), in which all branches are static. This code often results from switch-case statements with a sparse case numbering. As it lacks dynamic branches this code structure is simple to analyse (for CFG reconstruction, at least) and is not interesting for our purposes.

2. A table that is more or less directly indexed by the switch expression's value and yields more or less directly the address of the code which corresponds to this value. Code that uses such dense (contiguous) tables usually results from switch-case statements with a dense case numbering and more than a few cases. The table can take various forms. For example, an entry in the table can contain the case-code address directly, or an offset from some fixed base point to the case-code, or it can even contain a static jump instruction which jumps to the case-code.

3. A complex, perhaps compressed table which lists the case numbers and the corresponding case-code addresses, but is not directly indexed by the case number, combined with a library subprogram which can scan and interpret such tables. This kind of "switch-table" code is typically generated by C compilers for small microcontrollers when the case numbering is sparse and the case numbers are wide (for example, 32-bit integers on an 8-bit processor). For wide case numbers, each comparison in a cascade structure would require several instructions. However, as microcontrollers with very small code memories are becoming rare, this form of switch-table may also be disappearing.

In typical 32-bit RISC processors, switch-case statements using type 2 dense tables of addresses can be implemented by a handful of instructions in a very idiomatic way. Simple pattern-matching methods can detect such code, find the location and size of the table, and extract the possible target adresses (case-code locations) from the table. This is safe if the table is known to be in read-only memory, as is typically the case for small microcontrollers with limited RAM. However, in larger computers, the EEPROM code, including such address tables, is typically copied into RAM for execution, and it can then be difficult for the analysis tool to know what the table contains and whether the contents are really constant. Bound-T applies such pattern-matching analysis for some target processors, but just assumes that the table is constant. However, there is a always a risk that compilers will use different code idioms, or use the same code idiom for a table which is not constant during execution.

To analyse switch-case code of type 2, where a complex table is interpreted by a library subprogram, Bound-T uses partial evaluation of the interpreting subprogram [6]. However, the method assumes that Bound-T knows the names and calling protocols of the interpreting subprograms, which are compiler-specific and can change as the compilers evolve.

2.2.3 Calls Through Function Pointers

The value of a function pointer is usually not computed using arithmetic operations, as switch-case addresses can be. In a "higher" level programming language such as C or Ada, the only possible function-pointer values are the

static constants defined by taking the address of a statically known subprogram. The program can take the address of a function explicitly, using a primitive expression such as `&foo` in C or `Foo'Address` in Ada, or the address can be taken implicitly if the subprogram can be called "virtually" as an inherited operation of an object in a class hierarchy. These function-pointer values, originally static, can then be passed around the program according to simple or complex dynamic logic, stored in simple or complex data structures, and finally used to call the addressed subprogram. It is this dynamic, conditional copying and selection of static values which makes the target address of the call dynamic.

Of course, a C or Ada programmer can create truly dynamic, new function-pointer values by deliberate type-breaking, for example by casting a function pointer to an integer type, then modifying the integer value by arithmetic operators, and casting the value back to a function pointer. However, the effect of such manipulations is not defined by the language standards, and they are fortunately rare in real programs.

Because function pointers are not created by arithmetic operations, analysis methods based on arithmetic, numerical, abstract domains such as intervals are rarely useful. More useful are pointer-analysis methods which model pointer values as sets of discrete constants. Neither Bound-T nor SWEET currently provide such analyses for function pointers. Moreover, pointer analysis generally requires a *global* program analysis, which Bound-T tries to avoid (but SWEET provides). We will therefore not consider function-pointer analysis in the rest of this report. However, function pointers are becoming more common in real-time, embedded software, partly due to the increasing use of model-based programming tools and other code-generating frameworks. Automatic analysis of function pointers would certainly be an attractive feature of a WCET-analysis tool.

2.2.4 Virtual-Function Calls

In object-oriented programming, operations (subprograms) originally defined for a more general type or class of object are inherited by the more specialized, derived child classes, but can also be reimplemented (specialized, overridden) in the child classes. Moreover, programs can use variables of unknown specific class — that is, while such a "class-wide" variable *x* is statically known and declared to refer to an object of class *C* or a child class of *C*, it is not statically known to which specific class this object belongs. If *foo* is an operation defined on the statically known root class *C*, the program can apply *foo* to *x*, but the actual operation which is invoked depends on the run-time actual class of the object *x*.

Such virtual or "dispatching" operations are usually implemented by collecting pointers to the operations of a particular class into a table, called the *virtual-function table* or v-table of the class. (For this simplified explanation, we ignore the complications of multiple inheritance and inheritable interfaces.) Each operation is assigned its own index into the v-table. The same index is used when the operation is inherited by child classes. Each child class has its own version of the v-table. When the child class inherits and does not override an operation, the child's v-table points to the operation of the parent class. If the child class overrides an inherited operation, the child's v-table points to the overriding, child-specific operation. The run-time value of a class-wide variable *x* contains a reference to the v-table of the actual class of the object to which *x* refers. To implement a virtual-function call, the code simply indexes this v-table with the statically known index of the virtual operation, extracts the pointer to the actual operation, and then calls this operation.

Thus, virtual-function calls are a disciplined use of function pointers, in which the dynamic values are the v-table pointers, while the indices assigned to operations, the v-tables, and the actual function-pointers contained in the v-tables are static constants. A set-based, global data-pointer analysis can work for the v-table pointers. SWEET can implement such analysis if each v-table is modelled as its own ALF "frame". Numeric analysis of the operation indices would be overkill, because they are constant (with the possible exception of some implementations of multiple inheritance).

Bound-T provides no general analysis of virtual-function calls. However, some compilers (specifically, compilers from IAR Systems) embed a description of the class hierarchy and v-tables into the symbol-table information (also known as debugging information) in the file which contains the compiled and linked program. The compiler also provides information that shows which subprogram calls are virtual-function calls and for such calls gives the static root-class of the object and the index of the operation. Bound-T can then look up the operation in the v-tables of the root class and all its child classes and thus find the set of possible actual subprograms that may be called through this virtual-function call. However, this set is usually an over-approximation, perhaps even a crude one. An actual analysis of object classes (v-table pointers) would often given a more accurate result.

At the time of writing, we have not implemented virtual-function-call analysis in our prototype combination of Bound-T and SWEET. In the most natural implementation approach Bound-T would have to know about the structure of class-wide objects and v-tables as used in the program under analysis, which is compiler- and target-specific, and would then be able to translate each v-table into its own ALF frame.

2.3 Our New Combination

This work attempts to combine the strengths of the Bound-T tool and the SWEET tool [2]. SWEET was originally aimed at WCET analysis as its full name "Swedish Execution-Time Tool" indicates. However, SWEET is currently focused on value-analysis and control-flow analysis. SWEET analyses programs given in the ALF language [3]. In addition to the conventional value analyses based on abstract interpretation with various numerical domains, SWEET provides a unique feature called *abstract execution* [7, 8]. This is similar to abstract interpretation in that variable values are abstracted and program instructions are similarly abstracted to compute with abstracted values and produce abstracted results, but it differs from abstract interpretation by not using widening for loops. Instead, loops are abstractly executed as many times as necessary until the iterations reach an abstracted state where the loop must terminate. This makes the value-analysis more precise (all value-sets which were bounded initially remain bounded during the abstract execution) and also provides detailed information about control-flow in each iteration, but has the draw-back that the abstract execution may not terminate.

When the abstract execution reaches a control-flow join, it may or may not merge the abstracted values from the incoming paths. The merging is controlled by several command-line options and can even be completely shut off, in which case the abstract execution in effect simulates the program, traversing all possible paths and keeping their states separate, although with (possibly) abstracted values.

Our aim is to use abstract execution, with no merging, to compute the possible values of the target addresses of the dynamic branches that Bound-T finds in the machine-language program under analysis.

The first step is of course to translate the target program under analysis from Bound-T's internal representation into an ALF representation. This is mostly straight-forward, but with one major complication: if the program has unresolved dynamic branches, Bound-T's internal representation is incomplete, which means that the ALF program will also be incomplete. The ALF language has features which can model dynamic branches but these assume that all targets of the branches are already present in the ALF code, which is not (necessarily) the case here. Therefore, the analysis becomes iterative: Bound-T generates an incomplete ALF program, SWEET analyses the program to discover possible target addresses, and then Bound-T extends the program to include the code at these addresses, and the process is repeated until the CFG is stable.

3 Coupling Bound-T and SWEET

3.1 Overview

This chapter explains how we coupled Bound-T and SWEET to make use of SWEET's abstract execution analysis for resolving dynamic jumps and thus help Bound-T analyse the given machine-code program. We explain the overall process and some of the important details.

It is important to note that Bound-T at present requires all dynamic flow of control to be resolved in a *context-independent* way. In other words, when a subprogram contains a dynamic branch, the possible target addresses must be discovered by analysis of this subprogram (and its callees) only, without analysing the higher-level subprograms which call this subprogram. Therefore, the analysis of dynamic branches in Bound-T is focused on branches which can be resolved using local analysis, typically switch-case structures. Bound-T usually cannot analyse calls through function pointers because a global program analysis is usually necessary to find the possible values of the pointers. This limitation to context-independent analysis applies also in our current implementation of the Bound-/SWEET combination, because removing the limitation would require significant architectural changes to Bound-T.

The limitation to context-independent analysis of dynamic branches simplifies the coupling of Bound-T and SWEET. It means that the part of the program to be analysed is a subprogram call-graph in which unresolved dynamic branches occur only in the root subprogram — all other (deeper) subprograms contain only static

branches (which may be the results of resolving dynamic branches in earlier analyses of these subprograms). The analysis procedure consists of the following steps:

1. Bound-T exports the call-graph into an ALF program. The unresolved dynamic branches are represented as `{return}` statements. The ALF program is an incomplete representation of the actual machine-code program because the code reached through the dynamic branches may be absent. Moreover, all execution paths through the dynamic branches are absent, even if the target code is reached also through static paths and is therefore present in the ALF program. Bound-T also creates the other SWEET input files: the annotation file and the output-annotation specification file. The annotation file transfers known or assumed bounds on variable values from Bound-T to SWEET. The output-annotation specification file tells SWEET to analyse and output the possible values of the variables which determine the target addresses of the dynamic branches, at the point of the branch. Bound-T then invokes SWEET as a child process.

2. SWEET analyses the incomplete ALF program and generates various forms of results. For our purposes the principal result is the *output-annotation file* which contains SWEET's responses to the output-annotation specifications, that is, the possible values of the variables which determine the target addresses of the dynamic branches.

3. Bound-T reads the output-annotation file from SWEET. If all went well, this defines, for each dynamic branch, a small set of possible values for the variable which determines the target address of the branch. Bound-T adds the corresponding new edges to the control-flow graph and, if these edges lead to hitherto undiscovered instructions, fetches these new instructions from the machine-code file and continues decoding and extending the control-flow graph until all new, statically determined parts are found. If this reveals new dynamic branches, Bound-T repeats the whole process, iterating until the control-flow graph is complete, or until the remaining dynamic branches cannot be resolved using any of the available analyses.

Figure 1 below illustrates this procedure and its data and control flows.

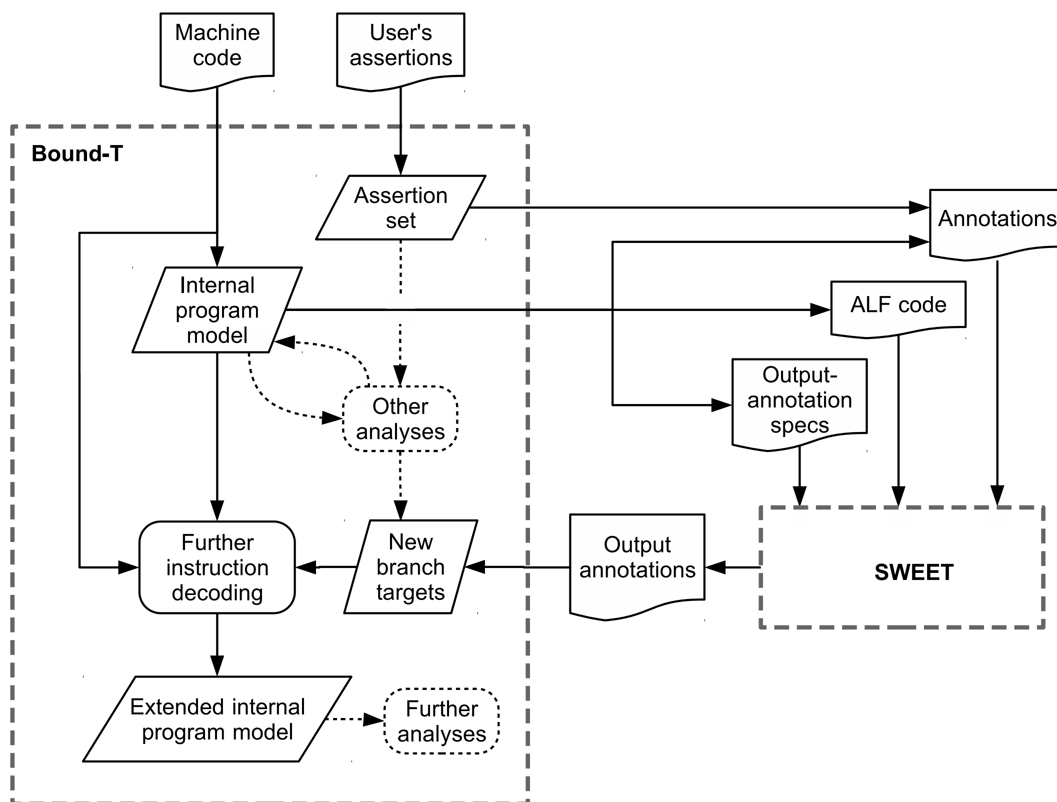

Figure 1: Overall Analysis Scheme Combining Bound-T and SWEET

3.2 Generating the ALF Program

3.2.1 Program Representation within Bound-T

Bound-T internally represents the program under analysis as a set of subprograms and a set of calls between subprograms. Each subprogram is represented as a control-flow graph consisting of *steps* and *edges*. A step is the smallest unit of execution and usually represents one machine-code instruction. An edge between steps represents execution flow from an instruction (the source of the edge) to a successor instruction (the target of the edge). Any number of edges can start from a step (the *out-edges* for that step) and end at a step (the *in-edges* for that step). There is exactly one step which has no in-edges; this is the entry point of the subprogram. Steps with no out-edges represent return points; there can be any number of return points in a subprogram.

