Automatic Test Data Generation from Embedded C Code

Eileen Dillon and Christophe Meudec

Institute of Technology, Carlow
Computing and Networking Department,
Kilkenny Road, Carlow, Ireland
Tel : +353 (0)59 9176266
Fax : +353 (0)59 9170517
{dillone,meudecc}@itcarlow.ie

Abstract. A fundamental area of software engineering that remains a challenge for software developers is the delivery of software with the minimum of remaining defects. While progress is constantly being made in the provision of static analysis tools to partly address this problem, the complementary dynamic testing approach, which remains an essential technique in the software industry for the verification and validation of software, has received less attention. Within the software testing activity, the actual generation of test data for the purpose of automated software testing is still mainly a manual task. We present CSET (C Symbolic Execution Tool) which automatically generates test data from C source code to fulfil code coverage criteria. CSET implements the symbolic execution technique with an intermediate path traversal conditions checker and a test data generation facility. We examine how the traditional problems associated with the symbolic execution technique have been overcome using Logic Programming and Constraint Logic Programming (CLP). The approach used to handle pointer manipulations is detailed. Interprocedural results on previously published sample code and industrial embedded C code with pointers are presented.

1 Introduction

Microprocessor based embedded systems are omnipresent in our everyday environment, from the telecommunication to the automotive industries. The thorough verification and validation of the software part of those embedded systems is critical for the overall success of the product. An automatic technique that has great potential in this phase of software development is symbolic execution. By abstracting the code under analysis using symbolic values rather than actual values, symbolic execution can be adapted via automatic test data generation to automatically support many of the tasks involved in the dynamic verification and validation of software [4], including: coverage testing (tests fulfilling adequacy criteria), security analysis (tests highlighting buffer overflows), reliability (tests exposing run-time errors) and validation (tests falsifying assertions).

M. Heisel et al. (Eds.): SAFECOMP 2004, LNCS 3219, pp. 180–194, 2004.
© Springer-Verlag Berlin Heidelberg 2004

However, symbolic execution has not thus far fulfilled its full potential because of the many seemingly intractable difficulties that arise in its exploitation. While most of the difficulties encountered are common to the analysis of all application software, such as the automatic satisfiability checking of large and complex Path Traversal Conditions (PTCs) or the difficulty in implementing a test data generation step, the C programming language presents a unique challenge for the developers of tools based on symbolic execution that require a strong path feasibility checker: the pervasive use of pointers in C code. C is the dominant programming language for embedded systems [26]. This is largely due to its inherent flexibility, the extent of support, small code size, speed and the availability of compilers for a wide range of hardware.

MISRA C [24] is a well defined subset of C that forbids the use of C features that give concern in a safety-related context. For example, while pointers are allowed they can only reference statically declared data. MISRA C is used in the automotive industry for software up to, and including, SIL 3 [6]. Most safety-related automotive software are given a SIL of 1 or 2. As our work covers the MISRA C subset it has direct relevance for the safety aspects of today's automotive industry.

In this paper we present our automated tool CSET (C Symbolic Execution Tool), aimed at automatically generating test data to fulfil a code coverage criterion, that can be integrated in automatic verification and validation tools for embedded software written in C. This works builds on our previous results on SPARK-Ada [21,13] and Java Bytecode [9]. We extend our previous results [8] on C by tackling interprocedural test data generation, comparing our tool against a commercial equivalent [15] and also by presenting refined experimental results using our improved path feasibility checker. Our work is based on the use of the symbolic execution technique, Logic Programming and Constraint Logic Programming. For completeness we acknowledge that new test data generation techniques with as wide a range of applications as symbolic execution have been investigated [11,27,12,28]. Other techniques, with a smaller focus, have also been proposed [18,7].

The remainder of the paper is structured as follows. Section 2 introduces the symbolic execution technique. Section 3 presents our rationale for using Logic Programming and Constraint Logic Programming to overcome the problems inherent to symbolic execution in an automatic test data generation context. Section 4 introduces CSET, our tool for the symbolic execution of, and test data generation from, embedded C code. Its handling of C pointers is presented. In section 5 we discuss our experience so far of CSET on previously published sample code as well as on industrial code. The paper concludes with an overall assessment of our results.

