Binary-Level Testing of Embedded Programs

Sébastien Bardin*, Philippe Baufreton†, Nicolas Cornuet‡,
Philippe Herrmann* and Sébastien Labbé‡

* CEA, LIST, Saclay, France
Email: first.name@cea.fr
† Sagem - SAFRAN Electronics, Massy, France
Email: philippe.baufreton@sagem.com
‡ EDF Research & Development, Chatou, France
Email: first.name@edf.fr

***Abstract*—Dynamic Symbolic Execution (DSE) is a powerful approach to automatic test data generation. It has been heavily used in recent years for finding bugs in desktop programs. In this article, we discuss the use of binary-level DSE for testing safety-critical embedded systems. More especially, we present several innovative features implemented in our DSE tool OSMOSE, and we show through four case-studies how these features can be used in practical situations.**

Keywords*—*Automatic testing, symbolic execution, binary-level analysis

I. INTRODUCTION

Dynamic Symbolic Execution (DSE) is a powerful approach to automatic test data generation [3], [13], [14], [16], [24], [26]. It has been heavily used in recent years for automatically finding bugs in desktop programs [13], [14], [17]. While many DSE tools work on the source code of the program under test, a few other ones work on a binary-level description [3], [17], [18] (i.e. the executable file). This approach shows a number of advantages, making it possible for example to analyse mobile codes or programs incorporating commercial off-the-shelf components [3], [9], [18].

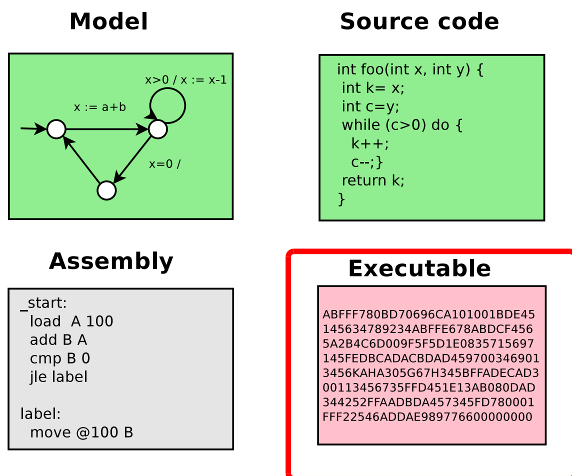


Fig. 1. Binary-level program analysis

In this article, we discuss the use of binary-level DSE for testing safety-critical embedded systems. Testing safety-critical embedded systems is significantly different from testing desktop applications. For example, while embedded programs are simpler in many ways (smaller size, no dynamic memory allocation), the testing process aims at full coverage and validation experts can pass a long time trying to achieve it or justifying for every uncovered branch. There are several ways in which companies working in safety-critical domains could benefit from automated binary-level testing.

- Even in the industry of safety-critical systems a company may not have access to the program source code of a piece of software it has acquired, typically when the company is not a major customer for the vendor. Then these executables have to be certified without any programming language description.
- A similar problem is the one of legacy code. Refactoring a certified code involves going through the certification process again. Yet, in some safety critical industries, product life-cycles span over several decades: the source code may no longer be available, or the programming language may not be supported anymore. In both cases, binary-level analysis is the only option left.
- In aeronautics, the DO-178B standard [22] imposes that verification must be performed on the binary level as soon as the conformity between the high level code and the machine code cannot be ensured. Since manual binary-level analysis is very expensive, constructors prefer to avoid any technology which would blur the conformity, including optimising compilers which would increase performances and lower costs.

In the past years, we developed and used our own binary-level DSE tool OSMOSE [3], [5] in a number of case-studies from aeronautics (in collaboration with SAGEM, French aeronautic equipment manufacturer) and energy (in collaboration with EdF, French energy supplier). In this article, we describe several features that we found of practical interest for the success of these case-studies. Our contribution include:

- the description of original and practically-relevant features for DSE tools, including a generic search API, search directives for reducing the exploration space and test suite replay & completion;
- an experience report on four real-life case-studies, describing the use of OSMOSE in different situations:

Work partially funded by ANR project BINCOA, grant ANR-08-SEGI-006.

unit-level testing of a medium-size program, system-level testing of a small but complex program, understanding and testing a third-party program and finally a comparison of binary-level testing strategies with several source-level testing strategies.

Outline. The rest of the paper is organised as follows. First, we describe DSE (Section II), we give a succinct characterisation of the class of programs we target (Section III) and we present the OSMOSE tool (Section IV). Then we describe the new features that we propose for DSE (Section V) and the case-studies we have performed (Section VI). Finally we discuss related work (Section VII) and conclude (Section VIII).

II. BACKGROUND

A. Notation

Given a program P over a vector of input variables V taking values in some domain D, a test data t for P is any valuation of V, i.e. $t \in D$. The execution of P over t, denoted $P(t)$, is formalised as a path (or run) $\sigma \triangleq (loc_1, S_1) \ldots (loc_n, S_n)$, where the loc_i denote control-locations (or control-points, or simply locations) of P and the S_i denote the successive internal states of P ($\approx$ valuation of all global and local variables as well as memory-allocated structures) before the execution of each loc_i. A test data t reaches a specific location loc with internal state S, denoted $t \rightsquigarrow_P (loc, S)$, if $P(t)$ is of the form $\sigma_1 \cdot (loc, S) \cdot \sigma_2$. We also write $t \rightsquigarrow_P \sigma$ to denote that test data t covers (or follows) the path σ. A test suite TS is a finite set of test data.

B. DSE in brief

We remind here a few basic facts about Symbolic Execution (SE) [19] and Dynamic Symbolic Execution (DSE) [16], [24], [26]. Let us consider a program under test P with a vector of input variables V over domain D and a path σ of P. The key insight of SE is that it is possible in many cases to compute a *path predicate* ϕ_σ for σ such that for any input valuation $t \in D$, we have: t satisfies ϕ_σ iff $P(t)$ covers σ. In practice, path predicates are often under-approximated and only the left-to-right implication holds, which is already fine for test data generation: SE outputs a set of pairs (t_i, σ_i) such that each t_i is ensured to cover the corresponding σ_i. Therefore, SE is *sound* from a testing point of view.