Each step is associated with a computational *effect* which is a sequence of *assignments* which are executed in parallel. An assignment evaluates an expression to produce a value, and then stores this value into a *storage cell*. The expressions apply arithmetic and logical operators to constants and the values of storage cells. The set of arithmetic and logical operators, and their respresentation, has been adapted to resemble the operators and expressions in the ALF language (originally, in early versions of Bound-T, this was not the case).

A call instruction is represented by two steps: one normal step which represents the call instruction itself, followed by a special *call-step* which represents the execution of the callee subprogram. The normal step represents the effect of *just* the call instruction, for example the storing of a return address somewhere. The call-step represents the effect of the callee. Bound-T has a very approximate idea of the overall effect of a subprogram: Bound-T tries to find out which storage cells are assigned in the subprogram, but not which values are assigned. Therefore, the effect of a call-step, in the Bound-T internal model, assigns unknown values to all these cells. (Note, however, that Bound-T does not discover cells assigned through unresolved dynamic data pointers, so here Bound-T is not sound.)

Each edge in a control-flow graph is associated with a logical condition which is a necessary, but perhaps not sufficient, condition for execution to flow along this edge. The condition is a Boolean expression of the values of storage cells and constants. The possible insufficiency of the condition is a deliberate feature to allow approximate modelling of control-flow decisions. For example, an edge which may be taken for unspecified (unmodelled) reasons can be given the condition "true" without implying that the edge *must* be taken.

An unresolved dynamic branch is modelled by a special kind of edge which has a known source step, an unknown target step, and information showing which storage cells determine the value of the target address and how that address is computed. Various analyses implemented in Bound-T can then provide bounds of various kinds on the values of these storage cells, from which one or several possible target addresses may be derived. These analyses include constant propagation, value-origin (def-use) analysis, and the so-called "arithmetic analysis" which models computations with Presburger Arithmetic relations, as (briefly) explained in [10], and solves them with the Omega Calculator [9]. The work we report here adds the use of SWEET to the set of analyses with the ability to resolve dynamic branches.

3.2.2 Basic Translation from Bound-T Internal Form to ALF

The translation from the internal Bound-T program representation to ALF [3] is fairly straight-forward:

- Each subprogram is translated into an ALF `{func}`.
- Each step in a control-flow graph is translated to an ALF `{store}` statement which models the effect of the step.
- If a step has a single out-edge, in other words, a single successor step, the ALF statement for the successor step is placed after the ALF statement for the first (predecessor) step. In other words, ALF's sequential execution semantics models this edge. If the successor step has other in-edges, those other in-edges are modeled as ALF `{jump}` statements which are placed after the ALF statements for those other predecessor steps.
- If a step has several out-edges, these edges are modelled by one or more ALF `{switch}` statements, depending on the number of out-edges and the nature of their conditions as described in more detail below.

The AVR program code (that is, the code-memory image) is translated to an ALF constant-data frame which is initialized from the memory image of the program under analysis. This gives the ALF program access to the constant data embedded in the code. We will see that dynamic branches often depend on such constant data.

3.2.3 Translation of Branching Control-Flow

For steps with several out-edges, the translation is complicated by the non-deterministic (necessary but not sufficient) nature of the conditions on Bound-T edges, which clashes with the exact, sequential semantics of the ALF `{switch}`. Moreover, a `{switch}` compares an integer expression to a set of distinct constant values, one per possible target address, while in Bound-T we have a set of Boolean conditions, one per edge, and not necessarily mutually exclusive. The translation considers three cases as follows:

1. If the step has exactly two out-edges, and their conditions are syntactically complementary (for example, one condition is $x = 0$, the other is $x \neq 0$, for some storage cell x), the edges are translated to one `{switch}` in which the expression is one of the conditions (a 1-bit value) and there are two targets for the values 0 (false) and 1 (true), respectively. Note that this case, with two complementary out-edges, is by far the most common form of branch in Bound-T control-flow graphs. Note also that in this case, the conditions on the two out-edges are exact, that is, each condition is both sufficient and necessary for the condition's edge to be taken.

2. Otherwise, if the conditions on all the out-edges have the form $expr = c$, where *expr* is some expression, same for all the out-edges, and c is some constant, different for each out-edge, the out-edges are translated to one `{switch}` in which the expression is *expr* and there are as many targets as out-edges, each target with the corresponding value of the constant c. Sets of out-edges with conditions of this form are often generated in Bound-T when a dynamic branch is resolved (within Bound-T), so this case is not so unusual as it may seem.

3. Otherwise, the following general translation is used. The out-edges are translated in some arbitrary order. Each out-edge, except the last one, is translated into two `{switch}` statements, both with 1-bit expressions and two targets for the values 0 and 1, respectively. The first `{switch}` has an "unknown" expression, one target which skips to the translation of the next out-edge, and one target which continues to the second `{switch}` for the current out-edge. This second `{switch}` has the out-edge condition as its expression, a target for 0 (false) which continues to the translation of the next out-edge, and a target for 1 (true) which branches to the translation of the current out-edge's target step. Here the first `{switch}` represents the possibly insufficient nature of the edge condition by letting control flow to some other edge even if this edge's condition is true. The second `{switch}` represents the necessary nature of the edge condition. The last out-edge is translated into a single `{switch}` which has the edge condition as its expression and a single target, for 1 (true), which branches to the translation of the out-edge's target step. The reason for treating the last out-edge differently is that execution reaches this point only if none of the other out-edges were taken, which means that this last out-edge must be taken.

An example may help to understand the general translation in the last point above. Assume that the step has three out-edges e, f, and g, with the conditions $c(e)$, $c(f)$, and $c(g)$. The ALF translation is then equivalent to the following pseudo-code:

```
-- Translation of edge e:
if unknown then
   if c(e) then
      goto target(e);
   end if;
end if;
-- Translation of edge f:
if unknown then
   if c(f) then
      goto target(f);
   end if;
end if;
-- Translation of the last edge, g:
```

```
assert c(g);
goto target(g);
```

3.2.4 Translation of Calls Between Subprograms

A step representing a call instruction is translated into an ALF `{store}` statement in the normal way. The call-step is translated into an ALF `{call}` statement. In our current implementation no ALF parameters are passed and no ALF results are returned in the `{call}`; all communication between caller and callee occurs through global ALF variables (including those representing machine registers). The proper modelling of stacks and local variables in ALF is still under consideration, with several open questions on aliasing, different stack growth directions, and the best way to model such things using ALF data frames.

Note that Bound-T's own analysis of which storage cells may be assigned (modified) by the callee (the effect of the call-step) is *not* exported into the ALF form. Instead, we rely on SWEET to analyse such inter-procedural data flow. This should be an improvement, because SWEET handles pointers safely, which Bound-T does not.

3.2.5 Translation of Unresolved Dynamic Branches

Each unresolved dynamic branch is translated into an ALF `{return}` statement or into a `{jump}` to a `{return}`. Although this allows ALF execution to flow through the dynamic branch, it does not introduce false paths, because such branches (currently) occur only in the root subprogram, and a return from the root subprogram terminates the ALF execution path.

If there could be unresolved dynamic branches in deeper subprograms, and if these were translated to returns, the corresponding execution paths in the ALF program would continue execution in the caller and would in general be false paths. Such false paths could harm the SWEET analysis and should be avoided.

If Bound-T is ever extended to allow context-dependent analysis of dynamic branches, the translation of an unresolved dynamic branch to ALF must use some ALF code which terminates execution at that point, for example some kind of halt instruction, which does not now exist in ALF. There is a suggestion to use instead the ALF equivalent of *assert*(*false*), which is a "blind" `{switch}` in which no target matches the expression's value. However, this ALF construct has alternative uses with different semantics; for example, to mark an infeasible execution path. Tidorum would therefore prefer that a true `{halt}` instruction be added to ALF, with the meaning that execution (*i.e.* SWEET's analysis) should stop at this point, but without implying that the execution path is infeasible or that any other sort of error has occurred. If an output-annotation specification requests some output at the `{halt}` point, SWEET should produce this output.

3.2.6 Generating Output-Annotation Specifications

SWEET's abstract-execution analysis gives two kinds of results: firstly *flow-facts*, which show the possible execution paths, including loop bounds and other execution frequency bounds, and secondly sets of possible *variable values* at each point in the program. For this work we are interested in the variable values, and in fact only in the values of the variables which determine the targets of dynamic branches, and only at the dynamic branches, not elsewhere in the program.

SWEET has a feature designed for reporting such analysis results: *output annotations* and *output-annotation specifications*. The output-annotation specifications are input to SWEET; each such annotation specifies a point in the program and a list of variables to be reported. During the analysis, SWEET collects the (abstracted) values of these variables, at this program point. At the end of the analysis, SWEET writes an output annotation file which reports these (abstracted) observed values in the form of a SWEET annotation (which can, if useful, be given as input to SWEET for another analysis).

After Bound-T has generated the ALF program to be analysed, with some unresolved dynamic branches, Bound-T writes an output-annotation specification file which contains one such specification for each unresolved branch in the ALF program. This specification asks for the values of the variables which determine the target address, on entry to the ALF `{return}` statement which represents this branch. As will be explained below in section 3.4, we apply this analysis only to dynamic branches in which the target address depends on only *one* variable, so each output-annotation specification names only one variable. Different dynamic branches can depend on different variables or on the same variable.

An example of an output-annotation specification, generated by Bound_T, is this one, which asks SWEET to record and output the values of the first 16 bits of the frame named "pZ", on entry to the ALF statement labeled

"KuiSnd5Z_Step_54" (with zero offset) in the subprogram named "KuiSnd5Z; this statement is a `{return}` which stands in place of a dynamic jump instruction:

```
STMT_ENTRY "KuiSnd5Z_1" "KuiSnd5Z_1_Step54" 0 "pZ" 0 16;
```

Section 3.4 shows the result, from SWEET, of this output-annotation specification.

3.2.7 Generating Annotations

Bound-T generates an annotation file to guide SWEET's analysis. The file contains the following kinds of annotations:

- Annotations which constrain all variables in the ALF program to have integer values, not ALF frame references, ALF statement-label references, or floating-point values. An ALF frame reference is a semi-symbolic value that refers to a location within an ALF "data frame" by giving the frame identifier (symbolic) and the offset (numeric). An ALF-statement-label reference is a similar semi-symbolic reference to an ALF statement. In the ALF translation, Bound-T does not at present use variables holding such references or floating-point values. Eliminating them by these annotations makes SWEET's analysis more precise.
- Annotations on the values to be assumed for certain variables on entry to a certain subprogram, translated from corresponding user-written assertions in the Bound-T assertion language. These annotation are used to constrain the analysis by giving additional information, for example on the register-usage conventions in the program under analysis.

An example of the first kind of annotation is this one, which tells SWEET to assign the "top integer" value to the variable consisting of the first 8 bits of the data frame named "d5F", on entry to the statement labeled "KuiSnd5Z_1_Step1" (with zero offset) in the subprogram named "KuiSnd5Z_1":

```
STMT_ENTRY "KuiSnd5Z_1" "KuiSnd5Z_1_Step1" 0 ASSIGN "d5F" 0 8 TOP_INT;
```

The "top integer" value excludes all ALF reference values. An example of the second kind of annotation is this one, which constrains the first 8 bits of the frame "r1" to tbe zero at the same program point:

```
STMT_ENTRY "KuiSnd5Z_1" "KuiSnd5Z_1_Step1" 0 ASSIGN "r1" 0 8 INT 0;
```

This usage of the AVR register `r1` is a *gcc* convention and is essential information for analysing AVR code generated by *gcc*, but cannot be discovered from the code itself (unless the analysis includes the boot and start-up code, where `r1` is set to zero). We used such assertions in some of the examples reported in Chapter 4. It is likely that future extensions of our prototype will incorporate more kinds of annotations, at least as translations of other kinds of Bound-T assertion inputs.

3.3 SWEET Analysis of the Incomplete ALF Program

After generating the ALF program, the annotation file, and the output-annotation specification file, Bound-T activates SWEET as a child process. In our current implementation, Bound-T asks SWEET to analyse the incomplete ALF program using *abstract execution* with *no merging*. This is the most exact (least over-approximating) analysis method in SWEET; it is equivalent to a concrete execution of the ALF program, saving all execution states, unless the program uses some input variable which has an abstracted set of values, rather than a single, concrete, initial value.

In our application, where SWEET analyses an ALF "program" which represents only a sub-call-graph of the whole machine-code program, the analysed part can have such abstracted input variables, either global variables or parameters to the root subprogram. These variables may be bounded by annotations, which are called "assertions" in the context of Bound-T. As explained in section 3.2.7 our present implementation translates to SWEET annotations only those variable-value assertions which are placed at the entry point of a subprogram.

In addition to the "no merge" option for the abstract execution, Bound-T also asks SWEET not to merge the values in output annotations. Thus, each value recorded during the abstract execution is produced separately, instead of producing an interval which contains all the recorded values. Again, this increases the precision and

reduces the over-estimation of the analysis, but can generate a large list of possible values if the abstract execution over-estimates the value-set.

3.4 Using SWEET Outputs in Bound-T

When SWEET finishes the analysis of the ALF program, Bound-T reads the output annotations which SWEET generated in response to the output-annotation specifications. An output annotation has the same form as an input annotation (examples of which are shown in section 3.2.7) except that, with the "no merge" option, a list of possible values and value-intervals can appear, instead of a single value or single value-interval. Thus, the set of values reported in an output annotation, with "no merge", is a disjunction of single values and intervals, and is not necessarily a convex (contiguous) set. This is an important feature, because the value-sets used in dynamic branches are seldom convex.

For example, here is the output annotation that SWEET generates in response to the output-annotation specification shown in section 3.2.6, after its analysis of the ALF program:

```
STMT_ENTRY "KuiSnd5Z_1" "KuiSnd5Z_1_Step54" 0 ASSIGN "pZ" 0 16 INT 574 OR INT 585 OR INT
583 OR INT 579 OR INT 578 ;
```

According to this output annotation, the 16-bit "pZ" frame, which Bound-T uses to model the AVR `Z` pointer register, can take five values: in increasing order 574, 578, 579, 583, 585. These are indeed exactly the possible target addresses for the dynamic jump at this statement in the AVR subprogram `KuiSnd5Z`. That instruction is an AVR `ijmp` instruction which jumps to the address held in the `Z` pointer.