2 Symbolic Execution

Symbolic execution [17] is primarily a static technique that follows the control structure of the code under analysis to generate symbolic information.

Table 1. PTC and symbolic values for path: `1, 2, 3, 4`.

Line	Code	Path Traversal Condition	den	a
1	int map (int den, int a) {	true	Den	A
2	den = den*(a+a);	true	Den*(A+A)	A
3	if (den == 90)	Den*(A+A) = 90	Den*(A+A)	A
4	return -1/(den-90);	Den*(A+A) = 90	Den*(A+A)	A
5	else return -2;	-	-	-
6	}	-	-	-

2.1 The Symbolic Execution Technique

Symbolic execution does not execute a program. The notion of execution implies the traversal of a path through the program using a set of data values to represent the input variables [5]. A program that is executed in this way will result in a set of output values. In symbolic execution, on the other hand, information is extracted from the source code of a program by representing inputs as symbolic values rather than using actual values. A set of symbolic expressions, one for each variable, is produced. Each of these symbolic expressions is made up in terms of input variables and constants. The presence of a conditional statement such as an `if ...else` splits the execution of the program into different paths. Symbolic execution records for each potential path, a Path Traversal Condition (PTC). This is the logical combination of the Boolean logic conditions that were encountered along that path [21].

For example, consider the artificial code in Table 1. There are 2 potential paths through this program only one of which is analysed here. The PTC must be satisfiable in order for the path to be feasible i.e. if a set of values for the variables in the PTC exists that satisfies it (within the type and range of values allowed), then that path is a feasible path through the program. Infeasible paths are common; no set of values for the variables in the PTC exists which satisfy that expression. For accurate program analysis infeasible paths must be detected. This involves checking the PTC for satisfiability. Further, for test data generation purposes a solution to the PTC of feasible paths must be generated. For the path illustrated, our CSET tool correctly identifies the PTC as satifiable and therefore generates a suitable test, e.g. `Den = -1, A = -45`, which when executed, using a third party tool, generates a division by 0 run-time error. On this example, CSET generates just 2 tests to achieve full path coverage. On the other hand, the dynamic analysis part of C++Test [15], a popular testing tool, generates 64 tests but fails to cover the path illustrated thus only achieving 50% path coverage overall. The static analysis part of C++Test also fails to flag the division by 0.

2.2 Traditional Problems of Symbolic Execution

Despite the high promises that the introduction of the symbolic execution technique engendered, it has not, to date, been used to its full potential in industry at

least for dynamic verification and validation purposes. This is due to a number of technical difficulties that have traditionally hampered its practical development.

Satisfiability Checking and Test Data Generation. The PTCs generated during symbolic execution are complex algebraic expressions and their satisfiability is in general undecidable [29]. Hence, automatic test data generators can never achieve completeness.

A PTC condition can contain many expressions involving integers, floating point numbers, pointer references and multi-dimensional input-dependent array references organised in arrays and structures combined by Boolean operators. Input dependent array references create ambiguities in PTCs.

In practice, determining the feasibility of such PTCs is very difficult. This implies that automatic test data generators based on symbolic execution are usually only applicable on restrictive subsets of programming languages, exhibit a high level of unsoundness and are not scalable.

Loops. Traditional symbolic execution cannot in general proceed beyond a loop unless the number of iterations is known. Difficulties arise when handling loops whose iterations are input-dependent. Analysing these accurately in the general case requires the use of recurrence relations [5].

Pointers. Pointers are problematic because all references to them are ambiguous until actual execution time (the aliasing problem). Further, pointer arithmetic, as allowed in C, is problematic since knowledge of the specific memory storage mechanism used is usually not included in symbolic execution tools.