A simplified view of SE is depicted in Algorithm 1. While high level, it is sufficient to understand the rest of the paper. We assume that the set of paths of P, denoted $Paths(P)$, is finite. In practice, this assumption is enforced through a bound on path lengths. SE relies on the availability of both a procedure for path predicate computation (with predicates in some theory T) and a solver taking a formula $\phi \in T$ and returning either *sat* with a solution t or *unsat*. The algorithm builds iteratively a test suite TS by exploring all paths from $Paths(P)$.

The major complexity issue in Algorithm 1 is that SE must in some ways explore all $Paths(P)$. Therefore, the size of $Paths(P)$ is one of the two major bottlenecks of SE, the other one being the average cost of solving path predicates.

Algorithm 1: Symbolic Execution algorithm

Input: a program P with finite set of paths $Paths(P)$
Output: TS, a set of pairs (t, σ) such that $P(t) \rightsquigarrow_P \sigma$

```
1  TS := ∅;
2  S_paths := Paths(P);
3  while S_paths ≠ ∅ do
4  |  choose σ ∈ S_paths; S_paths := S_paths \ {σ} ;
5  |  compute path predicate φ_σ for σ ;
6  |  switch solve(φ_σ) do
7  |  |  case sat(t):  TS := TS ∪ {(t, σ)}
8  |  |  case unsat:  skip
9  |  end
10 end
11 return TS;
```

DSE [16], [24], [26] enhances SE by interleaving concrete and symbolic executions. The dynamically collected information can help the symbolic step, for example by suggesting relevant approximations. State-of-the-art SE/DSE tools lazily explore the set of paths, maintaining a set of explored path prefixes and discovering new path prefixes through flipping a single branching instruction along an already explored path. Such a detailed view of SE can be found in Section V-A. Another standard improvement over Algorithm 1 is to add a coverage-based stop criterion (instructions or branches). The algorithm maintains a set `G` of uncovered branches (resp. instructions), it stops when `G` is empty (modify `line 3`), and `G` is updated when a new test data is generated (modify `line 7`).

III. A FEW WORDS ABOUT SAFETY-CRITICAL PROGRAMS

We are interested in safety-critical control-command programs typically found in aeronautics and energy. We recall here briefly a few characteristics commonly found in these systems.

From a high-level point of view, these programs are reactive, i.e. they are composed of a main (non-terminating) loop performing data acquisition, internal computation and actuator activation. They often begin with a long (and mostly sequential) initialisation phase. They also often contain self-tests, i.e. software-level mechanisms for hardware-fault detection. These idioms are of the form A := 0, assert(A==0): an `assert` violation here will typically denote a malfunction in memory. Self-tests add to the programs many artificial and (normally) infeasible paths. Finally, these programs are self-contained: there are no dynamic library and the (simple) operating system is included in the code.

From a low-level point of view, there is no dynamic memory allocation. Memory is allocated statically during initialisation. Programs can contain strings, but they do not perform any advance string manipulation. Floating-point numbers (floats) and floating-point arithmetic are commonly found. Finally, interrupts, multithreading and time-based synchronisations can be found as well (but no dynamic thread creation). We do not claim supporting such features automatically. The OSMOSE tool is designed to help the certification expert. It eases analysis and testing on sequential portions of the code, but non-sequential aspects of the program must still be dealt with

by experts. This view is in adequacy with the methodologies observed at EdF and SAGEM.

IV. The OSMOSE tool

A. Overview

OSMOSE is a binary-level DSE tool [3], [5]. It takes as input an executable file, an entry address, a description of the initial memory state and a coverage criterion (plus optional DSE parameters). The main output of the tool is a set of test data with their expected trace executions. Other output include a coverage measure (if relevant) and a description of the program under test (control-flow graph, call graph, statistics, etc.). Tests are correct by construction (they should follow their expected paths), while coverage measure and program information may be incomplete (the analysis may miss part of the program because of dynamic jumps [5], see Section V-F). Currently supported testing criteria include path, instruction and branch coverage. Moreover, for the last two criteria, we distinguish between unit-level coverage and system-level coverage. A user view of OSMOSE is provided in Figure 2.

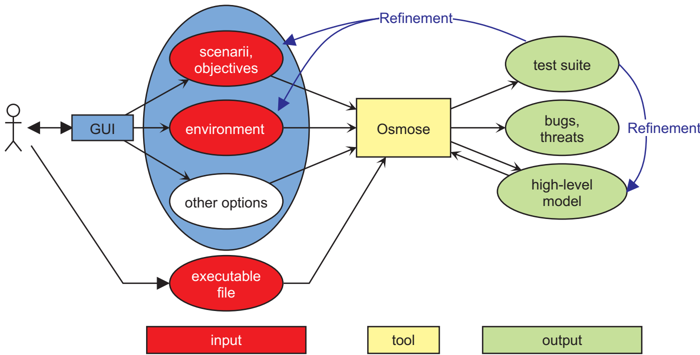


Fig. 2. User view of OSMOSE

OSMOSE does not address the oracle issue: we assume the availability of an external oracle, to which the generated test suite is passed. Yet, the tool can detect a few classes of run-time errors (typically division by 0) as well as emit warnings when an execution path contradicts user-given information (for example, the specification of dynamic targets, see Section V) or programming idioms recognised as good practice (for example, a violation of `return-call` policy).

Since we target reactive systems, we allow for the specification of volatile memory zones, i.e. part of the memory that can be non-deterministically modified (within a user-specified range). Therefore, a test data consists not only of the input of the program, but also of a sequences of values read for each volatile memory zone.

B. Intermediate language

OSMOSE is an architecture-independent tool: each instruction of the executable file under test is first translated into a small intermediate language (called DBA [6]) well-suited to formal analysis, then fed to the DSE core engine. This approach shows significant advantages:

- supporting a new architecture requires only to write a new front-end (we already support PowerPC 32, Motorola 6800 and Intel C509; x86 is in progress), then simulation and symbolic reasoning come for free;
- we can run OSMOSE on an architecture different from the one of the program under test.

