Symbolic Execution of Satellite Control Procedures in Graph-Transformation-Based EMF Ecosystems

Nico Nachtigall, Benjamin Braatz, and Thomas Engel

Université du Luxembourg, Luxembourg
firstname.lastname@uni.lu

Abstract. Symbolic execution is a well-studied technique for analysing the behaviour of software components with applications to test case generation. We propose a framework for symbolically executing satellite control procedures and generating test cases based on graph transformation techniques. A graph-based operational symbolic execution semantics is defined and the executed procedure models are used for generating test cases by performing model transformations. The approach is discussed based on a prototype implementation using the Eclipse Modelling Framework (EMF), Henshin and ECLiPSe-CLP tool ecosystem.

Keywords: symbolic execution, graph transformation, test case generation, triple graph grammars, EMF henshin

1 Introduction

Symbolic execution [4] is a well-studied technique for analysing the behaviour of software components with applications to test case generation. The main idea is to abstract from possibly infinite or unspecified behaviour. Uninitialised input variables or external function calls are represented by symbolic variables with the sets of possible concrete input values as value domains. Consequently, symbolic program execution is rather based on symbols than on concrete values leading to symbolic expressions which may restrict the value domains of the symbols. For each execution path, a path constraint (PC) is defined by a dedicated boolean symbolic expression. Solving the expression, i.e., finding a valuation for all contained symbolic variables so that the expression is evaluated to true, provides concrete input values under which the corresponding path is traversable. An execution path is traversable as long as its path constraint is solvable.

In this paper, we propose a framework for symbolically executing satellite control procedures (SCPs) and generating test cases based on graph transformation techniques [6]. We successfully apply graph transformation techniques in an industrial project with the satellite operator SES (Société Européenne des Satellites) for an automatic translation of SCPs from proprietary programming languages to SPELL (Satellite Procedure Execution Language & Library) [8]. The safety-critical nature of SCPs implies that extensive testing is required after

translation. The presented approach allows us to generate tests without leaving this graph-transformation-based ecosystem. We define a graph-based operational semantics for symbolically executing SCPs for a subset of the SPELL language. The executed procedure models are used for generating test cases by performing model transformations. We discuss our approach based on a prototype implementation using the EMF Henshin [7] and ECLiPSe-CLP [9] tools.

Sec. 2 introduces the symbolic execution framework. In Sec. 3, the graph-based operational symbolic execution semantics is defined. Sec. 4 presents model transformation rules for test case generation and a prototype implementation for the approach. Sec. 5 concludes the paper and compares with related work.

2 Models & Symbolic Execution Framework

The symbolic execution framework in Fig. 1 uses the abstract syntax tree (AST) of a procedure, which defines a typed attributed graph [6]. The AST can be constructed by parsing the source code with appropriate tools (see Sec. 4). The AST graph is symbolically executed using two graph transformation systems (GTSs). A GTS is a set of graph transformation rules where each rule may create or delete nodes and edges or update attribute values. In phase one, the GTS GTS_{Flow} is used to annotate AST with execution flow information leading to graph $AF = AST + FLOW$. In phase two (symbolic execution), the GTS GTS_{Sym} is exhaustively applied to AF leading to graphs $State_i = AF + SYM_i, i = 1..n$ representing the status of the execution.

Fig. 2 shows the running example in a small subset of the SPELL language [8]. SCP "Charge Batteries" retrieves the state of charge (SC) of both batteries of a satellite, defines a minimal threshold min of 50%, and switches to the battery with higher SC if it exceeds min. Otherwise, an alert is issued. Meta-model SCP specifies the general syntax of ASTs for such procedures. A procedure Proc consists of a list of statements Stmnt (assignments Asg, function calls FnCall, definitions FnDef or branching If structures) with explicit next pointers. Function calls contain a list of arguments (arg) and function definitions contain a list of parameters (pm). An assignment contains a variable (var) and an assigned expression (ex). Expressions (Expr) are either numbers (Number), variables (Var) or Boolean expressions (Bool) with operator ($<, <=, >, >=, and, or$) and operands (left (le) and right (ri)). Complex statements (If,FnDef) contain a block (B) that references a list of statements. Furthermore, If statements have a boolean condition cond and may have else and ElIf structures (edge el). The AST for the procedure is a graph typed over meta-model SCP. Graph $FLOW$ represents the flow annotation of AST. Places P are assigned (dotted edges asg) to nodes in AST that should