Static analysis tools have efficiently tackled pointers on large industrial code. Lyle and Binkley [20] decompose C programs into program slices using a variation of symbolic execution to deal with pointer variables. Their approach is however not applicable to automatic test data generation. LCLint [10] is a static analysis tool used to detect errors in programs written in C. It requires the use of user annotations and the emphasis is on tractability rather than soundness (not all errors found are true errors—false positives) or completeness (all defects reported). PREfix [3] is a fully automated compile-time analyser, which detects errors, using symbolic execution, in large real-world examples in C and C++ code. It outputs the execution paths through the source code where these defects lie. It tries to avoid false positives but at the expense of completeness. The SLAM project [1] uses static analysis on C programs to determine whether they violate given usage rules. The programmer does not have to annotate the code and false error messages (noise) are kept to a minimum. It has been successfully applied to industrial code. The tool can analyse the feasibility of paths in the C program. SLAM is incomplete but sound within its context of application.

It thus emerges that static analysis tools have successfully handled pointers at the expense of soundness and completeness. They have nevertheless successfully demonstrated their usefulness on large industrial code. However, the approaches

used, especially for pointer handling, cannot be used in the area of automated test data generation which requires stronger path satisfiability checking capabilities and an actual test data generation step.

Function calls. Interprocedural symbolic execution is problematic because of the complexity it engenders. In particular, the complex identifier scoping rules of high level programming languages are difficult to respect (including the pass-by-reference and pass-by-value mechanisms), the semantics of functions with side effects is intricate and finally, function calls substantially increase the complexity of the underlying control flow graph of the program under test.

2.3 Conclusion

Past symbolic executors that provide a test data generation facility only deal with subsets of programming languages that excludes pointers, do not incorporate a powerful PTC satisfiability checker, and do not integrate a subpath selection strategy. Further, the various techniques that have successfully been used in static analysis tools do not seem transferrable to dynamic tools.

Before detailing in section 4 how we have successfully tackled these issues, over a number of projects [21,13,9,8], we present the programming paradigm on which our work is based.

3 Logic Programming and Constraint Logic Programming

As seen, a symbolic executor needs to be implemented that is guided in its search by the feasibility of potentially large algebraic expressions. In particular, given an algebraic expression, along with the variables involved and their respective domains, it must be shown that there exists an instantiation of the variables which reduces the expression to true. In effect, an algebraic expression constrains its variables to a particular set of values from their respective domains. If any of the sets are empty, the PTC is unsatisfiable.

To implement the kind of solver required here, e.g. able to work with non-linear constraints over floating point numbers and integers, it is possible to implement heuristics by writing a specialized program in a procedural language (such as C, or using an existing solving routines library). Although the heuristics are readily available, this approach requires a substantial amount of effort and the resulting solver is likely to be hard to maintain, modify and extend. Further, because of the heterogeneity of the programming constructs that appear in PTCs, unsound simplifications need to made to make this approach tractable. We believe that this is the approach currently used in most static and dynamic state-of-the-art commercial tools that use the symbolic execution technique (although we have no means of verifying this) and that this is the source of their weak PTCs checking capabilities.

The advantages of using Logic Programming over procedural programming have long been recognized for testing tools [14] and in commercial static analysers (e.g. the SPARK Examiner [25]).

Prolog's in-built depth-first search procedure and its backtracking facilities make Prolog a strong candidate for implementing a symbolic executor that follows the control flow graph of the program under consideration according to a given testing criterion and backtracks whenever unsatisfiability of the current PTC is detected by a purpose built constraints solver.

Constraint Logic Programming (CLP) [16] improves the modelling capabilities of mathematical relationships between objects of Prolog by providing richer data structures on which constraints can be expressed and by using constraint resolution mechanisms (also known as decision procedures) to reduce the search space under consideration. When the decision procedure is incomplete—e.g. for non-linear arithmetic constraints—the problematic constraints are suspended, it is also said delayed, until they become linear. Non-linear arithmetic constraints can become linear whenever a variable becomes instantiated. This can happen when other constraints are added to the system of constraints already considered or during labelling.

The labelling mechanism further constrains the system of constraints according to some value choosing strategy. It can be viewed as a process to make assumptions about the system of constraints under consideration. This mechanism is used to awaken delayed constraints or generate a solution to an already known satisfiable system of constraints (as required for test data generation).