Both aspects are crucial when testing embedded systems, since many different architectures are used, sometimes with very low computational power.

DBA works over a finite set of variables ranging over fixed-width bit-vectors and a finite set of (disjoint) fixed-size arrays of bytes (bit-vectors of size 8). Some of the instructions are labelled with addresses ranging over $\mathbb{N}$. There are only four basic instructions:

- `lhs := rhs, goto addr`
- `goto addr`
- `ite(cond)? goto addr : goto addr'`
- `goto expr`

Expressions and conditions are built upon a small set of standard fixed-width bit-vector operators, including (signed / unsigned) arithmetic operators, (signed / unsigned) arithmetic relational operators, logical bitwise operators, size extensions, shifts, concatenation and restriction. Contrary to real processor instructions, these operators are side-effect free. Every expression evaluates to a bit-vector of statically known size. This is not a restriction considering current Instruction Set Architectures (ISA). Conditions are expressions evaluating to a bit-vector of size 1. Expressions and operators are summarised hereafter:

- `0x0010<16>, VAR<size>`
- `@(expr, k)   // memory access`
- `expr{i .. j}   // restriction`
- `expr :: expr   // concatenation`
- $\texttt{ext}_{u,s}$`(expr,n)   // extension`
- `expr` $\{<_{u,s}, \leq_{u,s}, =, \neq, \geq_{u,s}, >_{u,s}\}$ `expr`
- `expr` $\{+, -, \times, /_{u,s}, \%_{u,s}\}$ `expr`
- `expr` $\{\wedge, \vee, \oplus\}$ `expr, !expr`
- `expr` $\{<<, >>_{u,s}\}$ `expr`

Most architectures and ISAs can be modelled accurately using the following rules. Each register in the processor is modelled by a variable in the DBA, additional variables ("local" variables) may be introduced to encode ISA instructions needing intermediate results, e.g. for side-effects such as flag updating. A single array is usually sufficient for memory. Additional arrays may allow for example to distinguish an I/O bus from a memory bus. Figure 3 presents a few examples of ISA instruction modelling through DBA. We suppose that each ISA instruction is encoded on four bytes. The ISA instruction is on the left column, and the corresponding DBA is on the right. ISA instructions are supposed to be located at address `0x5003` in the executable file. For the second example (an addition instruction), we suppose that the instruction updates a carry flag `Fc` (the carry-flag is set to 0 iff the *unsigned* addition is correct).

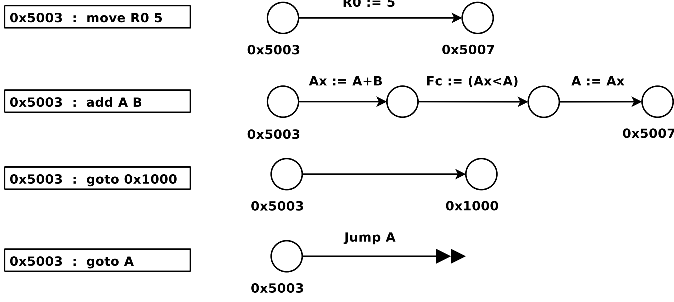


Fig. 3. DBA encoding of a few typical instructions

C. Advanced concerns

Constraint solving. OSMOSE performs bit-level reasoning through constraints expressed in the theory of bitvectors plus arrays [7]. Memory is modelled as an array of bytes, which is sufficient since safety-critical programs do not perform dynamic memory allocation. Strings are also viewed as arrays of bytes. Again, this is sufficient since the programs we consider do not rely heavily on string manipulations.

Floats are more problematic because they are commonly found in safety-critical systems. Some of the case-studies presented in the rest of the paper do contain floats, but they do not involve tricky reasoning over floating-point arithmetic (FPA) and perform only comparisons, load and store. So we encode them as simple bitvector operations. In case more complex floating-point operations are found, our solver can be naturally connected to FPA solvers based on constraint-programming [2], [20]. Yet, it must be clear that FPA solving is still a challenge in automated reasoning and software analysis.

Constraint-level optimisations include heavy preprocessing (constant and equality propagation, "dead-code" elimination, splitting a formula into independent subformulas), reuse of existing solutions and incremental solving. Such techniques are now quite common in DSE.

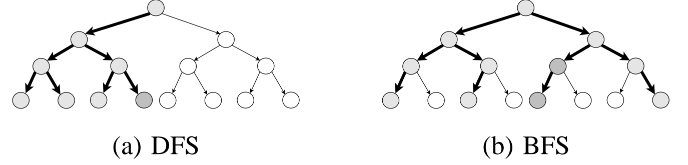
Path exploration. The path search heuristics is a standard depth-first search (DFS). Search-level optimisations include the LookAhead pruning [4], which consists in removing a path prefix when it cannot lead to yet uncovered items (branches or instructions). This technique involves performing global static reasoning within DSE, and we found it to be very effective for reducing the search space and quickly guide the exploration toward uncovered branches or instructions.

V. NEW FEATURES FOR PRACTICAL USE

We describe in this section several new features of OSMOSE which we found very useful for testing embedded programs.

A. A generic search engine

Designing a good search heuristics is a major concern in DSE. Search heuristics do not matter for path coverage, however they can make a huge difference for instruction (or branch) coverage, even more in the case of unit-level coverage. As an illustration, Figure 4 shows the instructions covered by DSE after 4 tests have been generated, in the case of (a) DFS search and (b) BFS search. Here BFS search achieves almost 50% more coverage than DFS search.



(a) DFS (b) BFS

Fig. 4. Coverage achieved by DSE for 4 tests

Yet, while DFS is considered as having a poor "coverage speed", BFS is not a panacea because it tends to get stuck into loops. Many heuristics have been proposed in the literature, such as Best First [14], Hybrid DFS [21] or Generational search [17]. Unfortunately, it appears that relying on a single heuristics is often not sufficient and that different programs require different heuristics in order to achieve very high coverage.