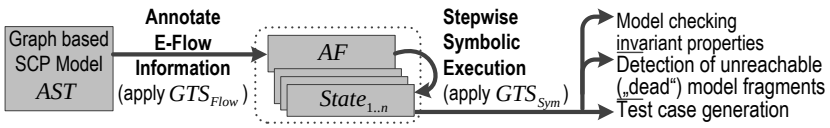

Fig. 1. Steps of Symbolic Model Execution

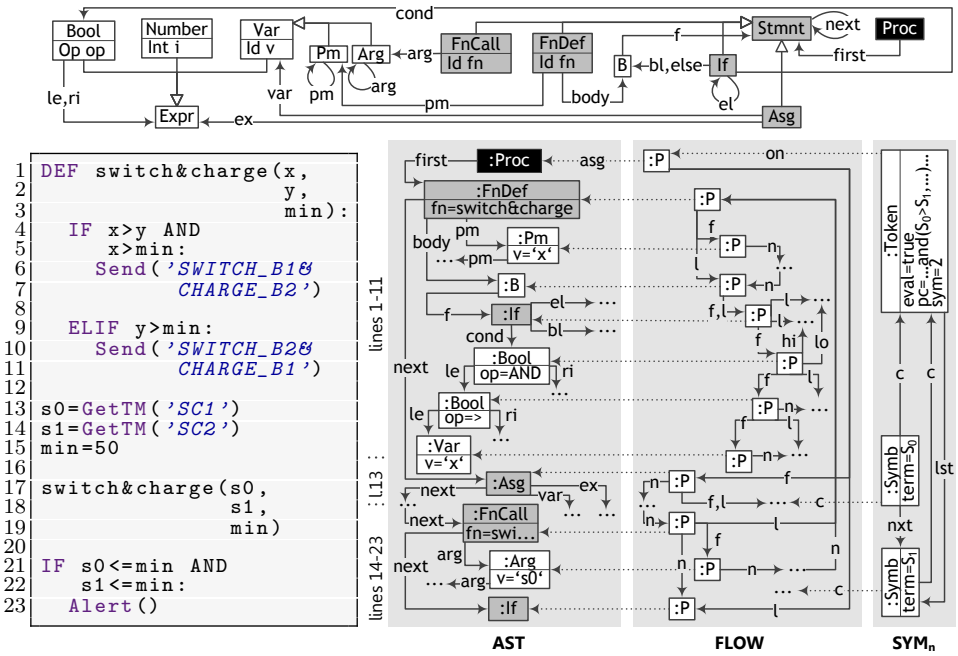

Fig. 2. SCP meta-model (top), SCP "Charge Batteries" (left), SCP model (AST), flow annotation (FLOW) and symbolic execution elements (SYM_n)

be executed. P nodes can be connected by f,l or n edges in order to indicate which other nodes need to be executed at first, next or last before finishing the execution of a node, e.g., in order to execute the procedure (node Proc), the assignment in line 13 (node Asg) needs to be executed first. For each execution path, a Token representing the current execution point with path constraint is created (in total six Tokens for the example). In graph SYM_n, the Token node on place P that is assigned to node Proc indicates that the procedure was evaluated (eval=true) with path constraint pc=and(and(S_0>S_1,S_0>50),not(and(S_0<=50,S_1<=50))) and symbolic variables S_0, S_1 (Symb) for GetTM('SC1') and GetTM('SC2') by entering line 6 but not line 23. The resulting graphs $State_{1..n}$ can be used for model checking invariants, detection of dead model fragments or test generation.

3 Operational Execution Semantics

The execution semantics is divided into the execution flow of the AST graph and the token semantics for traversing all flow paths. Fig. 3 shows the rules of GTS_{Flow} and GTS_{Sym} in short-hand notation, i.e., nodes and edges marked with <+> are created, those marked with <-> are deleted and attribute values of the form [x=>y] are updated from x to y when applying the rule. Nodes marked with <tr> have a "hidden" translation attribute that is updated [false=>true] during rule application so that the rule is only applied once. A more formal definition of graph transformation in general is given in [6].