4 CSET

4.1 Overview

CSET (C Symbolic Execution Tool) is a symbolic execution tool for embedded C code which incorporates intermediate PTC checking and a test data generation stage. The main output from the tool are test data that can exercise the paths found feasible through a C source code. As schematised in Fig. 1, CSET is composed of the following:

Preprocessor. gcc is used to generate a C program free from macros.

Parser. The parser converts the preprocessed C code to a Prolog readable format for input into the symbolic executor adding scoping information for all variables. For example the Prolog terms obtained for the function given in Table 1 is represented in Fig. 2.

Symbolic Executor. The symbolic executor is implemented in ECLiPSe [19]. It takes as input the Prolog readable format of the C code under analysis and interacts with the solver to return test data.

Solver. The solver used is the PTC Solver [22] as introduced in Section 4.3.

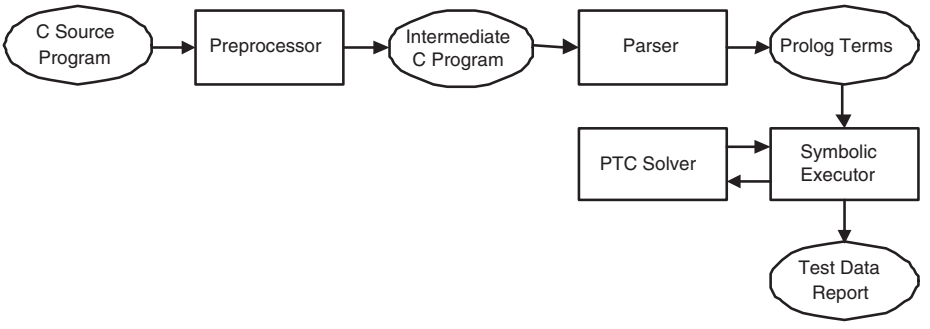

Fig. 1. CSET Architecture.

```
function_definition(map,[declaration(int,[Den]),declaration(int,[A])],
 [assignment(Den,multiply(Den,(A+A))),
  if_statement(8,expression(Den==90),[return(-1/(Den-90))],
                                     [return(-2)])
 ],int).
```

Fig. 2. Prolog Terms.

4.2 Algorithm Sketch

The main algorithm follows the design implemented in [21,9,13]. Details of the variables (such as their type, initial value and output value) in the Prolog readable format are added in a data structure that records changes to the variables during symbolic execution. The statements in the input file are processed sequentially. During an assignment the symbolic value of the assigned variable is updated to the assigned expression expressed in terms of input variables and constants only. Basic block statements proceed without creating choice points in the Prolog execution. Whenever a decision is encountered, it is symbolically executed i.e. expressed in terms of input variables and constants only, and a Prolog choice point is generated. The decision (or its negation) is then added to the current constraint store through the solver. If the solver fails to add a constraint then the PTC is unsatisfiable in which case, the symbolic executor will backtrack, undoing all actions up to the last choice point, and then proceeds forward again. This is achieved using the intrinsic backtracking facility of Prolog, which simplifies greatly the design of the symbolic executor and allows the intermediate checking of the PTCs. If successful, symbolic execution continues forward. On reaching a return statement all input variables involved in the current PTC are labelled by a call to the solver. This involves generating values for each input variables with respect to the PTC. If all the found feasible paths have been examined the symbolic executor terminates, otherwise backtracking occurs.

4.3 Addressing the Traditional Problems of Symbolic Execution

Here we detail how we have addressed the traditional problems of symbolic execution in CSET .

Satisfiability Checking and Test Data Generation. For the symbolic execution to be amenable, intermediate feasibility path checking is essential as otherwise many paths will be generated that are only discovered as infeasible at the end of the symbolic execution. In CSET, every time a decision in the code is encountered a choice point in the Prolog symbolic executor is created and the symbolically executed decision (or its negation) is immediately submitted to the PTC Solver. This allows many infeasible subpaths to be eliminated from the search space. This technique improves the scalability of CSET.

The PTC Solver [22], implemented in ECLiPSe [19], has been developed to check the satisfiability of PTCs as generated by a symbolic executor, and includes a labelling strategy. It has been used on a number of projects for a variety of target programming languages: Ada [21], Java Bytecode [9] and now C. It has a well defined Prolog interface and can be integrated in a Prolog or C++ [13] tool.