To illustrate the importance of the "no merge" option for SWEET's output annotations, the same analysis without this option produces an output annotation giving the interval 574 .. 585 as the possible values of "pZ". This interval has a total of 12 values: 7 false ones in addition to the 5 true ones.

Now consider what would happen if we would use the same procedure to analyse a dynamic branch which depends on the values of two or more storage cells. A SWEET annotation can only constrain variables separately; it cannot constrain combinations of variable values. In other words, the annotation domain is not *relational*. The same limitation applies to output-annotation specifications and output annotations. Therefore, if a dynamic branch depends on, say, two variables x and y, we can ask SWEET to produce the values of x as an output annotation, and the values of y as another output annotation, and then our best estimate of the possible target set is to take all combinations of a possible value of x and a possible value of y, even though many of these combinations are actually infeasible and produce false targets for the branch.

For example, the AVR 16-bit `Z` pointer register is actually composed of two 8-bit registers `r30` and `r31`, which form respectively the low and high octets of `Z`. We can let SWEET perform the same analysis of the AVR subprogram `KuiSnd5Z` as in the above examples, but now we use two output-annotation specifications to ask separately for the values or `r30` and `r31`, thus:

```
STMT_ENTRY "KuiSnd5Z_1" "KuiSnd5Z_1_Step54" 0 "r30" 0 8 || "r31" 0 8;
```

The resulting output annotation shows the possible values or `r30` and `r31` separately:

```
STMT_ENTRY "KuiSnd5Z_1" "KuiSnd5Z_1_Step54" 0 ASSIGN "r30" 0 8 INT 62 OR INT 66 OR INT
73 OR INT 67 OR INT 71 || "r31" 0 8 INT 2 ;
```

Thus, `r30` has five distinct possible values: 62, 66, 67, 71, 73, while `r31` has only one: 2, which was lucky because it means that there are still only five possible combinations, which are exactly the five true branch targets. For example, combining `r30` = 62 with `r31` = 2 gives the target address `Z` = $256 \cdot 2 + 62 = 574$. However, this good result happens only because the memory layout of the program is such that the five target address all like in the same block of 256 locations. If, instead, the memory layout happens to be such that the five target addresses are in different blocks, `r31` would have as many different values and the combination `r31:r30` would include several spurious addresses.

In fact, the abstract execution analysis (when done without merging) is mostly relational, because a state is a composite (tuple) of the values of all variables. The analysis only becomes non-relational if more than one variable has an abstracted value (because all combinations of concrete values of those variables are included in the state) or if merging is used (because the value-set of each variable is then merged and abstracted

independently of the simultaneous, related values of other variables). However, even if the abstract execution analysis is fully relational, the generation of output annotations hides the relations because a separate output annotation is generated for each variable, without considering the related values of other variables. Defining an (output) annotation syntax and generation procedure to preserve the relations discovered during abstract execution seems a worthwhile addition to SWEET.

4 Experimental Evaluation

4.1 Overview

This chapter explains how we evaluated and experimented with the Bound-T/SWEET combination to understand its abilities and limitations in resolving dynamic branches. Because the analysis of dynamic branches is quite important for practical machine-code analysis, Tidorum has had to put some effort into it, and therefore Bound-T contains a number of special analysis mechanisms which are aimed at this problem. On the other hand, SWEET's abstract execution is a powerful but general-purpose analysis method. Is SWEET's general-purpose analysis as powerful as the limited, specialized analyses in Bound-T? If not, could SWEET be improved to be competitive, and if so, how? Are there certain kinds of dynamic branches which the intrinsic analysis in Bound-T cannot resolve, but SWEET can? If so, can we characterize the cases that SWEET can solve better than Bound-T, perhaps even so specifically that Bound-T could choose automatically when to use its own analyses, and when to invoke SWEET instead? We hoped that the evaluation would answer such questions.

Most test programs for the evaluation were picked from Tidorum's test suite for Bound-T/AVR. Some test programs were written specifically for this evaluation (and then made part of the test suite). Table 1 describes the test programs in the order they are presented and discussed in the following subsections of this chapter.

Table 1: Evaluation Programs and Summary Analysis Results

Program	***Section***	***Description***	***Bound-T alone***	***Bound-T & SWEET***
`tp_avr_21`	4.2	A bottom-test loop which contains a switch-case statement implemented by an indexed dynamic jump into a table which contains static jumps to the cases. Two iterations of branch resolving are necessary, because the loop is discovered in the first iteration.	**Exact result**, but an **assertion** is needed to constrain the switch-case index (which is the loop index) to non-negative values. Problem is due to poor modelling of signed *vs*. unsigned operations and wrap-arounds.	**Exact result**.
`tp_c_2`, gcc	4.3	A switch-case statement implemented by loading the jump target address from a table in code memory.	**Fails**, because (1) the arithmetic analysis cannot model addressable memories, and (2) a specific pattern for this "load-address-from-table" idiom is not implemented in the AVR version of Bound-T.	**Fails**, because lack of congruence analysis in SWEET leads to important over-estimation of the octet pointer into the array, which causes huge over-estimation of the jump targets.
`tp_avr_6`	4.4	A switch-table and the corresponding handler subprogram. This is a simplified, artificial switch-table structure, from the example in [6].	**Fails**, because the switch handler in the program is an artificial one for which no detection is implemented in Bound-T. In principle, Bound-T's partial evaluation method **would work** here.	**Exact result**.

Program	*Section*	*Description*	*Bound-T alone*	*Bound-T & SWEET*
`tp_c_2 / KuiSnd5Z`, IAR	4.5	A sparse C switch-case, compiled to use the real switch-table form and real switch handler from IAR Systems.	**Exact result** using the partial-evaluation method, which however requires knowing the name of the switch handler and some-thing about its invocation idiom.	**Exact result** for the dynamic branch, but the **WCET is over-estimated**, being twice as large as for the partial evaluation method.
`tp_c_2 / KucDnd11Z`, IAR	4.6	A dense C switch-case, compiled to use the real switch-table form and real switch handler from IAR Systems.	**Exact result** using the partial-evalution method combined with arithmetic analysis of the indexing of the switch table.	**Fails,** because of weak-nesses in Bound-T's ALF generator and because SWEET's abs-tract execution is not relational and does not include congruence.
`tp_avr_7`	4.7	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". Jump-table entries are two addressing units long.	**Fails**, because the carry-out form the rotate operation is not well modelled in the arithmetic analysis.	**Fails**, because lack of congruence analysis in SWEET leads to im-portant over-estimation of the pointer into the jump-table, which causes huge over-esti-mation of the jump targets.
`tp_avr_8`	4.7	An indexed jump into a dense table of jumps, in which the 4-bit index is assembled from two 2-bit pieces using "shift" followed by "or". Jump-table entries are one addressing unit long.	**Exact result**, but depends on some unsound/unsafe assumptions in modelling left-shift operations.	**Exact result**, because the unit size of the jump-table entries makes congruence analysis unnecessary.

The rest of this chapter discusses each test program and its analysis in a fair amount of detail. Impatient readers may want to skip ahead to the summary and conclusions in chapter 5.

4.2 A Loop Containing a Switch-Case using a Jump Table (tp_avr_21, asm)

This test program, written in AVR assembly language for the purposes of this report, contains a loop which contains a switch-case structure, which is implemented by a dynamic jump with a computed target address. The loop has a *do-while* structure, that is, its termination test is at the "bottom" of the loop, after the switch-case structure. The loop counter runs from 15 to 19, which exactly matches the numbers of the cases in the switch.

Table 2 shows the AVR code and some description of the code. The three rightmost columns mark (by shading) the instructions discovered initially (before any resolution of the dynamic branch); in iteration 1 (in the first resolution of the dynamic branch); and in iteration 2 (in the second and final resolution of the dynamic branch).

This test program illustrates two general points. First, this program shows why an iterative analysis is necessary. Before the dynamic jump is resolved, only the initialization part of the loop-counting code has been discovered, and so the first, incomplete control-flow graph has no loop, and in particular no loop-termination test which could set bounds on the loop counter (register `r18`). Moreover, the switch-case structure has no default case, and the programmer knows that the loop counter is always in the range of the case numbers and therefore there is no explicit check of the table index that might indicate the full range of indices for the table. Thus, before the loop termination test is discovered, an analysis cannot place any bounds on the dynamic branch; all that can be deduced is that the target address resulting from the initial values (on first entering the loop) is a possible target.

Table 2: Test Program tp_avr_21

AVR assembly code	*Description*	*Discovered in iteration:* Init	1	2
`kases:`	Subprogram entry point.			
`ldi r18,15`	Initialize loop counter (`r18`) to 15.			
`kases_loop_head:`	Loop head (start of loop body).			
`mov r24,r18`	Compute the table index (`r24`) as			
`subi r24,15`	the loop counter (`r18`) minus 15.			
`ldi r30,lo8(pm(kases_table))`	Load the base address of the jump table			
`ldi r31,hi8(pm(kases_table))`	into `Z` = `r31:r30`.			
`ldi r25,0`	Zero-extend the index into `r25:r24`.			
`add r30,r24`	Add the index to the table base address,			
`adc r31,r25`	producing a pointer into the table.			
`ijmp`	Jump to that place, table[index].			
`kases_table:`	Here is the jump table.			
`rjmp kases_15`	Table index = 0, case = 15.			
`rjmp kases_16`	Table index = 1, case = 16.			
`rjmp kases_17`	Table index = 2, case = 17.			
`rjmp kases_18`	Table index = 3, case = 18.			
`rjmp kases_19`	Table index = 4, case = 19.			
`kases_19:`	Here are the cases (in reverse number			
`<code for case 19>`	order, just for fun).			
`rjmp kases_end`				
`kases_18:`				
`<code for case 18>`				
`rjmp kases_end`				
`kases_17:`				
`<code for case 17>`				
`rjmp kases_end`				
`kases_16:`				
`<code for case 16>`				
`rjmp kases_end`				
`kases_15:`				
`<code for case 15>`				
`kases_end:`	End of the switch-case statement.			
`inc r18`	Increment the loop counter.			
`cpi r18,20`	Compare to end value (20).			
`brlo kases_loop_head`	If counter < 20, repeat loop.			
`ret`	Terminate loop and return.			

Second, the fact that the table is dense (no "holes") and has elements (`rjmp` instructions) that are one addressing unit in length (16 bits in the AVR code memory) means that SWEET's interval-based domain is an exact abstraction of the possible pointers into the array. This test program was deliberately constructed to have this property, by making the dynamic jump a jump into the table itself, where the table elements are static jump instructions to the respective cases. While such tables of jumps do occur in real compiler-generated code, they are much rarer than tables which contain addresses or offsets and where the dynamically branching code first reads the address or offset from the table, as data, and then directly jumps to this address or offset, without chaining a dynamic jump and a static jump as in this test program. In this more common form, the tabulated addresses or offsets are usually not dense and are not precisely abstracted by intervals, as we will see in later test programs.

4.2.1 Analysis by Bound-T Alone

When Bound-T is applied to this program, without using SWEET, the first iteration of the analysis, when no loop is yet present in the control-flow graph, resolves the first target of the switch-case branch (to case 15) through the

constant-propagation analysis in Bound-T. The control-flow graph is then extended to include the first case of the switch, the loop termination test, and the back-edge to the loop head. The loop makes more variables vary, which makes constant propagation weaker, in fact too weak to produce any target addresses. Furthermore, the arithmetic analysis in Bound-T, although much more powerful than constant propagation gives only an upper bound on the target address of the dynamic branch. This happens partly because the loop termination test uses an unsigned comparison, while the Omega Calculator [9], which Bound-T uses for the arithmetic analysis, models only signed integers, and partly because Bound-T at present does not use the sign of the loop-counter step (here +1) to deduce that all values of the loop counter must be larger (for a positive step) or smaller (for a negative) step than the initial value. One reason why Bound-T does not use such reasoning is that the loop counter might wrap around and the reasoning would then be false. Consequently, the automatic analysis in Bound-T alone fails to resolve this dynamic branch.

While it is simple to write an assertion to constrain the loop index to non-negative values, which leads Bound-T to the exact result, this requires some insight into how Bound-T works and why it sometimes fails. An average user of Bound-T cannot be expected to write such an assertion. With this assertion, the second analysis of the dynamic branch gives an set of target addresses which includes all cases. Bound-T then extends the control-flow graph accordingly, and it is now complete.

4.2.2 Analysis by Bound-T and SWEET Combined

When Bound-T is used with SWEET (and without arithmetic analysis), the analysis proceeds in the same way as when Bound-T does it alone: in the first iteration, SWEET finds the branch to case 15; then Bound-T extends the control-flow graph accordingly; the second SWEET analysis finds also the other cases (16 .. 19); and Bound-T again extends the control-flow graph, which now becomes complete.

In both cases (Bound-T alone or with SWEET) the second iteration and the following extension of the control-flow graph introduces new execution paths to the dynamic branch. Bound-T therefore performs a third analysis, which finds no new targets for the branch, which shows that the result is stable.

4.3 Some C Switch-Case Statements with Address Tables (tp_c_2, gcc)

This test program, written in C, has several functions with different kinds of switch-case statements, using various types of index value, dense and sparse numberings, and increasing, decreasing, or random order of case numbers. This is a synthetic test program so the functions do not compute anything sensible.

