Abstract State Machines Mutation Operators

Jameleddine Hassine
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, Kingdom of Saudi Arabia
jhassine@kfupm.edu.sa

***Abstract*—Mutation testing is a well established fault-based technique for assessing and improving the quality of test suites. Mutation testing can be applied at different levels of abstraction, e.g., the unit level, the integration level, and the specification level. Designing mutation operators represents the cornerstone towards conducting effective mutation testing and analysis. While mutation operators are well defined for a number of programming and specification languages, to the best of our knowledge, mutation operators have not been defined for the Abstract State Machines (ASM) formalism. In this paper, we define and classify mutation operators for the Abstract State Machines (ASM) formalism. The proposed ASM mutation operators are illustrated using examples written in the CoreASM language. Furthermore, we have developed a tool for automatic generation of mutants from CoreASM specifications.**

***Keywords-Mutation testing*; *specification*; *mutation operator*; *Abstract State Machines (ASM)*; *CoreASM*.**

I. Introduction

Mutation testing [1] is a well established fault-based testing technique for assessing and improving the quality of test suites. Mutation testing uses mutation operators to introduce small changes, or mutations, into the software artifact (i.e., source code or specification) under test. A mutant is produced by applying a single mutation operator, and for each mutant a test is derived that distinguishes the behaviors of the mutated and original artifact.

In a recent survey on the development of mutation testing, Jia and Harman [2] have stated that more than 50% of the mutation related publications have been applied to Java [3], Fortran [4] and C [5]. Although mutation testing has mostly been applied at the source code level, it has also been applied at the specification and design level [6][2]. Formal specification languages to which mutation testing has been applied include Finite State Machines [7][8][9], Statecharts [10], Petri Nets [11] and Estelle [12].

Fabbri et al. [7] have applied specification mutation to validate specifications based on Finite State Machines (FSM). They have proposed 9 mutation operators, representing faults related to the states (e.g., wrong-starting-state, state-extra, etc.), transitions (e.g., event-missing, event-exchanged, etc.) and outputs (e.g., output-missing, output-exchanged, etc.) of an FSM. In a related work, Fabbri et al. [10] have defined mutation operators for Statecharts, an extension of FSM formalism, while Batth et al. [13] have applied mutation testing to Extended Finite State Machines (EFSM) formalism.

Hierons and Merayo [9] have investigated the application of mutation testing to Probabilistic (PFSMs) or stochastic time (PSFSMs) Finite State Machines. The authors [9] have defined new mutation operators representing FSM faults related to altering probabilities (PFSMs) or changing its associated random variables (PSFSMs) (i.e., the time consumed between the input being applied and the output being received).

The widespread interest in model-based testing techniques provides the major motivation of this research. We, in particular, focus on investigating the applicability of fault-based testing (vs. scenario-based testing) to Abstract State Machines (ASM) [14] specifications. We aim at assessing and further enhancing the fault-finding effectiveness of test suites targeting ASM-based models.

While mutation operators are well defined for a number of FSM related paradigms such as EFSM, PFSM and Statecharts, to the best of our knowledge mutation operators have not been defined for the Abstract State Machines [14] paradigm.

This paper serves the following purposes:

- Provide a set of mutation operators for Abstract State Machines [14] formalism.
- Present a classification of the proposed mutation operators into three categories: ASM domain operators, ASM function update operators, and ASM transition rules operators.
- Present a tool for generating and validating ASM mutants.

The remainder of this paper is organized as follows. The next section provides an overview of the Abstract State Machines (ASM) [14] formalism. In Section III, we define and classify a collection of mutation operators for ASM paradigm. Section IV describes the ASM Mutation tool. Finally, conclusions are drawn in Section V.

II. Abstract State Machines

Abstract State Machines (ASM) [14] define a state-based computational model, where computations (runs) are finite or infinite sequences of states $\{S_i\}$ obtained from a given initial state S_0 by repeatedly executing transitions δ_i:

 ISBN: 978-1-61208-230-1

$$S_0 \xrightarrow{\delta_1} S_1 \xrightarrow{\delta_2} S_2 \quad \cdots \quad \xrightarrow{\delta_n} S_n$$

An ASM $\mathcal{A}$ is defined over a fixed vocabulary $\mathcal{V}$, a finite collection of function names and relation names. Each function name f has an arity (number of arguments that the function takes). Function names can be static (i.e., fixed interpretation in each computation state of $\mathcal{A}$) or dynamic (i.e., can be altered by transitions fired in a computation step). Dynamic functions can be further classified into:

- Input functions that $\mathcal{A}$ can only read, which means that these functions are determined entirely by the environment of $\mathcal{A}$. They are also called monitored.
- Controlled functions of $\mathcal{A}$ are those which are updated by some of the rules of $\mathcal{A}$ and are never changed by the environment.
- Output functions of $\mathcal{A}$ are functions which $\mathcal{A}$ can only update but not read, whereas the environment can read them (without updating them).
- Shared functions are functions which can be read and updated by both $\mathcal{A}$ and the environment.

Static nullary (i.e., 0-ary) function names are called constants while Dynamic nullary functions are called variables. ASM *n-ary* functions have the following form: f: T_1 x T_2 x …... $T_n \rightarrow$ T.