The solver is composed of the following components:

- fd, a constraint solver over integers (using domain propagation techniques) provided as a library by ECLiPSe;
- clpq, a constraint solver over infinite precision rational numbers (used to incorrectly model floating point variables) provided as a library by ECLiPSe;
- a bespoke bridge between fd and clpq to handle mixed constraints;
- custom extensions to handle C bitwise operators, constraints over arrays, records and enumeration literals. These make extensive use of the ECLiPSe delaying mechanism to handle constraints which cannot be resolved immediately (e.g. array access with unknown index).

Whenever a constraint is submitted to the solver it is added to its existing store of constraints and the solver can then:

- fail: the system of constraints was unsatisfiable, the system of constraint remains as it was before the addition of the latest constraint (i.e. the solver backtracks automatically);
- succeeds: the system of constraints may be satisfiable (e.g. if non-linear constraints are present the solver may fail to detect their unsatisfiability at this stage).

When the PTC is used to derive test cases the ambiguity can persist during symbolic execution and be resolved at the actual creation step of the test cases during labelling. This sampling is exhaustively attempted for variables over integers but of course only partially performed for the infinite precision rational numbers.

The PTC Solver can handle complex algebraic expression over integers, infinite precision rational numbers, enumeration literals, that are organised in arrays

(including multi-dimensional arrays) and structures [21,9]. As the PTC solver properly models arrays and structures, these types can contain elements of any type including, of course, arrays and structures ...

To understand how ECLiPSe's delaying mechanism can be used to handle unknown index reference, consider the following example:

```
1:      x = a[j];
2:      a[j-1] = x-1;
```

On line 1, the symbolic executor submits a constraint to the solver of the form $eq\_cast(X_1, element(A, [J]))$ where X_1 represents the new value of `x` in the symbolic executor. The solver delays the following constraint on the variable J: $element(A, [J], R_1)$ where R_1 will be the result of the element at position J in A whenever J becomes known. $eq\_cast(X_1, R_1)$, used to perform implicit casting, delays if necessary.

On line 2, the symbolic executor submits $A_1 = up\_arr(A, J-1, X_1-1)$ where A_1 represents the new value of `a` in the symbolic executor. The solver delays $up\_arr(A, J-1, eq\_cast(X_1-1), A_1)$ on J.

During labelling, as J will be instantiated, the delayed constraints are automatically resumed (in any order) and are eliminated. This implies that the symbolic values of `a` and `x` are now free from unknown index references.

While this is only an overview of how ambiguous array references are handled in the PTC solver (we have omitted the extra constraints placed by the PTC solver on the indices for example) it illustrates the general principles by which CSET can support all C array constructs.

Loops. As mentioned, traditional symbolic execution cannot proceed as usual in the presence of input-dependent iterative constructs. However, in the restricted case of trying to fulfil a given test coverage criterion, loops can accurately be dealt with according to the feasibility (or not) of the intermediate path followed in conjunction with a heuristics-based intermediate path selection strategy targeted at the chosen test data adequacy criterion. Integrating this approach in a symbolic executor that includes an intermediate PTCs checker, and for branch coverage at least, all loop constructs can be handled adequately as demonstrated in [21,9,13].

As currently the aim of CSET is to fulfil the path coverage criterion (because our ultimate aim is automatic Worst Case Execution Time estimation), which is harder to achieve than branch coverage for example, it does not include a path selection strategy. Thus CSET, will attempt to cover all the combinations of the Prolog choice points, as introduced by loops in the code, during the symbolic execution. Only the size of the definition domain of variables and the actual decisions in the iterative constructs limits CSET in its search.

Pointers. Previous symbolic executors, that include a test data generation phase, have avoided implementing pointers. Multi-level pointers indirection and

static pointer arithmetic have been successfully incorporated into CSET. Similarly to [2] (which is not a dynamic analysis tool), CSET stores information about every variable in the source program in a memory map structure [8]. This structure is used to replicate the way that C pointers behave during execution.