When compiled with *gcc* for the Atmel AVR, only two of these C functions use a dynamic jump: the function `KucDnd11Z`, which has a switch-case statement with an index of type `unsigned char`, ten cases densely numbered 0..9, and a default case; and the function `KucDud11Z` which is the same except that the case numbers are in random order. Here is `KucDnd11Z`:

```
char KucDnd11Z (unsigned char index, char key)
{
   char result = '0';
   switch (index) {
      case 0: result = 'z';
              break;
      case 1: result += 1;
      case 2: if (result > 'a') result = 'b';
              break;
      case 3: return 'q';
      case 4: if (key == 'w') result = key - 1;
      case 5: result = key << 1;
              break;
      case 6: result += key;
              break;
      case 7: result -= key;
              break;
      case 8: if ((result & key) < key) result = '?';
              break;
      case 9: break;
      default:
         if (index < 77) return 'f';
```

```
    }
    return result;
}
```

The code generated by *gcc* for AVR first checks for the default case (`index` greater than 9) and otherwise (`index` in 0..9) loads the address of the corresponding case statement from a constant table in code space and then jumps to this address. Here is the main part of the AVR code:

```
KucDnd11Z:
   mov r30,r24       ; The parameter "index" is zero-extended from
   ldi r31,0         ; unsigned 8 bits (r24) to 16 bits (r31:r30).
   cpi r30,10        ; The parameter "index" (as 16 bits) is compared
   cpc r31,r1        ; to the constant 10 (note: r1 = 0 in gcc code).
   brcs in_range     ; Branch to in_range if index in 0..9.
   <code for default case>  ; Here index > 9.
   ret
in_range:            ; Here index is r31:r30, and in 0..9.
   subi r30,214      ; Add the base word address of the address
   sbci r31,255      ; table (by subtracting the negative).
   add r30,r30       ; Multiply the word address by two to
   adc r31,r31       ; produce an octet address for "lpm".
   lpm r0,Z+         ; Get the low octet of the target address.
   lpm r31,Z         ; Get the high octet of the target address.
   mov r30,r0        ; Put the whole address in Z = r31:r30.
   ijmp              ; Jump to the target address.
   <code for the other cases, referenced from the address table>
```

The addresses in the table are two octets long, but the `lpm` (Load Program Memory) instruction requires an octet address. The code therefore multiplies the word-address of the table element by 2 to compute the octet-address of the element.

4.3.1 Analysis by Bound-T Alone

Bound-T fails to resolve this dynamic branch because the target address is loaded from memory, and the arithmetic analysis with the Omega Calculator has no model for an addressable memory (or any other kind of indexable array). All variables in the Omega model are scalar integers.

For processors with better addressing capability than the AVR, Bound-T implements some pattern matching which detects jumps to addresses loaded from a table and generates a dedicated "boundable" object to represent such a dynamic jump. The boundable object depends on the array index expression, not on the loaded address value. When the arithmetic analysis finds bounds on the index (including congruence information), the resolution method for the boundable object fetches the corresponding addresses from the executable file and adds those edges to the control-flow graph. The address values themselves are never entered into the value analysis, only the index values are analysed there.

Detecting such load-address-from-table patterns is cumbersome for the AVR with its weak addressing modes and 8-bit operation width. Loading the indexed address from the table takes some seven AVR instructions, followed by the `ijmp` instruction which is the dynamic jump itself. There are several possible instruction sequences with the same effect, so any pattern detector would have to be quite flexible. However, this is certainly an important shortcoming of Bound-T/AVR and one that should be corrected, unless the use of SWEET solves the problem.

4.3.2 Analysis by Bound-T and SWEET Combined

ALF and SWEET are able to model addressable memories and can therefore find out the values loaded from the address table in this program (as long as Bound-T supplies the initial values as an ALF initialization). However, recall that the code multiplies the word address by two to get the octet address of the table entry. This has a nasty consequence: at present, SWEET does not implement congruence analysis, and therefore this multiplication makes SWEET overestimate the set of offsets into the address table. The true set consists of the even numbers in the range 0..18, but SWEET uses all numbers in this range, including the odd ones. When the offset is odd, the two-octet value (i.e. a putative but infeasible jump target) loaded from the table has the low octet of a real address in its high octet, and the high octet of a real address in its low octet. When SWEET computes the interval

which comprises all these "addresses", both the real and false values, the result is a huge over-estimate (-20 224.. 255 for `KucDnd11Z`, and -10 240..255 for `KucDud11Z`). Bound-T rejects these results as too loose, so the analysis fails to resolve these dynamic jumps.

Even if SWEET could use congruences to eliminate the odd offset values, the result would still be considerably over-estimated because SWEET would form an interval which contains all the addresses loaded from the table. These addresses are usually separated by the addresses of the instructions generated for the statements in each case. Thus the addresses form a *sparse* set rather than a contiguous interval. In other words, although the table indices are well (even exactly) represented by an interval (with congruence), the values of the table elements cannot be well represented by an interval, even with congruence.

Note that these over-estimates result from the appearance of an *abstracted* value for the case index: the interval 0..9. If the analysis were to consider each possible index in this interval separately, as if the code contained a loop stepping the index from 0 to 9, there would be no over-estimation (assuming that the "no merge" option is still in use). This improvement is observed in the switch-table examples, shown in later sections, because such loops occur naturally in the switch-table handler subprograms.

These examples suggest that SWEET could improve accuracy by enumerating abstracted states into the equivalent single-valued concrete states, processing the single-valued states, and then (if required) merging the results. Of course this risks combinatorial explosion, so it should be applied intelligently, perhaps only when the user so commands.

4.4 A Simple Switch-Table and Handler (tp_avr_6, asm)

In our classification (in section 2.2.2 above) of the three types of code generated for switch-case statements, *switch-tables* and their *handler subprograms* are the last and apparently most complex. For this evaluation, the test program `tp_avr_6` was hand-written in AVR assembly language to implement the deliberately simplified switch-table structure and the corresponding handler subprogram which were used as the running example for the description of Bound-T's partial-evaluation analysis method [6]. More complex, real examples of switch-tables occur in later test programs described in sections 4.5 and 4.6.

The simplified switch-table used in `tp_avr_6` is a list of entries of the form (*mask octet*, *match octet*, *target address*). An entry matches the 8-bit case-number if the logical bit-wise "and" of the case-number and the *mask* octet equals the *match* octet. The switch-table resides in AVR code memory. The switch handler is invoked by a jump instruction with the case-number value in register `r0` and a pointer to the switch-table in register `Z`. The switch handler executes a loop which traverses the table and jumps to the target address of the first matching entry. As usual in AVR code, this dynamic jump is implemented with an `ijmp` instruction, which jumps to the address in the `Z` register.

The switch-case example in the test program has four cases numbered 4, 8, 9, 11, and a default case. The default case is represented by the final entry in the switch-table with *mask* and *match* both zero.

4.4.1 Analysis by Bound-T Alone

The numerical analyses in Bound-T cannot resolve this dynamic jump, because the target address is loaded from a table rather than computed numerically, and there is no specific pattern-matching analysis for this case in the AVR version of Bound-T (as explained in section 4.3).

The dynamic jump could be resolved with the partial-evaluation method [6], but that requires detection and special handling of the invocation of the switch handler. As this particular handler is just an artificial example and is not used by any real compiler, we have not implemented detection of this handler in Bound-T. In summary, Bound-T cannot now resolve this dynamic branch, but can resolve similar real cases which are generated by real compilers. For an example, see section 4.5.

4.4.2 Analysis by Bound-T and SWEET Combined

The combination of Bound-T and SWEET *can* resolve the branch. The abstract execution method works in this case with single, concrete values: the base address of the switch-table is a static constant; the indices into the table are generated one at a time by the loop in the switch handler; and the "no merge" option to SWEET prevents their abstraction into intervals. Therefore, there is no over-estimation, and the dynamic branch is resolved precisely to its five cases.

4.5 A Complex Switch-Table: Sparse Form (tp_c_2 / KuiSnd5Z, IAR)

The Bound-T test program `tp_c_2` contains several C functions with various kinds of switch-case statements. In the function `KuiSnd5Z`, the switch-case statement has an index of type `unsigned int`, four sparsely numbered cases, and a default case:

```
char KuiSnd5Z (unsigned int index)
{
   char result = '0';
   switch (index) {
      case 0:
         result = 'z';
         break;
      case 16:
         result += 1;
      case 33:
         if (result > 'a') result = 'b';
         break;
      case 95:
         return 'q';
      default:
         if (index < 77) return 'f';
   }

   return result;
}
```

When this function is compiled with the IAR compiler, using the compiler option `-fst0`, the generated code uses the IAR compiler's switch-table format and the switch handler `?SV_SWITCH_L06`. Bound-T can analyse such code with the partial-evaluation method [6]. The IAR switch-table format and handler routine are much more complex than the simplified example shown in [6] and tested in `tp_avr_6`. It is interesting to see if SWEET can analyse this complex case.

4.5.1 Analysis by Bound-T Alone

Bound-T recognizes the jump to `?SV_SWITCH_L06` as the invocation of an IAR switch handler and begins the partial evaluation of the handler, as described in [6]. In brief, this means that Bound-T executes the AVR code in the switch handler, instruction by instruction, but the execution is partly concrete and partly abstract. The only concrete data are the address to the switch-table, which is held in the `Z` register on entry to the handler, and all data that are loaded from known addresses in the switch-table. The values in the switch-table are statically known because the switch-table is located in the AVR program memory, and the contents of the program memory are part of the file given to Bound-T, which contains the program under analysis. All other data are abstracted to a single abstract value "unknown". This includes in particular the value of the switch index expression.

The partial evaluation leaves a "trail", or residual program, which consists of steps and edges in the control-flow graph, added to the graph one by one when the instructions are evaluated and control transitions are taken.

When the partial evaluation encounters a dynamic branch in the switch handler — usually an `ijmp` instruction — it attempts to resolve the branch using the concrete part of the evaluation state. In the case of `ijmp`, which jumps to the address held in the `Z` register, resolution succeeds if and only if the `Z` register has a concrete value at this point in the evaluation. That is always the case in this switch handler, because the handler loads the target address from the switch-table into `Z` before executing `ijmp`.

Because dynamic branches must be resolved in a context-independent way in Bound-, the switch handler subprogram must be "integrated" (in-lined) into the the control-flow graph of the subprogram which contains the switch-case statement, here `KuiSnd5Z`. This in fact happens naturally, because the compiler-generated code invokes the handler by a jump instruction, not a call instruction. The handler has no use for a return address because it exits to some address that it takes from the switch-table.

This switch handler uses a number of "helper" subprograms to perform frequent small jobs such as loading an octet or a word of data from the switch-table. Because Bound-T models the computational effects of subprogram calls quite imprecisely, for the partial evaluation to work these subprograms must also be integrated

into the caller's control-flow graph. This integration happens automatically when Bound-T is partially evaluating the handler.

For the subprogram KuiSnd5Z, the partial evaluation of ?SV_SWITCH_L06 encounters eight ijmp statements in this switch handler (rather, it encounters the same ijmp statement in eight different evaluation states). Four of these resolve to the default case of the switch statement in KuiSnd5Z, and the four others to each of the four non-default cases. The final control-flow graph of KuiSnd5Z, with the switch-case statement fully resolved, contains 221 steps (instructions) and 229 edges (control-flow transitions including sequential flow), mostly coming from the code of the integrated switch handler ?SV_SWITCH_L06 and its helper subprograms.

Note that this partial evaluation method evaluates and includes in the control-flow graph only those parts of the switch handler which are reachable (feasible) under the current switch-table contents. In other words, if the value of the condition of a branch in the switch handler can be computed from the concrete part of the evaluation state then only the selected transition (branch taken if true, not taken if false) is evaluated and included in the control-flow graph. In this respect Bound-T's partial evaluation resembles SWEET's abstract execution which also discards infeasible execution states.

When the partial evaluation evaluates an instruction and adds the corresponding step to the control-flow graph, the step is labelled and identified by the evaluation state, in addition to the instruction address. Therefore, if the same instruction is evaluated in several different contexts, it will be represented by as many steps in the control-flow graph. This is a somewhat analogous to the way in which SWEET's abstract execution keeps its execution states separate when the "no merge" option is used.

The partial evaluation completes when all execution paths have ended in ijmp statements, which have been resolved. Thus, the loop in the switch handler is executed or evaluated, iteration by iteration, until it terminates. The final control-flow graph has no loop, so it is easy for Bound-T to calculate a WCET bound: 284 cycles.

4.5.2 Analysis by Bound-T and SWEET Combined

To test the combined Bound-T + SWEET analysis on this example, Bound-T's partial-evaluation method and arithmetic analysis were both disabled with command-line options. Also, to make this experiment as similar as possible to the analysis that Bound-T performs alone, assertions were used to make Bound-T integrate the "helper" routines into the control-flow graph, instead of modelling them as called subprograms. The switch handler itself is automatically integrated because it is invoked with a jump instead of a call.

In this analysis, where Bound-T constructs the control-flow graph in the ordinary way, and not with partial evaluation, the control-flow graph contains all the code of the switch handler, not just the part that is feasible for the current switch-table. The resulting control-flow graph contains two ijmp instructions; the partial evaluation found only one of them. Here, too, Bound-T decides that the other ijmp is unreachable, but here this decision comes from constant-propagation analysis of the control-flow graph, not from partial evaluation (the two analyses are of course very similar). The other ijmp comes into use for switch-tables with different contents as will be explained in section 4.6.

Figure 2 shows the control-flow graph as Bound-T sees it after the dynamic branches are resolved. The dotted edges show the infeasible paths (unreachable code); the bold solid edges show the worst-case path; the other (thin) solid edges show other feasible paths (well, feasible as far as Bound-T can determine).

When Bound-T translates the control-flow graph to ALF, the unreachable ijmp instruction (and the other unreachable code) is left out. SWEET's analysis of the ALF program, with its one unresolved branch, discovers the exact set of target addresses for this branch. The final control-flow graph of KuiSnd5Z, with the switch-case statement fully resolved, contains 167 steps, of which 29 are infeasible, and 184 edges of which 34 are infeasible. Again, most of these steps and edges come from the code of the integrated switch handler ?SV_SWITCH_L06 and its helper subprograms. The number of steps and edges is different from their number in Bound-T's partial evaluation analysis, because here the control-flow graph does not separate steps (instructions) by their evaluation state, only by their machine address.