Rule Init_1 specifies that the first statement of a procedure needs to be executed first. An initial token is put on the first place with path constraint true and eval=false. Rule Stmnt_1 defines that successive statements need to be executed successively. Note that the first statement in Fig. 2 is a FnDef. Therefore, another rule defines that the succeeding statement needs to be executed first until there is no more FnDef. Rule Asg_1 defines that the expression of an assignment needs to be evaluated first before assigning the resulting value. The rule for blocks is defined analogously. Rule Bool_1 defines that the left operand has to be evaluated before the right operand. Rule If_1, branches the flow - the condition is evaluated before executing the block (hi - positive condition) or an "empty" place (lo - negative condition). Rule EIIf_1 links the "empty" place of rule If_1 to the alternative If. Rule Else_1 is defined analogously.

Rule TFst_2 moves the token to the first child place (edge f) as long as possible. Rule TNxt_2 moves the token of an evaluated place to the next place (edge n) and changes attribute eval to false. The rules implement a left-most inner-most evaluation strategy. Rule GetTM_2 evaluates each GetTM-FnCall to a path-wide unique symbolic variable (Symb) S_i (uniqueness is given by token attribute sym which is increased by one). Note that Symbs are ordered in their occurence of evaluation which is important for a later test generation. Rule GetTM_2 requires that a last Symb already exists. An analogue rule creates a last Symb if not existent. Rule Asg_2 assigns the term of the evaluated expression to the variable Var and sets the assignment as evaluated by moving token edge on. Rule $\mathsf{BoolAnd}_2$ concatenates

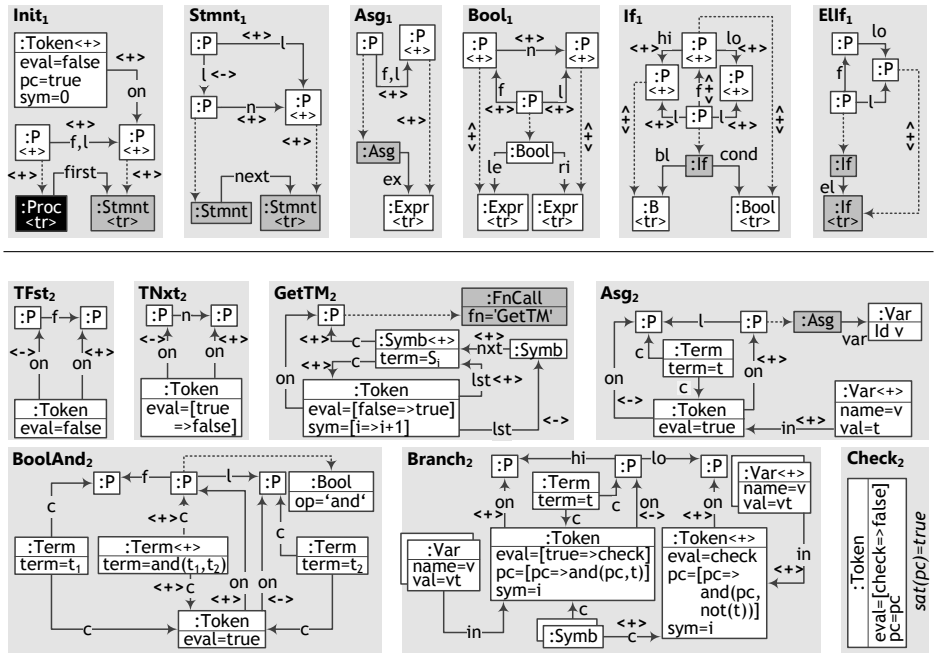

Fig. 3. Rules for annotation (top, GTS_{Flow}) and symbolic execution (bottom, GTS_{Sym})

both evaluated operands with and. Analogue rules for boolean expressions with other operators (or,<,>,etc.) are defined. Amalgamated rule Branch_2 duplicates the token with all connected variables and symbols, negates the condition (not(t)) for the lo path and concatenates the condition with the path constraint. Rule Check_2 checks, if the path constraints of duplicated tokens are still satisfiable after a branch (attribute condition sat(pc)=true). If the path constraint of a token becomes unsatisfiable, the token status eval remains check and the token can not be moved any more. After simulating the example in Fig. 2, three tokens from six possible paths are assigned to node Proc with eval=true while the other three tokens remain at the last If statement with unsolvable path constraints. Only tokens that are assigned to node Proc with eval=true are considered during test case generation (they represent execution paths with solvable path constraints). Additional rules are defined for annotating and traversing function calls and definitions. A function is traversed every time it is called so that global side effects in execution can be respected, e.g., operations on call by reference arguments.