& Operator. When a variable is declared all its details are stored into the memory map, this includes its name, type, initial value (if any) and an arbitrary memory address. When a pointer is assigned the address of a variable, that variable is located in the memory map and its memory address is retrieved. This is assigned to the output field of the pointer in the memory map. An array is stored as one variable in the memory map since arrays are stored in contiguous memory in C. Therefore assigning the address of an individual element to a pointer variable is more complex than just described. The name of the array has to be extracted and its position in the memory map is found. Next the memory address of the individual element is calculated. For example, if the array starts at memory address 1000 and is of type int, the third element is located at memory address 1008 since the type int is represented by 4 bytes in CSET. In a similar way the address of objects such as structures are handled.

*** Operator.** When a pointer is de-referenced, it is symbolically executed to see what memory address it holds. The memory map is searched to find the details of the variable stored at that address and if found the corresponding output value is returned. There are a number of reasons why the memory address may not be found in the memory map. If the pointer has not yet been assigned the address of any object the pointer is assigned a random value. If its value is not within the memory range allocated to the program the symbolic execution will halt. The pointer may refer to the memory address of an array element. As stated, arrays are stored as one variable in the memory map. In this instance, the memory address of the nearest variable is found and this variable is checked to see if it is an array. If so, the memory address is checked to see if it legitimately refers to an element of that array, and if so the value that that element contains is returned. Pointers to structures are handled in a comparable way.

Pointer Arithmetic. CSET only supports static pointer arithmetic within single array and structure objects. In other words, the symbolic executor must be able to identify the resulting element pointed to by a pointer arithmetic expression and it must be an array element or structure field value. For example, when a pointer to an array of integers is incremented, the type that the pointer is currently pointing to is first found and as the type int is represented as 4 bytes in CSET, the pointer now holds the memory address 4 bytes on from the one it initially held.

Dynamic Memory Allocation. The standard library functions `malloc` and `free` have been simulated in CSET. The allocation of memory is achieved by adding a variable of the type being allocated to the memory map. The pointer that points to this location is updated with the appropriate memory address. When memory is freed, the corresponding pointer and variable are

removed from the memory map. This memory is then available for use by the program whenever other variables need to be allocated.

Function Calls. CSET handles interprocedural test data generation using an approach we previously described for SPARK-Ada code [21]. On encountering a function call, the symbolic executor matches the parameters with the arguments passed and processing proceeds as during actual execution. In addition, our memory map structure allows us to deal naturally with the pass-by-reference mechanism.

Furthermore, as during parsing of the source code identifiers are uniquely identified according to their scope our Prolog readable source code has a flat scoping level structure. Thus during symbolic execution scoping does not have to be taken into account.

4.4 Current Limitations

CSET does not check whether the input C code is beyond its limitations.

C Subset. While, the subset of C handled by CSET subsumes the MISRA subset [24], CSET cannot handle full ANSI C. The incompleteness of CSET is due to the following restrictions: only static pointer arithmetic within single array and structure objects is supported; and the following features are not handled: assignments within || and && expressions, pointers to functions, implicit modulo assignments, variables qualifiers such as extern, static and volatile. Finally only the `malloc` and `free` standard library functions are supported.

Large Integers. The PTC Solver can only handle 16 bit-encoded integers. This limitation can cause unsoundness in CSET.

Numerical Precision. Floating-point numbers in the code are incorrectly represented using infinite precision rational numbers within linear constraints but reverts to double precision floating-point numbers whenever non-linear constraints are posted to the solver. Again this can lead to unsoundness. Constraints solving over floating-point numbers is an area of on-going research [23].

PTCs Size. Symbolic execution can generate thousands of constraints involving thousands of variable occurrences. This is of course particularly true when analyzing code that contains loops that iterate a large number of times. Analyzing such code can therefore take several hours and in practice prevents the analysis of code containing large iterations.

Non-Linear Constraints. The analysis of PTCs that contain non-linear constraints can be hampered because the PTC Solver cannot always detect on submission their unsatisfiability. If such a path reaches the labelling stage its complete analysis is intractable in general using our approach.