We design a generic search engine allowing to quickly implement search heuristics. We describe the API, then we show how to emulate existing heuristics (both standard and state-of-the-art) and finally we describe the heuristics implemented in OSMOSE.

The search API. We first need to define the *active prefixes* of a path [17]. Let us consider a path π and a branch (C,b) of the program under test covered by π, where C is a condition and b is a truth value. We denote by $\preceq$ the prefix relation between paths. A path π' is an active prefix of π if $\pi' \triangleq \pi'' \cdot (C, \neg b)$ and $\pi'' \cdot (C, b) \preceq \pi$. Intuitively, π' is obtained from π through modifying a single decision along the path.

We assume the availability of two following functions `init_val` and `update`. Function `init_val` : $\varepsilon \mapsto$ `data` creates an initial input data (an arbitrary fixed value, or a non deterministic value). Function `update : data` $\times$ `set<path>` $\mapsto$ `set<path>` launches the program on `data` and updates the set of active path prefixes, removing (resp. adding) those path prefixes covered (resp. defined) by the new execution. Our generic search engine requires the following input:

- an abstract data type `VAL` representing the score of a prefix,
- function `score : path` $\mapsto$ `VAL` for evaluating a prefix,
- function `compare : VAL` $\times$ `VAL` $\mapsto \{<, =, >\}$.

We then easily deduce function `get_best : set<path>` $\mapsto$ `path`. The generic DSE algorithm is presented in Algorithm 2.

Emulating existing heuristics. We consider `VAL` values defined by tuples formed over the following basic path information: prefix length, call-depth of the last instruction of the prefix, an integer random seed chosen at prefix creation, the generation of the prefix and the estimated gain of covering the

Algorithm 2: DSE algorithm with generic search engine

```
   Input: a program P with finite set of paths Paths(P)
   Output: TS, a set of pairs (t,σ) such that P(t) ⇝_P σ
1  TS := ∅;
2  S_paths := update(init_val(), ∅) ;
3  while S_paths ≠ ∅ do
4  |  σ := get_best(S_paths);
5  |  S_paths := S_paths \ {σ} ;
6  |  compute path predicate φ_σ for σ ;
7  |  switch solve(φ_σ) do
8  |  |  case sat(t):
9  |  |     TS := TS ∪ {(t,σ)} ;
10 |  |     S_paths := update(t, S_paths) ;
11 |  |  case unsat: skip
12 |  end
13 end
14 return TS;
```

last branch of the prefix. The last two kinds of information require a few explanations. The initial path has generation 0, and a new prefix gets the generation of its "father" plus 1. The gain estimates roughly how many uncovered items could be accessed through covering the prefix. It can be defined in different flavours, depending on the nature of the gain (optimistic, pessimistic, average) and of the quality of the estimation (typically, bounded analysis versus global static analysis), yet a higher gain is assumed to be better than a smaller one. Finally, the step of the DSE algorithm is the number of loop iterations of the main algorithm.

We use two notations for orders over `VAL`: $\mathcal{R}_{knd}$ denotes the order $\mathcal{R}$ applied to the knd element of the tuple `VAL`, and $\circ$ denotes the lexicographic order composition. For example, the order defined by $min_{depth} \circ max_{length}$ must be read as follows: it implies that `VAL` values encompasses at least a $depth$ element and a $length$ element, and $(d,l) \leq (d',l')$ iff $d < d'$ or $d = d' \wedge l \geq l'$.

We list several existing heuristics and a possible order for emulating them into our framework:

- DFS: order max_{length}
- BFS: order min_{length}
- random prefix: order min_{seed}
- hybrid-dfs [21]: we consider a static interleaving of k steps of DFS followed by k steps of random path. We use the following order: if $(step\%2k) < k$ then min_{seed} else max_{length}, where $\%$ is the integer remainder operand.
- best-first [14]: we consider a static interleaving of k steps of DFS followed by one step of "maximal gain". The order is the following: if $(step\%k) < k-1$ then min_{seed} else max_{gain}.
- generational [17]: the order is $max_{gen} \circ max_{gain}$.

Implemented heuristics. OSMOSE implements three basic heuristics (random path, DFS, BFS) and two (original) advanced heuristics: MinCall-DFS and MinCall-BFS. The last two ones are defined in the following way: the score is a pair $(length, depth)$, and the order is defined by $min_{depth} \circ max_{length}$ for MinCall-DFS, and by $min_{depth} \circ min_{length}$ for MinCall-BFS. These two heuristics give priorities to prefixes of low call-depth. They appear to be very effective in a unit testing framework where we are interested only in covering the function under test. Actually, it turns out that MinCall-DFS combined with the Look-Ahead pruning technique is a very robust choice for unit testing.

Interestingly, we do keep in OSMOSE two other testing algorithms with dedicated search heuristics: (1) a random test data generation algorithm, which can be hardly emulated through our generic API, and (2) a standard DFS-based DSE algorithm, allowing efficient memory consumption compared with its emulation through the generic API.

B. Search directives

Even cleverly-tuned search heuristics are not always sufficient to achieve full coverage, and user assistance is required in order to guide the DSE tool. We design and implement a set of *search directives* allowing to provide guidance to the core DSE engine through reducing the search space. The former version of OSMOSE offers only a directive limiting the path length (`maxLength`) and a directive defining the entry point of the test generation process (`entry`). We present a list of the new directives hereafter.