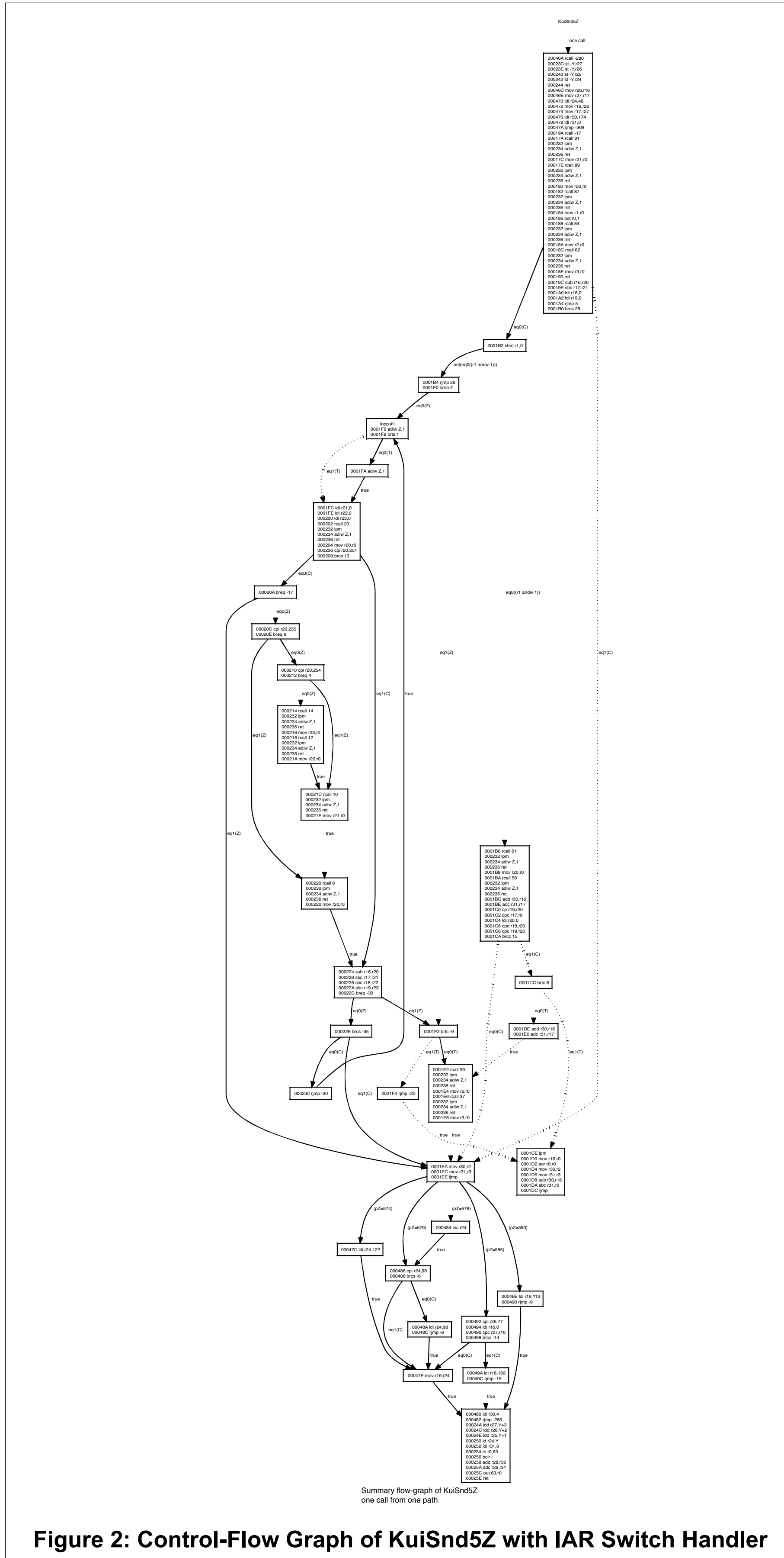

Figure 2: Control-Flow Graph of KuiSnd5Z with IAR Switch Handler

In this analysis with SWEET, Bound-T cannot produce a WCET bound, because it cannot find iteration bounds on the loop in the switch handler. In the absence of partial evaluation, this loop remains as a loop in the final control-flow graph. The problem is that this loop is not a "counted" loop but a loop terminated by a logical condition (matching switch entry found). SWEET's abstract execution has found an iteration bound for this loop (else SWEET would not terminate), but at present this bound is not conveyed from SWEET to Bound-T in our prototype implementation. If we assert this loop-bound (4 iterations) for Bound-T, the resulting WCET bound is 577 cycles. This is more than twice the WCET bound of 284 cycles that Bound-T computed using its partial-evaluation method. The likely reason is that the larger bound is calculated assuming that each loop iteration takes the worst-case path through the loop-body, while the smaller bound is the sum of the WCETs of the actual execution paths in each iteration. These paths are discovered by the partial evaluation method, based on the actual contents of the switch-table, when the evaluation completely unrolls the loop into the control-flow graph.

However, this WCET over-estimation is more due to an incomplete transfer of SWEET's analysis results to Bound-T than to a weakness in SWEET's analysis. The abstract execution in SWEET does find the actual execution path in each iteration of the loop, but in our current prototype that information is not transferred to Bound-T. One way to recover the smaller WCET bound from SWEET would be to define the cumulated execution time as a program variable and let the abstract execution find bounds on this variable, as suggested in another context [10].

4.6 A Complex Switch-Table: Dense Form (tp_c_2 / KucDnd11Z, IAR)

Among the subprograms in the test program tp_c_2 is `KucDnd11Z` which contains a switch-case statement using an `unsigned char` index expression and having 10 cases numbered densely from 0 to 9 plus a default case. When this function is compiled with the IAR compiler, using the compiler option `-fst0`, the generated code uses the IAR compiler's switch-table format and the switch handler `?SV_SWITCH_L06`, as in the case of the sparsely numbered switch statement discussed in section 4.5. However, for the densely numbered switch-case statement in `KucDnd11Z`, the switch-table contents are of course different and in fact drive the switch handler to use its other `ijmp` instruction, which was not used at all for the sparse form. Indeed, the IAR switch-table format supports also densely numbered sets of cases (perhaps as a subset of the cases in a switch-case statement which has both sparsely and densely numbered parts), and this second `ijmp` instruction implements this.

For both the "sparse" and the "dense" `ijmp` instruction the target address is loaded from the switch-table. However, for the "sparse" `ijmp` the address is loaded form a place which is concrete (known) to the partial evaluator, and therefore the evaluator also knows the target address, while for the "dense" `ijmp` the place in the table is computed using the value of the switch-case index expression, which is not concretely known to the partial evaluator. We explain below how Bound-T handles this.

4.6.1 Analysis by Bound-T Alone

Bound-T detects the invocation of the IAR switch handler and starts its partial evaluation. If this evaluation would do only what it does when the case-numbering is sparse, it would encounter the `ijmp` instruction for the "dense" numbering with an evaluation state which does not determine the concrete value of register `Z`, and would fail to resolve the jump. Some further or other analysis is therefore needed; in Bound-T, this means the arithmetic analysis must be used. However, activating the arithmetic analysis at the `ijmp` instruction, to find the possible target addresses as the possible values of the `Z` register, does not work, because `Z` is loaded before the `ijmp` from some dynamically determined place in the switch-table, and the arithmetic analysis cannot model such a load. To get around this problem Bound-T uses the same approach as for a simple dense address array: it applies the arithmetic analysis to find the possible table indices, and then reads the corresponding possible values from the table content (which is known, because the table is in program memory).

This is implemented as follows: during partial evaluation of a switch handler, if Bound-T encounters an instruction which loads data from the program memory (in the AVR that is an `lpm` or `elpm` instruction), and the concrete part of the evaluation state does not define the concrete source place (memory address) for the load, Bound-T creates a special kind of boundable edge *from* the step *preceding* this load instruction. This boundable edge is subjected to the arithmetic analysis, but unlike most other boundable edges the goal is not to find the target-instruction addresses, but to find the possible values of the *data* address for the load. In the AVR, this is again the value of the `Z` register at the load instruction. If this analysis discovers the possible load addresses, Bound-T resolves this special boundable edge into one actual flow-graph edge for each possible load address. All

these edges go to the load instruction, but they have different conditions and result in different evaluation states which contain different concrete values of the load address. In effect this "forks" the partial evaluation into as many parallel paths, and sets the initial concrete part of the evaluation state of each path to hold the corresponding load address. As the partial evaluation proceeds along these paths it uses these load addresses to read the concrete data from the program memory. The loaded data later becomes the target address in the "dense" `ijmp` instruction, and so the partial evaluation can resolve this `ijmp`.

Perhaps a concrete example is necessary to illustrate this process. Table 3 shows an outline of the code of the IAR switch handler for the dense case-numbering.

Table 3: IAR Switch Handler Outline for Dense Case Numbering

Address (hex)	*AVR assembly code*	*Description*
	KucDnd11Z:	The root subprogram for our analysis.
...	...	Unimportant code elided.
302	mov r16,r25	Sets the switch index into `r16`.
304	ldi r30,40	Sets `Z` = `r31:r30` to point to the switch-table.
306	ldi r31,0	
308	rjmp ?SV_SWITCH_L06	Invokes the switch handler. Bound-T starts the partial evaluation with `Z` = 40 as the concrete part of the evaluation state. Note that the switch index in `r16` is abstract (not concrete).
	?SV_SWITCH_L06:	The switch handler.
192	...	Code (which we elide for clarity) which reads data from the switch-table (using the `GetByte` subprogram, see below), notices that this switch has dense case numbering, and computes the address in the switch-table which contains the address of the case to be executed. The former address is computed into the `Z` register in an affine way from switch-table data (concretely known to the partial evaluator) and the switch index (`r16`) which is not concretely known.
1E2	rcall GetByte	Calls a subprogram which reads an octet from the switch-table, at address `Z`, returns it in `r0`, and increments `Z`. The value read is the low octet of the case address.
1E4	mov r2,r0	Saves the value read in `r2`.
1E6	rcall GetByte	Reads the high octet of the case address into `r0` (and increments `Z`, but the incremented value is not used).
1E8	mov r3,r0	This and the two following instructions collect the case address into the `Z` register (`r31:r30`).
1EA	mov r30,r2	
1EC	mov r31,r3	
1EE	ijmp	Jumps to the case address in `Z`, leaving the switch handler.
???	???	The first instruction of the selected case, wherever it is.
	GetByte:	Subprogram to read one octet from the switch-table.
232	lpm	Loads the current octet (at `Z`) from the switch-table into register `r0`.
234	adiw Z,1	Advances `Z` to point to the next byte.
236	ret	

When Bound-T partially evaluates this code, all calls of the `GetByte` subprogram are automatically integrated in the main control-flow graph. In other words, as you read the table above, you should think of the instruction `rcall GetByte` as a macro call which copies the three instructions in `GetByte` into the control-flow graph of `KucDnd11Z`.

The partial evauation finds that all loads from the switch-table use concrete addresses and yield concrete data, up to the call of `GetByte` at address 1E2. At this point, the partial evaluation has one active state, located at this instruction. In this state, the `Z` register contains an abstract value, which means that the `lpm` instruction in `GetByte` at address 232 does not yield concrete data. As explained above, Bound-T therefore creates a boundable

edge which originates at the preceding instruction (1E2: rcall GetByte) and uses arithmetic analysis to compute the possible values of the load address, which is the Z register.

The arithmetic analysis yields the value-set of the ten even integers between 46 and 64, inclusive. The resolving operation of the boundable edge therefore adds ten new static edges into the control-flow graph, all going from the instruction (1E2: rcall GetByte) to the instruction (232: lpm), but each with a different evaluation state at the target instruction, namely a state which assigns one of 46, 48, ..., 62, 64 as the concrete value of Z.

The partial evaluation resumes from these edges which represent ten valid but different evaluation states. In each of these states, the value of Z for the instruction (232: lpm) is concretely known and therefore the partial evaluation of this instruction returns a concrete value from the switch-table. Moreover, when evaluation then reaches the next call of GetByte at address 1E6, register Z still has a concrete value, and so this GetByte can be evaluated as usual and also returns a concrete value. All ten partial evaluation "threads" thus reach the ijmp instruction at address 1EE with a concrete Z-value, which means that the ijmp is resolved in each thread (to a unique target address in that thread). Overall, the ijmp instruction is resolved into ten possible target addresses.

Figure 3 on page 26 shows the completed control-flow graph of KucDnd11Z, as Bound-T constructs it without help from SWEET. This includes the integrated code from the IAR switch handler and its helper subprograms such as GetByte. The crucial point is the basic block which ends with the instruction shown as "0001E2 rcall 39", which is the rcall GetByte instruction at which Z is (at first) abstract. The ten edges leaving this basic block represent the ten threads of partial evaluation, each with its own concrete value of Z selected from the ten possible values of Z which the arithmetic analysis discovered. The edges are labeled with their Z values, shown as "pZ" to distinguish the 16-bit Z pointer register from the 1-bit condition flag Z.

4.6.2 Analysis by Bound-T and SWEET Combined

To test the combined Bound-T + SWEET analysis on this example, Bound-T's partial-evaluation method and arithmetic analysis were both disabled with command-line options. Also, to make this experiment as similar as possible to the analysis that Bound-T performs alone, assertions were used to make Bound-T integrate the "helper" routines into the control-flow graph, instead of modelling them as called subprograms. The switch handler itself is automatically integrated because it is invoked with a jump instead of a call. Furthermore, we disabled Bound-T's use of constant-propagation results for detecting infeasible paths, because we wanted to evaluate SWEET's ability to detect such paths. We left enabled the refinement of arithmetic effects by constant-propagation, so this result of Bound-T's own analysis enters into and simplifies the ALF code given to SWEET.

(We record as an aside that if we disable Bound-T's constant-propagation analysis entirely, and so translate the raw, unrefined AVR code into ALF, then SWEET's abstract execution of the ALF code does not converge, and SWEET complains of running into out-of-bounds ALF memory accesses. The reason seems to be that under these conditions SWEET does not detect that the switch-handler code for the sparse case-numbering is unreachable, and therefore abstractly executes the loop which traverses the "sparse" switch-table, but does not detect the termination of this loop and so runs off the defined program-memory address range. However, this explanation is conjectural and investigation is pending; the reason could be a poor or too approximate translation to ALF. We now continue discussing how SWEET works when constant-propagation refinements are allowed for the computational effects of steps/instructions but not for edge conditions. Under these conditions, SWEET's abstract execution does terminate.)

SWEET agrees with Bound-T that the ijmp instruction dedicated to the sparse case-numbering is unreachable in this example program. SWEET also detects that the switch-table-traversing loop for the sparse case-numbering is unreachable (at least, no loop-bound flow-facts are generated under the --flow-facts option).