5 Experience

CSET has been applied to a selection of functions from [28] and industrial C code made available to us by Pi Technology (Cambridge UK). All timings have been generated on a 2.5Ghz processor. Our results are compared against C++Test [15] which was also used to generate the path coverage measures quoted. For test objects containing loops (indicated by `*`) the path coverage quoted is an approximation.

5.1 Previously Published Samples

Table 2 illustrates the results obtained for test objects from [28] in where several thousands tests on each test objects are generated to achieve a high level of branch coverage. The number of iterations of both loops for `ComplexBranch` was reduced from 100 to 5 to make the code amenable to CSET. `Netflow` has proved too complex for CSET to handle.

Table 2. Published Samples Results.

Test Object			CSET Results			C++Test Results		
Name	Cyclomatic Complexity	Max. Nesting Level	No. Tests Generated	%Path Covered	Execution Time	No. Tests Generated	%Path Covered	Execution Time
Atof*	16	2	262	13	122secs	3	0	3secs
ClassifyFloat	14	2	26	25	94secs	1000	6	28secs
ClassifyInt	14	2	32	27	3.8hrs	1000	13	32secs
ComplexBranch*	13	2	304	51	24hrs	1000	29	26secs
IsElem	2	1	2	100	0.2sec	1000	100	26secs
LineCover*	8	4	12	16	9.2hrs	1000	16	25secs
Netflow*	14	2	0	0	failed	1000	0	58secs

For objectiveness we need to mention that C++Test is not explicitly targeted at path coverage and that it is a general purpose commercial tool with many functionalities that our tool does not provide. On the other hand, CSET is able to flag the subpaths that have been detected as infeasible.

The disappointingly long CSET running time of the some of these examples can be explained by the presence of non-linear constraints in infeasible paths in the code under analysis. To increase the soundness of CSET our timeout labelling strategy has been set to a generous value. Hence a high proportion of the total running time can be fruitlessly spent trying to generate tests for these paths. For `LineCover`, for example, this proportion amounts to 95%. We have no satisfactory solution to this problem.

While these results illustrate the limitations of our approach for handling complex algorithms, we are encouraged by the higher level of path coverage obtained by CSET, using a minimal number of tests, when compared to a popular dynamic analysis tool as it suggests a higher level of completeness for our path feasibility checking facility.

5.2 Industrial Code

Table 3 illustrates the complexity measures and the results obtained by CSET for a selection of C functions from industrial code. The selection was chosen for its wide variety of C features including arrays (single and multi-dimensional), enumerations, loops, casting, function calls, pointers and pointer arithmetic. Interprocedural test data generation was performed on the original code.

Table 3. Industrial C Code Samples Results.

Test Object			CSET Results			C++Test Results		
Name	Cyclomatic Complexity	Max. Nesting Level	No. Tests Generated	%Path Covered	Execution Time	No. Tests Generated	%Path Covered	Execution Time
Aip_med_filter	5	1	6	37	15secs	50	37	2secs
Byc_reset_boost*	3	2	1	16	51secs	50	16	3secs
Iti_engine_sync	6	2	10	55	18secs	1000	11	5secs
Oop_add_to_list*	7	3	123	59	59secs	50	7	18secs
Std_check-lrc*	2	1	1	33	4secs	50	33	2secs

Results reported in Table 3 are very promising as the running time is low and the percentage of paths covered by our tests is always higher or equal to what can be achieved using a common dynamic analysis tool. No limitations were applied to the range of data inputs or to the number of iterations of any loops encountered. It is worth noting that the industrial C code examples have a lower cyclomatic complexity and nesting level than the test objects from [28] that we examined.

6 Conclusions

CSET is the first automatic test data generator based on symbolic execution able to handle pointers as found in embedded C code.

Thus, we believe that the many applications of the symbolic execution technique can finally be successfully implemented for the most popular programming language in embedded systems. For example, although CSET is aimed at fufiling the path coverage criterion, we have shown [21,9] how for our approach, the easier, branch coverage criterion can be targeted. Further, CSET is able to handle a larger subset of C than the MISRA [24] subset that has specifically been developed for the automotive industry.

Our use of Logic Programming and Constraint Logic Programming, to address the traditional problems associated with implementing a useful automatic test data generator based on the symbolic execution technique, has been vindicated in this work by the results obtained on industrial code.