- `unsat_br (addr,bool)`: the branch going from instruction at address `addr` with condition evaluating to `bool` is considered unsatisfiable. The symbolic part of DSE will not try to fire this branch and a warning is reported if a concrete execution does, pointing out a specification issue. This directive is very useful when testing programs with many infeasible branches, typically embedded programs with software-level protections against hardware defects. It may also serve as a basic communication mechanism for combining DSE with global static analysis [10], the latter being used to detect infeasible branches.
- `exit addr`: a path reaching address `addr` is stopped. This directive is especially useful for incremental DSE (see other new features).
- `repeat addr [addr-reset] N`: the search is restricted to paths going at most `N` times through address `addr`. Considering reactive systems, the directive allows to limit the unfolding of the top-level loop. The optional `addr-reset` argument allows to reset the "loop count" each time the path goes through address `addr-reset`. The goal is typically to limit inner-loop unfolding, through specifying the loop exit address for reset point.
- `maxTryBranch (addr,bool) N`: the whole DSE process will be limited to try covering branch `(addr,bool)` at most `N` times. Once the limit is reached, active path prefixes covering the branch are discarded. The key difference with `repeat` is that the limitation does not concern a single path, hence the two directives are complementary. It can be useful in an incremental DSE setting in order to

detect likely-unsatisfiable branches (then check their status, and possibly add an `unsat` directive).

Finally, the user is also given the possibility to restrict possible values of any variable or memory location (`lhs`) to values in a given interval or list, through a `restrictDomain lhs dom [addr]` directive. The restriction can be global or local to a given address. This option is commonly found in DSE tools.

C. Goal-oriented testing

DSE is fundamentally a blind forward approach, well-suited for covering large sets of items such as paths, branches or instructions. The former version of OSMOSE proposed only these three coverage objectives. Yet, it is often useful to support more goal-oriented objectives, typically covering one specific instruction. We add support for this kind of objective, specifying a set of branches or instructions as objective sets. It integrates smoothly within our existing coverage-based DSE framework: instead of precomputing the set of all targets (branches or instruction, unit mode or not), we simply let the user specify it. While DSE is clearly not the most appropriate choice here (a backward approach would seem better [12]), combining DSE with LookAhead pruning provides decent results for goal-oriented testing.

D. Test suite replay and completion

We implement the possibility of storing generated test suites and then replaying them in OSMOSE. In replay mode, the tool checks consistency between the intended execution trace and the observed execution trace. Moreover, the DSE algorithm can now be started from an existing test suite. The tests are all run and corresponding active prefixes are generated. Then the DSE loop is launched. These two simple feature show several significant practical advantages:

- **Validation:** the test suite can be replayed in a trusted simulation environment, comparing observed execution traces and coverage information to the information output by OSMOSE. It proves useful for both external validation purposes and internal debugging concerns.
- **Test suite completion:** it provides the ability to complete existing test suite, regarding how the tests have been obtained (manual testing, random testing, etc.). Especially, it allows to integrate smoothly binary-level testing within an existing testing process.
- **Complex search heuristics:** test replay is a very convenient way for chaining different search heuristics, through launching DSE with one heuristics, saving the tests and completing them with another heuristics.

E. Output of concrete or symbolic states

Many reactive systems present a long initialisation / self-testing phase. It makes path space exploration expensive, since the behaviours of the core functionalities are mixed up with this first step. Moreover, one can note that initialisation is often deterministic with only a very few number of (very long) paths.

To cope with this issue, we add the possibility to output a summary of all the paths starting from an entry address and reaching a given target address. Together with the already available features for specifying an entry address and an initial memory state, the user can try to analyse the program under test in a compositional manner.

Memory state description includes constant values, range constraints as well as equalities between some registers or memory locations. This is a very restricted specification mechanism, so it cannot model properly the output of any piece of code. Hence the technique must be used with care:

- loss of completeness: if some paths to be summarised have been discarded (typically, a too small path length limit) then the resulting memory state will not be complete and subsequent DSE can miss feasible paths;
- loss of correctness: if the intended summary falls out of our memory state formalism, then subsequent DSE will be run on an over-approximated memory state and correctness may be compromised. One can still replay the tests in OSMOSE on the whole program to check for inconsistencies in the test suite.

Yet, while this feature cannot be used in any setting, it proves very valuable for some simple cases of modular reasoning, especially the initialisation or self-testing phases.

F. Specification of dynamic targets

Dynamic jumps are instructions of the form `goto R` where the value of `R` may vary at runtime. They pose a serious issue in binary-level program analysis since they prevent an easy syntactic recovery of the control-flow graph [8]. DSE is much less sensitive to this issue [5] since one needs only to recover a subset of feasible jump targets. Yet, in presence of dynamic jumps the recovered CFG is only partial, posing two problems: (1) the coverage measure cannot be trusted (which is problematic for critical systems), and (2) the precision of LookAhead pruning is affected (any dynamic jump must be considered to point anywhere), reducing the ability to prune irrelevant paths.

We add the possibility for the user to specify sets of dynamic targets. The specification is of the form: `jump@addr leads to: addr1 addr2 ...` , meaning that the instruction at address `addr` is a dynamic jump which can lead only to addresses contained in `addr1`; `addr2`, etc. (over-approximation). The benefits are the following:

- LookAhead performs better,
- the coverage can be trusted if all possible targets have been specified.

The important point here is that the specification must be an over-approximation. Additional (infeasible) targets will be discarded by symbolic reasoning: they can slowdown the resolution process, but they cannot affect the correctness guarantee of DSE. However, missing targets will hide legitimate parts of the program to the DSE procedure, leading to overly optimistic coverage measure and incomplete testing (within the limited search space).

Since specifying dynamic targets can be error-prone, OSMOSE performs several consistency checks. We report a warning whenever the instruction at address `addr` is not a dynamic jump or the dynamic jump at address `addr` does lead to an address outside the specified set. In the second case, we systematically check this possibility for each prefix path through a constraint of the form $\phi \wedge e \notin \{addr_1, \ldots, addr_n\}$, where e is the jump expression (`R` in the example given above) and ϕ is a path predicate.

Dynamic target specifications can be obtained either manually or through dedicated static analysis tools. In aeronautics for example, validation engineers using the widespread aiT tool [15] for wcet computation already have to provide such sets of targets. It is a much easier task if the executable code comes from a home-compiled source code. In that case most targets can be retrieved automatically with help from the compiler. When the source code and the compilation chain are not available, it becomes a much more challenging problem. Recent binary-level static analysis techniques try to address it. We implement such a connexion between our own CFGBuilder prototype [8] and OSMOSE: CFGBuilder is first used for computing upper-sets of targets, then these sets are used as input for OSMOSE (cf. Figure 5).