However, SWEET's abstract execution cannot find good bounds on the target address for the reachable ijmp instruction for dense case-numbering. There are several reasons for this, all stemming from a central but difficult sequence of instructions in the switch handler. Table 4 shows and describes this sequence.

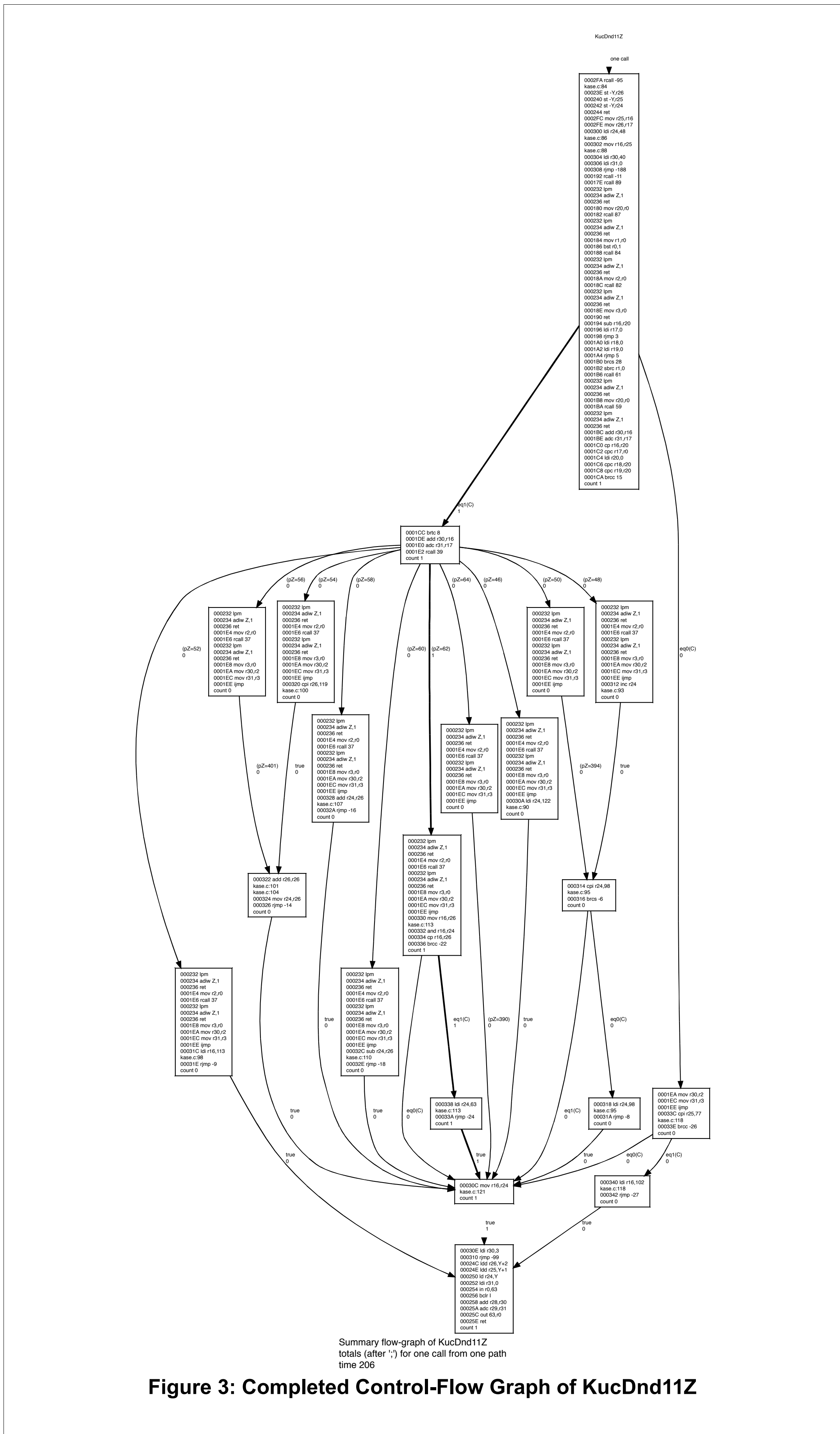

Figure 3: Completed Control-Flow Graph of KucDnd11Z

Table 4: IAR Switch Handler Core for Dense Case Numbering

Address (hex)	*AVR assembly code*	*Description*
		Here, register Z points into the switch-table, at the location holding the code address of the first case. Registers r17:r16 contain the switch-index with the number of the first case subtracted, so r17:r16 = 0 for the first case, 1 for the second, and so on. Registers r0:r20 (note the non-standard pairing) contain the number of cases, which is 10 for KucDnd11Z.
1BC 1BE	add r30,r16 adc r31,r17	Start computing the place in the switch-table which contains the case address, by adding the zero-based switch index (r17:r16) to Z. If the switch-table entries were one octet long, this would give the address of the entry. However, here the entries are two octets long, so this addition is not enough; see address 1DE below.
1C0 1C2	cp r16,r20 cpc r17,r0	Compare the zero-based switch index (r17:r16) to the total number of cases (r0:r21). The carry flag, C, shows the result.
1C4 1C6 1C8	ldi r20,0 cpc r18,r20 cpc r19,r20	Extend the comparison from 16 bits to 32 bits. For a 16-bit switch index, as used in KucDnd11Z, this code has no significant effect, because r18 and r19 are zero. Most likely the IAR switch handler has other entry points for switch-case statements using 32-bit indices, and those entry points also use this code.
1CA	brcc not_in_range	Branches to the default-case code if the zero-based switch index is greater or equal to the number of cases, in other words if no case has this index number. Here we assume that some case does match this index, so we assume that the branch is not taken.
1CC	brtc <1DE>	We do not fully understand the function of this branch, but the data in the switch-table for KucDnd11Z force it to be taken.
1DE 1E0	add r30,r16 adc r31,r17	Finish computing the place in the switch-table which contains the case address, by again adding the zero-based switch index (r17:r16) to Z. The switch index must be added twice because the table entries are two octets long.
1E2	... (see Table 3)	Here, Z points into the switch-table, at the entry holding the 16-bit code address of the selected case. As shown in Table 3, the switch-handler code proceeds to load the address into Z (using two calls of GetByte) and then executes ijmp.

The code shown in Table 4 poses the following problems for Bound-T and SWEET:

1. The AVR version of Bound-T identifies "chains" of 8-bit operations which implement 16-bit computations. For example, the cp-cpc pair at address 1C2 effectively compares the 16-bit value r17:r16 to the 16-bit value r0:r20, with the result in the final value of the carry flag C. However, this chaining currently stops at 16 bits (for no very good reason). Therefore, Bound-T does not understand that the next three instructions, starting at address 1C4, extend the comparison to 32 bits. The carry flag, which after the first cpc shows the result of the 16-bit comparison, is further modified by the other two cpc instructions. This means that Bound-T is not able to "in-line" or substitute the relational expression "r17:r16 < r0:r20" into the condition for the branch at address 1CA. Therefore, the ALF {switch} for this branch depends on the 1-bit C flag, not on the relational expression.

2. The above shortcoming of Bound-T's analysis and ALF translation causes the first SWEET problem. Because the {switch} modelling the branch at address 1CA uses the logical value of a 1-bit flag, instead of a relation between variables, SWEET cannot infer restrictions on the values of the variables for the {switch} targets. Thus, although the {switch} target corresponding to the address 1CC is reached only if r17:r16 is in the range 0..9, SWEET can only deduce the bounds 0..255 (the same as before the {switch}). The addition at addresses 1DE..1E0 then leads to a severe over-estimation of the possible values of Z.

3. The second SWEET problem stems from the peculiar structure of the code, where the first addition of r17:r16 to Z occurs at addresses 1BC..1BE *before* the comparison of r17:r16 to 10. This means that even if SWEET could deduce the correct bounds 0..9 for r17:r16 *after* the branch at 1CA, SWEET's abstract

execution would still over-estimate the range of Z because it would not *retroactively* apply these bounds to the addition at 1BC..1BE which uses the same value of r17:r16. At that addition, SWEET's bounds on r17:r16 are only 0..255; these are indeed the best bounds that can be inferred from the code executed before the addition, but they lead to a severe over-estimation of the range of Z on the way to the ijmp. A relational value-analysis, such as SWEET's polyhedral value-analysis, should be able to make this connection between the bounds on the same value of r17:r16 at different points in the code. Unfortunately Tidorum did not have the polyhedral analysis available in their installation of SWEET.

4. Finally, even if SWEET could infer the correct bounds 0..9 for r17:r16 at both additions of this value to Z, and could thereby infer the correct bounds (46..64) on Z, the current lack of congruence analysis in SWEET would lead to an over-estimation of the value-set of Z, because the fact that the total added value is an even number would escape SWEET. When the code uses Z to load the target address for the ijmp from the switch-table, the abstract execution would use all values (odd and even) between the lower and upper bound on Z. This would include false target addresses with mixed-up combinations of low and high octets. By simulating this analysis with a SWEET annotation that restricts r16 to 0..9 at the start of the subprogram we found that SWEET would bound the target address to the interval -31232..511, which Bound-T would reject as far too loose to be useful as the resolution of a dynamic branch.

Thus, the failure of this analysis results from defects or shortcomings both in Bound-T and in SWEET. Can these defects be corrected? We answer point by point in the same order as above:

1. Extending Bound-T's narrow-to-wide operation-chaining to any width of values and operations is easy in principle. However, in practice chaining should be a general, target-independent function, not (as now) specific to the AVR version of Bound-T, in which chaining was implemented as a pilot experiment with a view to later implementation in a general, target-independent fashion (this is also the reason why the AVR chaining implementation stops at 16 bits). A general implementation of chaining requires a more general concept of hierarchical or overlapping storage cells, which Tidorum has been pursuing for a considerable time without practical result. Hope remains, however.

2. The SWEET development group at Mälardalen University has discussed implementing in SWEET itself some kind of in-lining of {switch} conditions so that uses of condition flags are replaced, for the analysis, by the numerical relation which defined the condition flag. This should improve analyses using linear models, such as polyhedra, because the setting of a condition flag according to a numerical relation is not a linear operation, even if the defining relation is linear in its variables. However, no design or implementation plan exists yet.

3. As already said, a relational domain should solve this problem. However, SWEET does not currently implement abstract execution with a relational domain, and such a combination might be quite expensive in analysis time. Standard relational domains such as polyhedra have problems in the modelling of variables with a bounded number of bits where the analysis should cover phenomena such as wrap-around, overflow, underflow and alternative signed and unsigned views of variables. These phenomena can be modelled with "bounded polyhedra" [11] but the implementation of bounded polyhedra in SWEET is not quite ready for routine use.

4. Congruence analysis should not be hard to implement in SWEET. It was implemented in an earlier version of SWEET which used the NIC language instead of ALF.

4.7 Complex Boolean Address Computation (tp_avr_7/8, asm)

This test program, mostly written in assembly language, tests the ability of Bound-T, assisted by SWEET, to analyse the complex dynamic jump in a library function called ?C?COPY, originally part of the C library of the Keil C compiler for the Intel-8051 processor. The inability of Bound-T alone to resolve this jump was probably an important reason for a recent evaluator of Bound-T/8051 to decide against purchase.

For testing with the Bound-T and SWEET combination, the function ?C?COPY was translated into AVR assembly language with as little logical or structural changes as possible.

Some explanation of why this function has a complex dynamic jump may help to motivate this example. The Intel-8051 processor architecture has four different memory address spaces that can hold data that is accessed through a C-language pointer variable (there also also other address spaces, not relevant to C pointers):

- the *internal* data memory, known as I-data, usually 256 bytes in size (8-bit address)
- the *external* data memory, known as X-data, typically 64 kilo-octets in size (16-bit address)
- the *paged* address space, known as P-data, usually a 256-octet window (8-bit address) into the X-data, with the window base address (multiple of 256) defined by another 8-bit register (an output port)
- the *code* memory, which is usually read-only and 64 kilo-octets in size (16-bit address).

Note that the program must use different 8051 instructions to read/write data for each memory space; there is no general instruction that could read/write in any space.

C compilers targeting the 8051 usually support extended source-language keywords by which a certain variable can be specified to lie in one of these memory spaces, and a certain pointer can be specified to point into one the spaces. However, in order to conform to the C standard the compilers also support "generic" pointers which can point into any of the four spaces; which space is referenced is determined dynamically at run time. The run-time representation of a generic pointer typically consists of one octet which gives the memory space (only a 2-bit code is required, of course) and two octets which give the address (although one octet would be enough for some spaces).

Now consider a standard C function such as `memcpy(d,s,n)`, which copies *n* bytes from the area pointed to by *s* to the area pointed to by *d*. As this is a standard function, the pointers *s* and *d* are standard or generic C pointers, and can point to any of the four memory spaces. The function must be prepared to copy, for example, from the code memory to the internal data memory, or from the paged address space to the external data memory, or any other combination of memory spaces except those where *d* points to code memory (assuming that code memory is read-only). Different combinations of instructions must be used for each combination of source and destination memory space. Moreover, the loop which counts *n* bytes must also be different because some spaces can make do with an 8-bit counter in a single 8-bit register, while others require a 16-bit counter composed of two 8-bit registers.

The function `?C?COPY` is the core of `memcpy` and contains 12 different cases corresponding to the 12 memory-space combinations with a choice of four source spaces and three destination spaces. The function chooses the correct case for the actual *s* and *d* arguments by taking the 2 bits of memory-space code from each of *s* and *d* and concatenating them into a 4-bit combined code which it then uses as the offset in a dynamic jump instruction. Thus, the dynamic jump implements a dense switch with 16 cases. (Four of these cases, where *d* points to code memory, are actually illegal, but that is irrelevant for us.)

Surprisingly, for the first version of this test program Bound-T/AVR *was* able to resolve the dynamic jump even if Bound-T/8051 was not. The AVR instructions chosen for the bit-manipulation that computes the 0..15 offset to the jump base address happened to fit well in Bound-T's analysis. Other equally plausible choices of instructions make Bound-T's analysis fail, as explained below in section 4.7.1.