Whilst CSET is not without its limitations, and much work remains to be done to be able to tackle efficiently general C code, it seems sufficiently powerful already to deal with industrial C code, the target language of this work.

Acknowledgements. Thanks to Mike Ellims (Pi Technology, Cambridge UK) for giving us the C source from an Engine Control Unit and to Joachim Wegener (Daimler-Chrysler, Berlin Germany) for making available his examples from [28].

References

1. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. *ACM SIGPLAN Notices*, 37(1):1–3, January 2002. URL http://research.microsoft.com/slam/.
2. J. Blieberger, B. Burgstaller, and B. Scholz. Interprocedural symbolic evaluation of Ada programs with aliases. In *In Ada-Europe'99 International Conference on Reliable Software Technologies*, pages 136–145, Stantander, Spain, June 1999.
3. W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, 2000.
4. L.A. Clarke and D.J. Richardson. Application of symbolic evaluation. *Journal of Systems Software*, 5:15–35, January 1985.
5. P.D. Coward. Symbolic execution systems—a review. *Software Engineering Journal*, 3(6):229–239, November 1988.
6. B.J. Czerny, J.G. D'Ambrosio, P.O. Jacob, and B.T. Murray. Identifying and understanding relevant system safety standards for use in the automotive industry. In *Proceedings of the Society of Automotive Engineers World Congress*, Michigan, USA, March 2003.
7. R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
8. E. Dillon and C. Meudec. CSET: Symbolic execution and automatic test data generation of embedded C code. In *Proceedings 16th IFIP International Conference on Testing of Communicating Systems*, Oxford, UK, March 2004. Position Paper.
9. J. Doyle and C. Meudec. Automatic structural coverage testing of Java bytecode. In *Proceedings of the Third Workshop on Automated Verification of Critical Systems*, April 2003.
10. D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation*, 1996.
11. M.J. Gallagher and V.L. Narasimhan. ADTEST: A test data generation suite for Ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.
12. A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings ISSTA'98*, pages 53–62, 1998.
13. M. Grogan. Visual symbolic execution. Master's thesis, Institute of Technology, Carlow, Ireland, 2002.
14. D. Hamlet. Implementing prototype testing tools. *Software-Practice and Experience*, 25(4):347–371, April 1995.
15. Parasoft Inc. *C++Test – version 2.2*, 2004. http://www.parasoft.com/.
16. J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, January 1987.
17. J.C. King. A new approach to program testing. In *Proceedings International Conference on reliable software*, pages 228–233, April 1975.

18. B. Korel. Automated test data generation for programs with procedures. In *Proceedings ISSTA'96*, pages 209–215, 1996.
19. Parc Technologies Ltd. *ECLiPSe Release 5.6*, 2003. http://www.icparc.ic.ac.uk/eclipse/.
20. J.R. Lyle and D.W. Binkley. Program slicing in the presence of pointers. In *Proceedings of the Foundations of Software Engineering*, pages 255–260, Orlando, FL, USA, November 1993.
21. C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Journal of Software Testing, Verification and Reliability*, 11(2):81–96, 2001.
22. C. Meudec. The PTC solver user manual – version 1.5.1. Technical report, Institute of Technology, Carlow, Ireland, May 2004.
23. C. Michel, R. Rueher, and Y. Lebbah. Constraints solving over floating-point numbers. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 524–538, Paphos, Cyprus, November 2001.
24. MISRA. Guidelines for the use of the C language in vehicle based software. Technical report, Motor Industry Software Reliability Association, 1998.
25. Praxis Critical Systems Ltd, UK. *The Spark Examiner*, 2004. http://www.sparkada.com/.
26. V. Seppanen, A-M. Kahkonen, M. Oivo, H. Perunka, P. Isomursu, and P. Pulli. Strategic needs and future trends of embedded software. Technical Report 48/96, TEKES Development Center, Finland, October 1996.
27. N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proceedings ISSTA'98*, pages 73–81, 1998.
28. J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43:841–854, 2001.
29. E.J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal of Computers*, 8(4):587–589, 1979.