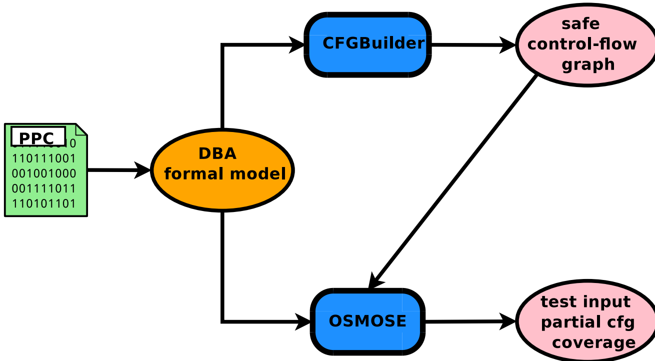


Fig. 5. Binary-level analysis

VI. CASE-STUDIES

A. Unit-level testing of medium-size application

The first case-study is the automatic binary-level testing of a critical embedded program from aeronautics (from SAGEM). The program is part of the control-system of an aircraft engine. It is designed in SCADE, and a C program is automatically generated. The underlying architecture is a PowerPC 32-bit. The program under test is medium size: it is divided into 250 procedures for about 30,000 machine instructions. Maximal call-depth is 10.

The goal was to use OSMOSE in an automatic manner in order to assess its unit-level coverage abilities. We perform two sets of experiments: we run unit-level test generation over 40 procedures with small call-depths (between 0 and 4), and we also run it on a few procedures with higher call-depths (between 6 and 10). We measure the achieved coverage, and we also compare it to the coverage achieved through random testing (built on top of OSMOSE simulator, run with a budget of 20x more tests than those required by OSMOSE). The DSE search heuristics is DFS, and LookAhead pruning is activated. Procedures under test are randomly chosen.

TABLE I. UNIT-LEVEL TESTING OF AN AERONAUTIC APPLICATION

name	#I	#Br	OSMOSE cover	OSMOSE time (sec. †) [#tests ‡]	random cover	random time (sec. †)
plane0	237	36	100%	10 [19]	40%	20
plane1	290	140	98%	60 [43]	64%	100
plane2	201	72	100%	10 [37]	35%	20
plane3	977	190	50%	60 [3]	96%	60
plane4	2347	500	87%	600 [15]	68%	600
plane5 (*)	121 4103	2 509	100%	1 [2]	100%	10
plane6 (*)	250 425	18 34	94%	100 [9]	83%	120
plane7 (*)	506 15640	20 2790	80%	20 [4]	75%	500
plane8 (*)	957 30969	14 4952	14%	10 [3]	50%	500
plane9 (*)	627 31793	74 5034	77%	600 [12]	63%	600

†: Time is given in seconds, rounded to the nearest ten.

‡: only tests providing incremental branch / instruction coverage are output to the user

(*): first line give #I and #Br for the function under test (FUT) only, second line give #I and #Br for the FUT and all its callees

Part of the data is available in Table I. We report the following results.

- Very good coverage results have been achieved for procedures with low call-depth (even for some procedures with up to 2,000 instructions). Actually, full coverage was achieved for 31 procedures out of 40, good coverage ($75\% - 95\%$) for 4 procedures, and unsatisfactory coverage ($< 50\%$) for the remaining 5 procedures.
- Results are much mitigated for procedures with high call-depth, the path explosion problem becoming a serious issue. OSMOSE still achieves good coverages on some procedures, but results are less robust.
- OSMOSE performs better than random testing for the vast majority of the procedures, in terms of both time and coverage, while on each procedure random testing generates 20x more test data.

B. Full testing of a small application

The second case-study is another aircraft program from SAGEM. Yet, the context of analysis is significantly different: the program is rather small (17 procedures, 2,600 instructions and 113 branches), but the goal is to perform full testing (i.e., cover every branch starting from the top-level entry point) and the program is recognised by testing teams to be hard to analyse.

The program was a challenge for OSMOSE, and we had to use many of the new features described in this paper to achieve high coverage, namely: advanced search heuristics (Call-BFS, Call-DFS) and several search directives (`unsat_br` and `repeat`). We list here the main difficulties.

- First of all, many branches are simply hard to cover, in the sense that only very few paths exercise them. Random testing or a direct use of OSMOSE with DFS stuck to 50% branch coverage. Going further requires careful guidance. Indeed, many paths turns out to be infeasible (we count 2x more infeasible paths than feasible ones).
- Second, covering all branches require to explore a huge path space. There are two difficulties here: (1) one loop requires to be unrolled at least 380 times, leading to long paths; and (2) a volatile memory zone (time register) is read within a simple loop ("read-loop"), leading to many artificial paths.

We proceed the following way, taking advantage of the new features described in Section V.

- Search directives have been used to reduce the search space, through restricting the main loop (we put a limit of 400) with the `repeat` directive, and restricting each read-loop on volatile memory to a single read each time the loop is entered (the `reset` option is required here). Hence we can still search for long paths (required by the program) without a major blowup of the number of paths.
- We use a combination of Call-BFS and Call-DFS in order to achieve most of the coverage by exercising top-level branches in priority. Then we use DFS (with LookAhead) for the last remaining branches.

We achieved the following results: 15 procedures out of 17 have been fully covered, the two remaining leaf procedures being covered at only 50% (for branches). The two uncovered procedures are library functions, and some of their branches have been shown to be uncoverable within the context of the program under test. Yet, we did not investigate every uncovered branch.

C. Software comprehension and testing

The third case study is an embedded program used in power plant, provided to EdF (major French energy provider) by a third-party vendor. We do not work directly on the original program, but on a representative version specifically created by the vendor for the case-study. The goal is both to understand better the program behaviour and structure, and to automatically test it.