Accordingly, we created two versions of this test program. The program `tp_avr_7` is most similar to the original 8051 function. It uses a rotate instruction instead of a left-shift, and this makes Bound-T fail. Its switch entries are two addressing units long, which makes SWEET fail to produce an exact result (due to lack of congruence analysis). In contrast, `tp_avr_8` uses left-shift, not rotate, and so Bound-T succeeds; and this program also has switch entries of length one addressing unit, which makes SWEET produce the exact result.

4.7.1 Analysis by Bound-T Alone and Combined with SWEET

For this test case we discuss Bound-T's analysis and SWEET's analysis together, because we look in detail into the AVR code to be analysed and consider how Bound-T and SWEET analyse the critical instructions.

Table 5 shows the computation of the jump offset in `tp_avr_7` and how Bound-T models it. The "Address" column uses octet addresses (per *gcc* convention), although the AVR code-memory address unit is 16 bits. The five significant areas that are highlighted with bold and a yellow background are discussed after the table.

Table 5: Computation of Memory-Space Combination Index in ?C?COPY

Address (hex)	*AVR assembly code*	*Description*
		On entry to this function, the source-space code is in `r16`, with an offset of -2 (that is, the code is (0..3) - 2, and the destination code is in `r17` with the same offset. Note that this offset causes wrap-around if these registers are taken as unsigned 8-bit numbers. (We do not understand why such an offset is used in the original function, but we do the same for realism.)
9C..B0	...	Elided code that checks if the length parameter is positive and whether the length is a multiple of 256 (which influences the copy-loop counters). The code for checking the memory-space codes and combining them into the switch index follows:
B2 B4	`ldi r30,2` `ldi r31,4`	Load some helpful constants: the memory-space code offset (2) and the number of spaces (4).
B6 B8	`mov r0,r17` `add r0,r30`	Remove the offset from the destination memory-space code by adding 2 to `r17`. The result is in `r0` and should be in 0..3.
BA **BC**	**`cp r0,r31`** **`brcc -10`**	**Check that the destination memory-space code is in 0..3, and otherwise return (`brcc`) with an error code.**
BE C0	`mov r1,r16` `add r1,r30`	Remove the offset from the source memory-space code in the same way; result in `r1`.
C2 **C4**	**`cp r1,r31`** **`brcc -14`**	**Check that the source memory-space code is in 0..3, as above.**
		At this point, the destination memory-space code (0..3) is in `r0` and the source memory-space code (0..3) is in `r1`. Now we combine them into a four-bit code (0..15):
C6 **C8** **CA**	**`bclr C`** **`rol r1`** **`rol r1`**	**Shift `r1` two bit-places to the left, but using the "rotate through carry flag" instruction `rol r1`. This instruction rotates the 9-bit combination `C:r1` left by one bit position. It is equivalent to the instruciton `adc r1,r1` (add `r1` to `r1` including the carry flag) and in fact has the same encoding.**
CC	`or r0,r1`	Merge the two memory-space codes into a single 4-bit code (0..15) in `r0`. This is the switch index.
CE	**`adc r0,r0`**	**Multiply the switch index by two, because the jump-table entries are two addressing units (two AVR instruction words) each. The result is the jump-table case-offset, an even number in the range 0..30.**
D0	`eor r1,r1`	Zero-extend `r0` from 8 bits to 16 bits in `r1:r0`.
D2 D4	`ldi r30,110` `ldi r31,0`	Load the base address (word address 110, octet address 220 = hex DC) of the jump-table into `Z` = `r31:r30`.
D6 D8	`add r30,r0` `adc r31,r1`	Add the case-offset (r1:r0) to the base address (r31:r30).
DA	**`ijmp`**	**Jump to that place in the jump-table.**
		Here is the jump table itself:
DC	`rjmp CCOPY00`	Case 0 (source and destination space-code both 0).
DE	`nop`	A spacer inserted to make the jump-table entries two addressing units long as in the Intel-8051.
C0	`rjmp CCOPY01`	Case 1 (source space 0, destination space 1).
C2	`nop`	Spacer as above.
...	...	And so on, until:
118	`rjmp CCOPY33`	Case 15 (source and destination space code both 3).
...	...	The copying code itself (`CCOPY00` .. `CCOPY33`) is elided.

The significant, highlighted areas of the code are the following:

- The dynamic branch is the `ijmp` instruction at address DA. The analysis aims to discover the set of possible values of the `Z` register at this point.

- The two comparison instructions (`cp`) at addresses BA and C2 and their following conditional branches (`brcc`) tell the analysis that the memory-space codes are in the range 0..3. Both Bound-T and SWEET deduce this range, but for Bound-T this deduction is fragile because it makes unsound assumptions regarding signed and unsigned views of the values in storage cells.
- The three instructions starting at address C6 shift the two bits of destination-space code two positions left in preparation for combining them with the source-space code. This is the part where Bound-T can either fail or succeed depending on the specific instructions chosen.
  - The Intel-8051 processor has no shift instructions, only rotate instructions. The original `?C?CASE` function uses the 8051 "rotate left" instruction here, which *with these operand values* works as a left-shift because the high bit (bit 7) of these 8-bit values is zero. Bound-T/8051 originally modelled the result of "rotate left" as unknown because "rotate" is not *in general* a linear arithmetic operation. This made Bound-T/8051 fail to resolve the dynamic branch.
  - For the AVR version of `?C?CASE` in `tp_avr_7` we use the closest corresponding AVR instruction, `rol r1`, which rotates leftwards the 9-bit combination of the carry flag `C` and register `r1`, `C:r1`. It is easy to see that this instruction is equivalent to adding `r1` to itself with carry (`adc r1,r1`) and these instructions have the same encoding in AVR machine code. Bound-T uses its model for addition, which makes the new value of the carry flag unknown, because Bound-T does not in general model overflow. Bound-T's model is accurate for the first `rol r1` instruction at address C8: the preceding instruction `bclr C` zeroes the carry bit so Bound-T's model reduces to a doubling of the value of `r1`, which is equivalent (in the absence of overflow) to a left shift by one position. However, this doesn't work for the second `rol r1` instruction at address CA, because the input value of the carry flag for this instruction is the out-carry from the first `rol r1`, which Bound-T models as unknown, and so the value of `r1` after this second `rol r1` is also unknown to Bound-T, causing Bound-T to fail to resolve the dynamic jump.
  - In the variant test program `tp_avr_8` we changed the `rol r1` instructions into the AVR left-shift instructions `lsl r1`, which Bound-T models as a doubling of the value of `r1` with no input from the carry flag. In effect this assumes that the original value of `r1` is between 0 and 63. This assumption is unsafe in general but true for this particular case. Bound-T's arithmetic analysis is then accurate and resolves the dynamic branch precisely.
- The instruction at address CE (`add r0,r0`) takes the index into the jump table (in `r0`) and doubles it, because each entry in the jump table (which starts at address DC) consists of an `rjmp` instruction followed by a `nop` instruction and is thus two addressing units long. The offset, in addressing units, to the jump for index value *i* is thus 2*i*. The superfluous `nop` instructions are used in `tp_avr_7` to mimic the original jump table in the 8051 code, where the jump instructions themselves are two addressing units long. Because SWEET currently does not use the congruence domain, its model of `r0` after this doubling includes odd values, and this over-estimate prevents the resolution of the dynamic branch. In the variant test program `tp_avr_8` the `nop` instructions are removed, and then SWEET's analysis resolves the branch.

In summary, Bound-T fails on `tp_avr_7` because its model of possible overflow from an addition is too approximate (carry-out is unknown), while SWEET fails because it currently lacks congruence modelling.

For the variant test program `tp_avr_8`, Bound-T succeeds because its model of possible overflow in a left-shift operation is unsafely optimistic, while SWEET succeeds because the test program is adjusted to make congruence modelling unnecessary.

As an aside, this discussion shows that Bound-T's assumptions regarding overflow are not applied consistently when the model is translated into Presburger formulae for the arithmetic analysis. For example, Bound-T assumes that "left shift" does not overflow (not even into the sign bit), but assumes that addition can overflow into the carry bit. These conflicting assumptions were added to Bound-T over time, as *ad-ho*c patches to make this or that example program analysable. This situation is obviously undesirable.

As an example of this ad-hoc approach, the current version of Bound-T/8051 *is* able to resolve the dynamic branch in the original Intel-8051`?C?COPY` function, because the model of the 8051 "rotate left" instruction has been changed so that the output value is modelled by the conditional expression "if (input value) in 0 .. 64 then 2*(input value) else unknown". Such conditional expressions can be precise models of the instructions, and can be translated into disjunctive Presburger formulae for the arithmetic analysis. However, experiments show that if

this is done for all instructions then the Omega Calculator cannot solve the resulting Presburger formulae with reasonable resources, probably because they contain too many disjunctions. Moreover, for processors with wider words, for example 32-bit processors, the constants in the conditional expressions are large (on the order of two raised to the number of bits in a word), and there is reason to believe that the solution cost of Presburger formulae scales with the magnitude of the constants. Therefore, the general use of conditional expressions to model non-linear instructions is infeasible for Bound-T's current arithmetic analysis.

A possible approach may be to first make a value-analysis using abstract interpretation (not abstract execution) and the interval domain. In SWEET, the interval domain safely models overflows and wrap-arounds, and the abstract interpretation always terminates, through widening. The resulting interval bounds could perhaps then be used to show at which points in the program overflows or wrap-arounds might occur. With this information, it might be possible to make a safe and sound translation to Presburger formulae in which the expensive conditional expressions are used only where necessary, that is, only for the instructions where the interval-based analysis shows possible overflow. Unfortunately, when the program contains loops with non-obvious iteration bounds, ordinary widening tends to produce unbounded intervals, which means that overflows are not excluded. The widening used in the bounded polyhedral analysis [11] always generates bounded intervals, but these intervals can have large bounds (on the order of 2 raised to the number of bits in a word), which again bodes ill for the Presburger analysis.

5 Summary and Conclusions

5.1 Goals and Methods

To summarise this report, we have coupled two program-analysis tools, Bound-T and SWEET, so that Bound-T can use SWEET's analyses to supplement or replace Bound-T's own analyses. The coupling is based on a translation, implemented in Bound-T, of Bound-T's internal program model into the ALF language, which is the input language for SWEET. Our prototype extends the version of Bound-T which analyses machine-code programs for the Atmel AVR architecture. The prototype can also be seen as a front-end for SWEET through which SWEET can analyse AVR machine-code programs.

We hoped that the combined Bound-T and SWEET tool would better be able to analyse and resolve dynamic branches in machine code, because this is a critical problem that any practical analysis of machine-code programs must solve precisely and mostly automatically, in order to construct a control-flow graph of the program. All other analyses need the control-flow graph, which had better be valid and exact (not significantly over-estimated).

5.2 Overall Success Rates

We evaluated the Bound-T - SWEET combination on seven example programs or, rather, seven kinds of dynamic branches either extracted from various real programs or constructed to resemble dynamic branches in real programs. The examples concentrate on "switch-case" constructs because dynamic branches from switch-case constructs can be analysed locally. The other sources of dynamic branches — function pointers and virtual functions — were not considered because their resolution demands a global analysis which Bound-T does not provide at present. The example programs demonstrate several kinds of switch-case code constructs, with dense or sparse numbering, and implemented by several types of code, including direct indexing of a table of jumps, direct indexing of a table of target addresses, and various forms of tables encoding the case numbers and their code addresses.

Table 6 cross-tabulates the overall results of the evaluation by showing the number of example programs for which Bound-T alone, or Bound-T in combination with SWEET, failed or succeeded to resolve the dynamic branch. Note that we tested the combination with some or all of Bound-T's own analyses disabled, and this explains how the "combination" in some cases fails although Bound-T alone succeeds (with all its analyses enabled). All failures manifested as so loose bounds on the target address that Bound-T decided to consider the analysis failed, instead of adding thousands or tens of thousands of apparently possible but mostly false targets and edges to the control-flow graph.

Table 6: Results by Number of Failures and Successes for Each Tool Configuration

		Bound-T alone		
		Successful	***Failed***	***Total***
Bound-T combined with SWEET	***Successful***	3	1	4
	Failed	1	2	3
	Total	4	3	

Overall, both tool configurations (Bound-T alone, and Bound-T combined with SWEET) were equally successful, scoring four successful analyses and three failures. However, the total number of example programs is not very large, and some of these programs were deliberately constructed or tweaked to make one or the other tool configuration succeed or fail. Therefore, this success/failure proportion is not very predictive of practical performance. The apparent success rate (four out of seven) is really too small for a practical tool, at least for a commercial tool. However, one can hope that serious tool users would be able to control their coding style to ensure that a higher proportion of switch-case statements can be analysed.

Of more interest is understanding the reasons for the successes and failures. We discuss this below.

5.3 Why Some Analyses Failed

First, please note that when we talk of failures of SWEET in this section, we mean failures of the *combination* of Bound-T and SWEET. Some of the failures may be due to defects in Bound-T's internal program model or defects in the translation of this model to ALF for SWEET to analyse.

The main cause of failures for SWEET (that is, the Bound-T and SWEET combination) is the current lack of congruence analysis. Why is this so critical for these example programs? The entries in tables of jump addresses or jumps are often more than one addressing unit in length, which means that the code multiplies the table index by the entry length in order to compute the offset to the indexed entry. The lack of congruence analysis means that SWEET assumes that the offset can take any value in the interval covered by this product, a considerable over-estimate which leads to a horrible over-estimate in the values of the entries that may be read from the table, because the analysis includes "misaligned" reads. For the AVR, with octet addressing and 16-bit addresses, the problem is already quite bad, but it will be worse on the wider but still octet-addressed processors such as a 32-bit ARM where a typical table entry is four or more addressing units long.

The lack of congruence information is the direct reason for two of SWEET's failures and a contributing factor in the third failure. Thus, adding congruence to SWEET would bring its success rate up to 6 out of 7 programs.

The third SWEET failure, described in section 4.6.2, has a more fundamental cause: the non-relational nature of SWEET's abstract execution. This example program computes the final switch offset from the given index in three stages: the first stage computes an intermediate result from the index; the second stage compares the index to the index-bounds of the switch-case statement and branches to the default code if the index is out of bounds; the third stage, which is executed only if the index is within bounds, completes the case selection using both the index *and* the intermediate result computed in the first stage. When stage 1 is abstractly executed, there are no bounds on the index and therefore the intermediate result is also unbounded. When stage 2 is abstractly executed, it provides bounds on the index for stage 3, but these bounds are not applied "retro-actively" to stage 1. The intermediate result from stage 1 remains unbounded and, therefore, so is the result of stage 3.