4 Implementation & Test Case Generation

A procedure is parsed with Xtext [5] to an EMF AST graph first. Then, the EMF Henshin tool [7] is used in combination with the ECLiPSe constraint solver [9] to execute the AST graph by automatic rule applications and satisfiability checking / solving constraints. The AST graph is completely preserved during execution. Correspondences between symbolic variables of path constraints (nodes of type Symb) and AST graph structures are used for test generation by applying the forward model transformation (FT) rules in Fig. 4. An FT rule [8] consists of a source graph G_S, correspondence graph G_C and target graph G_T. While G_S is parsed, nodes and edges in G_C, G_T are created. Applying the rules yields a graph that is serialised to test case files with Xtext. Rule Proc2Test creates a Test suite for a procedure. Rule Token2Case creates a test case for each execution path with solvable path constraint. Rule LstSymb2KeyElem adds a key tm for each last (edge lst) evaluated GetTM(tm) function call of an execution path to the test case with a list containing a test input valuation for symbolic variable s as first (edge fst) element so that path constraint pc is satisfied (solve(s,pc)). A second rule handles all previous symbolic variables. Symbolic variables are ordered in order to reflect the sequential execution order of the represented GetTM function calls which is needed for proper test generation with test inputs in correct order.

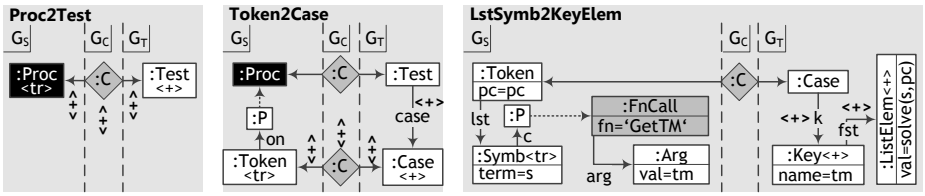

Fig. 4. Rules for test case generation

5 Conclusion & Related Work

We have presented a framework for symbolically executing simple satellite procedures. The approach preserves the correspondences between symbolic variables and the AST, so that the result graph can be used for test case generation by performing model transformations afterwards. A prototype implementation for the symbolic execution and test case generation framework has been presented.

In contrast to interpreter based symbolic execution engines [1,3] of programming languages, our graph-transformation-based approach allows symbolic execution on a more abstract level enabling its formal analysis and application to other languages and behavioural diagrams in future work. In [2], an abstract symbolic execution framework based on term rewriting is proposed. In contrast to our approach, the correspondences between symbolic variables and the program term are not preserved. Research on how to transfer and enhance results from term to graph rewriting approaches for symbolic execution is topic of future work. In [10], the execution of UML state machines is presented but diagram–path-constraint correspondences are not specified explicitly.

In future work, we plan to extend the symbolic execution semantics for applicability to industrial SPELL SCPs [8] and analyse its correctness w.r.t. a formal SPELL semantics that needs to be defined first. We will investigate important properties of symbolically executed models that should be preserved during model refactorings and how to ensure their preservation. Moreover, we will assess the scalability of our approach.

References

1. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to java pathfinder. In: TACAS. pp. 134–138 (2007)
2. Arusoaie, A., Lucanu, D., Rusu, V.: A Generic Approach to Symbolic Execution. Tech. Rep. RR-8189, INRIA (Dec 2012)
3. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008)
4. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 82–90 (Feb 2013)
5. The Eclipse Foundation: Xtext (2013), `http://www.eclipse.org/Xtext/`
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation, vol. EATCS Monographs in Theoretical Computer Science. Springer (2006)
7. EMF Henshin (2013), `http://www.eclipse.org/modeling/emft/henshin/`
8. Hermann, F., Gottmann, S., Nachtigall, N., Braatz, B., Morelli, G., Pierre, A., Engel, T.: On an Automated Translation of Satellite Procedures Using Triple Graph Grammars. In: Proc. ICMT'13, LNCS, vol. 7909, pp. 50–51. Springer (2013)
9. Schimpf, J., Shen, K.: Eclipse - from lp to clp. Theory and Practice of Logic Programming 12, 127–156 (2012)
10. Zurowska, K., Dingel, J.: Symbolic execution of UML-RT State Machines. In: Proc. of SAC '12. pp. 1292–1299. SAC '12, ACM (2012)