The program is written in assembly language for the Motorola 6800 architecture. It counts about 3,000 instructions divided into an initialisation module, 10 computation modules (forming the main loop of the program) and 10 library functions. Again, while rather small, the code is difficult to analyse: many branches are unsatisfiable (self-test), the initialisation step is long and complex, and each main loop cycle adds about 1,000 branches. Moreover, the documentation provided by the vendor is very sparse, making it difficult to identify all information needed for OSMOSE initial state specification (volatile memory zones).

We first use OSMOSE for recovering missing information about infeasible branches. We test each library function / module in isolation within the most generic input context. We manage to detect 65 uncovered branches. Manual inspection proves that all of them are indeed infeasible. In the rest of the analysis, we use the new keyword `unsat` for preventing OSMOSE to try covering these branches. Inspecting the infeasible paths also allows us to detect likely-volatile memory zones and a few important constant values not referenced in the available documentation. Asking the third-party vendor, it turns out that our intuition is correct, and that we have obtained all the information required to define a correct initial memory state.

We then try to generate a test suite starting from the entry point of the program. We limit the number of cycles of the top-level loop to 1. With this parameter, OSMOSE needs 35 min to generate 6 tests achieving 70% coverage of branches. When `unsat` directives are added, computation time drops to 25 min for the same result. This result contradicts the documentation statement that "all branches (except consistency checks) should be feasible within the first cycle".

We also try a modular approach: the initialisation module is analysed alone, and we record the memory state obtained at the end of the module. We save it and launch the analysis on the other 10 modules, starting from the saved memory state. Results are remarkable: a similar test suite is generated, yet computation time drops to only 2 min. Finally, test generation over more cycles (2 and 3) appears to be much more expensive and leads to the same coverage.

As a conclusion, on this example OSMOSE proves to be a powerful tool for recovering meaningful facts about the program under test (infeasible branches, entries, etc.). Moreover, it helps to pinpoint problems in the documentation (acknowledge by the vendor). The modular approach to DSE allowed by the new features of OSMOSE seems promising, especially for programs with complex initialisation phase.

D. Automatic testing and test suite completion

The first goal of the case-study is to evaluate the binary-level coverage achieved by several source-level testing methods and compared them with those achieved by OSMOSE. Then, when OSMOSE performs better, we want to check the feasibility of completing the source-level test suite by OSMOSE.

We consider function blocks defined in `Scicos`, a graphical dynamical system modeller and simulator [23]. Programming based on function blocks is a widespread approach in the development of automation and control systems. We consider 4 basic blocks commonly found in control applications: signal saturation, integration, product and selector. The block for integral is depicted in Figure 6. Blocks are written in C then compiled into PowerPC 32. Once compiled, the blocks have between 200 and 580 instructions, and between 10 and 86 branches.

We consider the following coverage criteria: instruction (I), branch (B) and MCDC. We add $_{s}$ or $_{b}$ to distinguish between source-level and binary-level. We compare test suites generated

```c
void int_euler(scicos_block *block, int flag){
  int i;
  int **_work=GetPtrWorkPtrs(block);
  int *_u1=GetInPortPtrs(block, 1);
  int *_y1=GetOutPortPtrs(block, 1);
  int *_rpar=GetRparPtrs(block);
  int dt=_rpar[0];
  int *rw;

  /* conditional statement inserted to prevent from
       segmentation faults */
  if(_u1 != NULL && _y1 != NULL && GetInPortRows(block
       ,1) <= 5){

    if(flag == 1){
      rw=*_work;
      for(i=0 ; i<GetInPortRows(block,1) ; i++){
        _y1[i]=dt*_u1[i]+rw[i];
        rw[i]=_y1[i];
      }
    }
    if(flag == 4){
      if((*_work= scicos_malloc(sizeof(int)*
      (1+GetInPortRows(block,1)) ,block)) == NULL ){
        set_block_error(-16);
        return;
      }
      rw=*_work;
      for(i=0 ; i<GetInPortRows(block,1) ; ++i){
        rw[i]=_rpar[1];
      }
    }
    if(flag == 5){
      scicos_free(*_work);
    }
  }
}
```

Fig. 6. Integral processing

by OSMOSE to the following generation strategies: (manual) functional testing through partition testing and boundary testing, (manual) test suites achieving source-level instruction / branch / MCDC coverage, and test suites generated by CREST [11], a source-level DSE tool.

We record results in tables such as those depicted in Table II. We can observe that OSMOSE provides better coverage than other testing strategies, whatever criterion is considered. Notably, OSMOSE provides better binary-level branch coverage than (manual) functional testing, (manual) source-level MCDC coverage and source-level DSE. Moreover, experiments show that OSMOSE is indeed able to complete existing test suites. For example, Table II shows the coverage achieved by OSMOSE when completing CREST test suite. This is interesting from a testing methodology point of view: binary-level testing does not have to be performed from scratch, instead it can be smoothly integrated within standard testing processes and takes advantages of existing test suites in order to focus test generation only on uncovered parts of the program under test.

From these experiments, we tried to understand why binary-level DSE seems more powerful than source-level strategies. We found mainly three reasons. First and obvious reason, decisions with multiple conditions at the source-level are compiled in separate decisions at the binary level. Second (less obvious) reason, it appears that `nop` instructions are sometimes added to empty branches during compilation, forcing binary-level instruction coverage to covers them. Finally, binary-level tools are more amenable to reason on low-level data manipulation than source-level tools, and sometimes they manage to cover branches that are (wrongly) considered uncoverable by tools or experts working at a higher level of description.

Finally, concerning the relationship between source-level coverage criteria and binary-level coverage criteria, the present work can be seen as an empirical complement to the work done in COUVERTURE [1], where the authors identify assumptions under which binary-level branch coverage implies source-level MCDC coverage. On the one hand we do not impose any hypothesis on the compilation chain, on the other hand our results are experimental rather than theoretical.