In contrast, the Presburger-arithmetic analysis in Bound-T maintains the relationship between the value of the index and the value of the intermediate result, which is established in stage 1. When stage 2 provides bounds on the index, the analysis propagates these bounds through this relationship to produce bounds on the intermediate result. Therefore, stage 3 has bounded inputs and produces a bounded result.

The main cause for Bound-T's failures (from the viewpoint of Bound-T's analysis principles) is that it lacks special analyses or detectors for the specific forms of dynamic branches, switch tables, and switch-table handlers used in some of these example programs. For example, if Bound-T had a detector for the switch-table format used in `tp_avr_6`, its partial-evaluation analysis would succeed. If this form of switch-table were to be generated by a commonly used cross-compiler for the AVR, it would be worth-while to add such a detector to Bound-T, just as Tidorum has implemented detectors for the switch-tables and handlers used by the IAR compilers.

If such detectors were added to Bound-T/AVR, two failures would become successes, making Bound-T's success rate 6 out of 7 programs.

A secondary cause of Bound-T's failures, which also contributes to failures of the Bound-T and SWEET combination, is the unsystematic and unsafe modelling of signedness and overflow/wrap-around in Bound-T. For some example programs, Bound-T's unsafe assumptions are not strong enough; for example, the analysis of the program `tp_avr_21` in section 4.2 needs supporting assertions that constrain some variables to be non-negative (that is, unsigned). In other cases, such as in the programs `tp_avr_7` and `tp_avr_8` discussed in section 4.7, the assumptions are either too weak, or strong enough, depending on which instruction the program happens to use, from a set of functionally equivalent instructions.

With just a little bit of exaggeration we can say that SWEET fails because its general analysis is not general enough: it does not analyse congruences and is not relational. Conversely, Bound-T fails because its specific, specialized analyses were not specifically specialized for the kind of dynamic branches appearing in some of these example programs.

Clearly, there is more potential in further generalizing SWEET's analysis, but that may also be more difficult, at least if a relational analysis is desired. Extending Bound-T's set of specialized analyses is theoretically trivial but may be practically useful in the short term. A somewhat non-trivial direction for Bound-T would be to develop more general patterns of switch-case code, patterns which would be flexible enough to match the code-idioms generated by various cross-compilers for various target processors. This would be particularly useful for code that picks addresses or offsets from tables in memory. Such tables cannot be modeled directly in the Presburger analysis because that analysis cannot model memories or arrays.

5.4 Precision of Successful Analyses

For all analyses of these example programs, when the bounds on the dynamic branch were so tight that the analysis was considered successful, the bounds were in fact precise and the precisely correct set of target addresses was found.

When we compare the WCET bounds in the three cases where both analyses succeeded, we find that they compute the same WCET bound in two cases, but in the third case the WCET bound from the Bound-T and SWEET combination is over twice as large as the WCET bound from Bound-T alone. Table 7 shows the numbers; the WCET bounds are given in units of AVR processor cycles.

Table 7: Comparison of WCET Bounds

Program	*Section*	*Description*	*WCET Bound*	
			Bound-T alone	*Bound-T & SWEET*
`tp_avr_21`	4.2	A bottom-test loop which contains a switch-case statement implemented by an indexed dynamic jump into a table which contains static jumps to the cases.	129	129
`tp_c_2 / KuiSnd5Z`, IAR	4.5	A sparse C switch-case, compiled to use the real switch-table form and real switch handler from IAR Systems.	284	577
`tp_avr_8`	4.7	An indexed jump into a dense table of jumps, in which the 4-bit index is assembled from two 2-bit pieces using "shift" followed by "or". Jump-table entries are one addressing unit long.	40	40

This WCET over-estimation is explained by the different treatment of the loop in this test program. The loop is in the IAR switch-table handler. The loop traverses the switch-table from start to end, looking for the case that matches the given index number, and branching (dynamically) to that case statement if it finds a match. Bound-T uses its partial-evaluation analysis on the switch-handler subprogram, which in effect expands the loop into non-looping code which looks at each entry in the switch-table and conditionally branches to the corresponding case-statement code. Some of the case statements are "heavier" (take more time to execute) than others, which means that the worst-case path is not necessarily the one that picks the last entry in the switch-table. The partial evaluation produces a non-looping control-flow graph which shows exactly how many iterations of the loop (that

is, how many expanded repetitions of the loop body) lead to each case statement: one iteration leads to the first case, two iterations to the second case, and so on. Therefore, when Bound-T calculates the WCET bound from this control-flow graph, the heaviest case statement is combined with the true number of loop iterations for this case, not with the total number of iterations.

In contrast, although SWEET's abstract-execution analysis also "executes" each iteration of the loop in sequence (and, with the "no merge" option, without using abstracted value-approximations for switch-table entries), the output visible to Bound-T is only the list of targets (case-statement addresses) of the dynamic branch, which follows the exit from the loop. The loop remains in the control-flow graph and is not expanded into separate iterations. The relationship between the number of loop iterations actually executed, and the target address actually used in the dynamic branch, is lost. Therefore, when Bound-T calculates the WCET bound, it must combine the largest possible number of loop iterations with the heaviest of the case statements.

This kind of over-estimation, where the worst-case bounds of two parts of the program are added to make a worst-case bound for the whole program, although the worst cases for the parts cannot occur together in the same execution, is typical and common in WCET analysis. That Bound-T's partial-evaluation analysis avoids it is of course good, but it merely reflects the method's exhaustive path exploration and explicit representation of all (local) paths in the control-flow graph, which would be an infeasibly expensive method to use more globally. For other kinds of inter-related control-flow, Bound-T alone can produce WCET over-estimates comparable to the over-estimate from the Bound-T and SWEET combination for `tp_c_2`.

One way to remove this WCET over-estimation in the Bound-T and SWEET combination is to let SWEET, and not Bound-T, compute the WCET bound using the "execution time as a program variable" method [10]. This would require two SWEET abstract-execution analyses: once on the incomplete control-flow graph, to discover the possible targets of the dynamic branch, and a second time on the complete control-flow graph, to compute the WCET bound. But the analysis of the dynamic branches must usually be confirmed by an analysis of the completed control-flow graph anyway, to verify that no new targets appear, so this second, confirming analysis could be used to compute also the WCET bound.

5.5 Is the Combination More Powerful than its Component Tools?

In our evaluation, the combination of Bound-T and SWEET succeeded in one case where Bound-T alone failed, so the combination is more powerful than Bound-T alone.

The increase in power would have been much greater if the combination had been implemented before Bound-T was extended with the partial-evaluation method [6]. This method is much more similar to SWEET's abstract execution than are the other analyses in Bound-T and its analysis-abilities overlap with those of abstract execution. However, the partial-evaluation method requires significant target-specific and perhaps cross-compiler-specific implementation effort, while the abstract execution in SWEET is target-independent. On the other hand, implementing ALF export in Bound-T also requires some target-specific effort, because the modelling of registers and memories currently has several target-specific aspects.

Bound-T's partial-evaluation method works with a numerical domain that has just two levels: a value is either exactly known (a single concrete value), or completely unknown. In contrast, SWEET's abstract execution has the additional capability of working with abstracted sets of values (intervals) and of merging the abstracted values at control-flow joins. However, this brings with it the risk of over-estimating value-sets and target-address sets, because additional approximations (over-estimates) occur in merging (which we prevent with the "no merge" option) and also when some operations are applied to abstracted values (which we cannot prevent now, but see the suggested "atomizing procedure" in section 5.6).

Another difference between the methods is that the partial-evaluation method provides a residual flow-graph which shows the exact instruction sequence leading to each case. All loops and branches in the switch-table handler are executed, and the actual instruction sequence taken is recorded in the flow-graph, up to each resolution of the dynamic branch. This makes the calculation of the WCET bound more precise. SWEET's abstract execution, as we use it, provides only the target addresses, but not the execution paths nor the execution time to reach each target address. This can make the WCET bound less precise, as we saw in section 5.4. The abstract execution could certainly record more information in its states, such as the number of the iteration of the table-scanning loop on which each target address occurs. This could be seen as making the abstract execution more "relational" in style. Some such information could already be extracted from SWEET by activating the tracing output, but extending the basic abstract-execution method to a relational view seems a better approach.

5.6 Suggestions for Improvements and Future Work

We conclude this report by listing some possible improvements and areas for future work.

Our study in section 5.3 of the reasons for failed analyses shows that some relatively minor improvements and extensions to Bound-T and SWEET could immediately and significantly increase analysis power: SWEET should implement congruence analysis, and Bound-T should extend the modelling of operation chaining beyond the current 8-bit-to-16-bits level.

The over-approximation inherent in some of SWEET's abstract operations, when applied to abstracted values, could perhaps be reduced or eliminated by locally splitting abstracted operand values (intervals) into their atomic (singe value) components, applying the operation to each single concrete value, giving the exact single concrete result, and then again collecting all these results into an abstracted result value (an interval). Such an "atomizing" step could be inserted selectively for operations which might need it, depending also on the "size" of the abstracted operands (to avoid computing billions of separate cases). It seems fairly simple to implement atomizing in abstract execution: the single state represented by the abstracted operand values is divided into a set of single-value states, these are passed through the operation, and the set of single-valued results is merged into an abstracted result.

A similar atomizing method could make Bound-T's partial evaluation method more generally useful. For example, if a dynamic branch in a subprogram can be shown to depend only on an 8-bit parameter, Bound-T could partially evaluate the subprogram for each possible parameter value, for a total of 256 evaluations, and in this way collect all the possible target addresses, without detecting target-specific or compiler-specific idioms for switch-case code.

In the longer term, Bound-T should improve its modelling of signedness, overflows, and wrap-arounds. The present ad-hoc and unsafe models contaminate the exported ALF form of the program and weaken SWEET's part of the analysis.

Considering other directions of extension, we could use SWEET's abstract-execution method for other parts of WCET analysis, for example to find loop iteration bounds and other flow-facts. Conversely, loop-bounds found by Bound-T could be used to constrain SWEET's analysis (if SWEET would be extended to support the required annotations).

In the APARTS project, we originally planned to apply a new relational and bit-precise numerical domain to the problem of dynamic branches. The bit-precise nature of the analysis was expected to model signedness and wrap-around effects. The new domain developed in APARTS, bounded polyhedra [11] is relational and models wrap-around, but unfortunately it does not model congruences. As seen in our example programs, congruences are very important for precise modelling of dynamic branches when the branch code uses some form of array or table with elements that are larger than one addressing unit. It may be possible to combine the bounded-polyhedra domain with a congruence domain into a product domain. This is one possible subject for future work, to be undertaken when the implementation of bounded polyhedra in SWEET is robust and automatic enough to be practically useful.

Here is a more speculative idea. The SWEET group has been working on a novel form of WCET or control-flow analysis which uses a relational value-analysis to set bounds on the number of executions of each part of the control-flow graph [12, 13]. The idea is that if we assume or know that the program terminates, then no part of the control-flow graph can be executed more often than the total number of possible variable-value states at that point in the program. This number is called the *census* of the variable values. The census computation can be limited to the variables which influence the flow of control. While the census number can be used directly in a WCET computation, it could perhaps also be used to help an analysis tool decide if it is worth-while and practical to atomize the abstracted values at this point in the program, in order to improve the precision of the value-analysis. Bound-T could perhaps also use the census number to decide whether to apply its partial-evaluation method case by case, in an atomized manner, as suggested above.

To conclude, Tidorum judges that the combination of Bound-T with SWEET works well enough to become a standard feature of Bound-T, assuming that SWEET is extended to implement congruence analysis. Thus, future work will include moving the current prototype combination into the main-line of Bound-T development.

References

1 Tidorum Ltd., "Bound-T time and stack analyser". http://www.bound-t.com/.

2 SWEET (SWEdish Execution Time tool). http://www.mrtc.mdh.se/projects/wcet/sweet/index.html.

3 J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg and L. Källberg, "ALF – A Language for WCET Flow Analysis". In N. Holsti (ed.), *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis* (WCET09), June 2009.
http://www.es.mdh.se/publications/1420-ALF____A_Language_for_WCET_Flow_Analysis.

4 R. Wilhelm et al., "The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools". ACM Transactions on Embedded Computing Systems, Volume 7, Issue 3 (April 2008), pp.-36:1-36:53.

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

6 N. Holsti, "Analysing Switch-Case Tables by Partial Evaluation". 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007), Pisa, Italy, July 3, 2007.
http://www.bound-t.com/reports/wcet2007/abstract.html.

7 J. Gustafsson, A. Ermedahl, C. Sandberg and B. 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) (Dec. 2006).

8 J. Gustafsson, A. Ermedahl and B. Lisper, "Towards a Flow Analysis for Embedded System C Programs". Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005), Sedona, Arizona, USA, February 2 - 4, 2005.

9 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.

10 N. Holsti, "Computing Time as a Program Variable: A Way Around Infeasible Paths". 8th International Workshop on Worst-Case Execution Time Analysis (WCET'2008), Prague, Czech Republic, July 1, 2008.
http://www.bound-t.com/reports/wcet2008/abstract.html.

11 S. Bygde, B. Lisper, N. Holsti, "Fully Bounded Polyhedral Analysis of Integers with Wrapping". International Workshop on Numerical and Symbolic Abstract Domains (NSAD11), September 2011.

12 S. Bygde, A. Ermedahl and Björn Lisper, "An Efficient Algorithm for Parametric WCET Calculation". *Journal of Systems Architecture* vol. 57, pp. 614-624, May 2011.

13 S. Bygde: "Parametric WCET Analysis". Mälardalen University, School of Innovation, Design and Engineering, 2013. Mälardalen University Press Dissertations No. 138. ISBN 978-91-7485-109-0, ISSN 1651-4238. http://www.es.mdh.se/publications/3020-Parametric_WCET_Analysis.

Document Status and Change Log

Version	*Date*	*Changes/status*
Draft 1	2014-01-23	First draft, for review and discussion among authors.
Issue 1	2014-02-28	Incorporates results of review and discussion among authors.