VII. RELATED WORKS

DSE tools have blossomed up recently [3], [11], [13], [14], [16], [24]–[26], and the technique has been heavily used in recent years for finding bugs in desktop programs. Most of these tools work from source code and target bug finding rather than full coverage. SAGE [17] is a famous binary-level DSE tool. Yet, there are a number of significant differences with our own work: SAGE is heavily optimised toward finding bugs in very large x86 applications in a fully automatic way, while we are interested in full coverage of (smaller) embedded programs run on different architectures, with possible guidance from validation experts. Our generic search API gets inspiration from the Fitnex approach [27] of PEX [25].

VIII. CONCLUSION

In this article, we discuss the use of binary-level DSE for testing safety-critical embedded systems. We present several innovative features of practical interest when analysing such systems, including a generic search API, search directives for reducing the exploration space and test suite replay & completion. These features have been implemented in our DSE tool OSMOSE, and we show through four real-life case-studies how they can be used in practical situations.

REFERENCES

[1] Bordin, M., Comar, C., Gingold, T., Guitton, J., Hainque, O., Quinot, T.: Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework. In: Embedded Real Time Software and Systems (ERTSS 2010)

[2] Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Journal of Softw. Test., Verif. Reliab., 16(2), 2006

[3] Bardin, S., Herrmann, P.: Structural Testing of Executables. In: ICST 2008. IEEE Computer Society, Los Alamitos (2008)

[4] Bardin, S., Herrmann, P.: Pruning the search space in path-based test generation. In: ICST 2009. IEEE Computer Society, Los Alamitos (2009)

[5] Bardin, S., Herrmann, P.: OSMOSE: Automatic Structural Testing of Executables. Softw. Test., Verif. Reliab. 21(1): 29-54(2011)

[6] Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A.: The BINCOA Framework for Binary Code Analysis. In: CAV 2011. Springer, Heidelberg (2010)

[7] Bardin, S., Herrmann, P., Perroud, F.: An Alternative to SAT-based Approaches for Bit-Vectors. In: TACAS 2010. Springer, Heidelberg (2010)

[8] Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG Reconstruction from Unstructured Programs. In: VMCAI 2011. Springer, Heidelberg (2011)

[9] Balakrishnan, G., Reps, T. W., Melski, D., Teitelbaum, T.: WYSINWYX: What You See Is Not What You eXecute. In: IFIP Working Conference on Verified Software: Theories, Tools, Experiments. (2005)

[10] Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Combining Static Analysis and Test Generation for C Program Debugging. In: TAP 2010. Springer, Heidelberg (2010)

TABLE II. COVERAGE MEASUREMENTS FOR Scicos BLOCKS

(1) Integration block

	Functional testing	CREST	I_s (manual⋆)	B_s (manual⋆)	$MCDC_s$ (manual⋆)	OSMOSE	Completion of CREST
I_s	100 %	100 %	100 %	100 %	100 %	100 %	100 %
I_b	98.7 %	97.9 %	97.9 %	98.7 %	98.7 %	98.7 % %	98.7 %
B_s	100 %	90 %	85 %	100 %	100 %	100 %	100 %
B_b	85.2 %	73.5 %	70.5 %	79.4 %	85.2 %	88.2 %	88.2 %

(2) Saturation block

	Functional testing	CREST	I_s (manual⋆)	B_s (manual⋆)	$MCDC_s$ (manual⋆)	OSMOSE	Completion of CREST
I_s	100%	100%	100%	100%	100%	100%	100%
I_b	99%	99%	100%	100%	100%	100%	100%
B_s	88.4%	80.7%	100%	100%	100%	100%	100%
B_b	86.6%	76.6%	93.3%	100%	100%	100%	100%

(3) Product block

	Functional testing	CREST	I_s (manual⋆)	B_s (manual⋆)	$MCDC_s$ (manual⋆)	OSMOSE	Completion of CREST
I_s	100%	100%	100%	100%	100%	100%	100%
I_b	100%	100%	100%	100%	100%	100%	100%
B_s	100%	100%	83%	100%	100%	100%	100%
B_b	87.5%	100%	75%	87.5%	100%	100%	100%

⋆: manually-crafted test suite achieving 100% coverage for the specified criterion

[11] CREST: automatic test generation tool for C. http://code.google.com/p/crest.

[12] Charreteur, F., Gotlieb, A.: Constraint-Based Test Input Generation for Java Bytecode. In: ISSRE 2010. IEEE Computer Society, Los Alamitos (2010)

[13] C. Cadar, D. Dunbar, D. Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: OSDI 2008. Usenix Association (2008)

[14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler. EXE: automatically generating inputs of death. In: CCS 2006. ACM, New York (2006)

[15] Ferdinand, C., Heckmann, R.: aiT: worst case execution time prediction by static program analysis. In: IFIP Congress Topical Sessions 2004. Kluwer, Dordrecht (2004)

[16] Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: PLDI 2005. ACM, New York (2005)

[17] Godefroid, P., Levin, M. Y., Molnar, D.: Automated Whitebox Fuzz Testing. In: NDSS 2008.

[18] Godefroid, P., Levin, M. Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Commun. ACM 55(3): 40-44 (2012)

[19] King, J. C.: Symbolic execution and program testing. Communications of the ACM, 19(7), july 1976.

[20] Marre, B., Michel, C.: Improving the Floating Point Addition and Subtraction Constraints. In: CP 2010. Springer, Heidelberg (2010)

[21] Majumdar, R., Sen, K.: Hybrid Concolic Testing. In: ICSE 2007. IEEE, Computer Society, Los Alamitos (2007)

[22] Software Considerations in Airborne Systems and Equipment Certification. In: RTCA 1992.

[23] SCICOS: Block diagram modeler/simulator. http://www.scicos.org

[24] Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: ESEC/FSE 2005. ACM, New York (2005)

[25] Tillmann, N., de Halleux, P.: Pex-White Box Test Generation for .NET. In: TAP 2008. Springer, Heidelberg (2008)

[26] Williams, N., Marre, B., Mouy, P.: On-the-Fly Generation of K-Path Tests for C Functions. In: ASE 2004. IEEE, Computer Society, Los Alamitos (2004)

[27] Xie, T., Tillmann, N., de Halleux, P., Schulte, W.: Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In: DSN 2009. IEEE, Los Alamitos (2009)