Given a vocabulary $\mathcal{V}$, an ASM $\mathcal{A}$ is defined by its program $\mathcal{P}$ and a set of distinguished initial states S_0. The program $\mathcal{P}$ consists of transition rules and specifies possible state transitions of $\mathcal{A}$ in terms of finite sets of local function *updates* on a given global state. Such transitions are atomic actions. A transition rule that describes the modification of the functions from one state to the next has the following form:

if *Condition* **then** <*Updates*> **endif**

where *Updates* is a set of function updates (containing only variable free terms) of form: $f(t_1,t_2,\ldots,t_n) := t$ which are simultaneously executed when *Condition* (called also *guard*) is true. In a given state, first, all parameters t_i, t are evaluated to their values, v_i, *v*, then the value of $f(v_1,\ldots,v_n)$ is updated to *v*. Such pairs of a function name *f*, which is fixed by the signature, and an optional argument ($v_1,\ldots,v_n$), which is formed by a list of dynamic parameters value v_i, are called *locations*.

Example1: The following rule yields the update-set {(x, 2), (y(0), 1)}, if the current state of the ASM is {(x, 1), (y(0), 2)}:

$$\textbf{if} \quad (x = 1) \quad \textbf{then} \quad \begin{array}{l} x := y(0) \\ y(0) := x \end{array}$$

In every state, all the rules which are applicable are simultaneously applied. A set of ASM updates is called *consistent* if it contains no pair of updates with the same locations, i.e., no two elements *(loc,v)* and *(loc,v')* with *v*≠*v*'. In the case of inconsistency, the computation does not yield a next state.

Example2: The following update set {(x, 1), (y, 3), (x, 2)}, is inconsistent due to the conflicting updates for x:

$$\begin{array}{l} x := 1 \\ y := 3 \\ x := 2 \end{array}$$

For a detailed description of Abstract State Machines, the reader is invited to consult [15].

In what follows, we describe mutation operators for Abstract State Machines. Although, we illustrate the applicability of our approach using features and examples from *CoreASM* [16], our proposed mutation operators can be applied to any ASM-based language, thus maintaining the discussion generic.

III. Abstract State Machines Mutation Operators

We use the following guiding principles, introduced in [17], to formulate our mutation operators:

- Mutation categories should model potential faults.
- Only simple, first order mutants should be generated.
- Only syntactically correct mutants should be generated.

There exist several aspects of an ASM specification that can be subject to faults. These aspects can be classified into three categories of mutation operators:

1) ASM domain mutation operators.
2) ASM function update mutation operators.
3) ASM transition rules mutation operators.

Each category contains many mutation operators, one per a fault class.

A. ASM Domain Mutation Operators

A domain (called also *universe*) consists of a set of declarations that establish the ASM vocabulary. Each declaration establishes the meaning of an identifier within its scope. For example, the following *CoreASM* [16] code defines a new enumeration background *PRODUCT* having three elements (i.e., Soda, Candy, and Chips) and three functions *selectedProduct*, *price*, and *packaging*:

enum PRODUCT = {Soda, Candy, Chips}
function selectedProduct: → PRODUCT
function price: PRODUCT → NUMBER
function packaging: PRODUCT*PRODUCT → NUMBER

ASM domains/universes can be mutated by adding or removing elements. Table I shows examples of the following domain mutation operators:

- Extend Domain Operator (EDO): the domain is extended with a new element.

- Reduce Domain Operator (RDO): the domain is reduced by removing one element.
- Empty Domain Operator (EYDO): the domain is emptied.

Table I
EXAMPLES OF ASM DOMAIN MUTATION OPERATORS FOR *CoreASM* [16]

Mutation Operator	CoreASM Mutant S'
Extend Domain Operator (EDO)	enum PRODUCT = {Soda, Candy, Chips, Sandwich}.
Reduce Domain Operator (RDO)	enum PRODUCT = {Soda, Candy}.
Empty Domain Operator (EYDO)	enum PRODUCT = {}.

B. ASM Function Update Mutation Operators

A function update has the following form:

$$f(t_1,\ t_2,\ \ldots,\ t_n) := \text{value}$$

Depending on the type of operands, the traditional operators [4] such as Absolute Value Insertion (ABS), Arithmetic Operator Replacement (AOR), Logical Operator Replacement (LOR), Statement Deletion (SDL), Scalar Variable Replacement (SVR) etc., can be applied. In addition to these traditional mutation operators, we define:

- *Function Parameter Replacement (FPR)*: parameters of a function are replaced by other parameters of a compatible type.
- *Function Parameter Permutation (FPP)*: parameters of a function are exchanged.

Table II
EXAMPLES OF FUNCTION UPDATE MUTATION OPERATORS FOR *CoreASM* [16]

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
AOR	x := a + b	x := a - b
ABS	x := a + b	x := a + abs(b)
LOR	y := m and n	y := m or n
SDL	x := a + b	skip
SVR	selectedProduct:= Soda	selectedProduct:= Candy
FPR	price(Soda):=70	price(Candy):=70
FPP	packaging(Soda, Candy):= 1	packaging(Candy, Soda):= 1

Table II describes the proposed function update mutation operators.

C. ASM Transition Rules Mutation Operators

The transition relation is specified by guarded function updates, called *rules*, describing the modification of the functions from one state to the next. An ASM state transition is performed by firing a set of rules in one step.

1) Conditional Rule Mutation Operators:: The general schema of an ASM transition system appears as a set of guarded rules:

$$\textbf{if } \textit{Cond} \textbf{ then } Rule_{then} \textbf{ else } Rule_{else} \textbf{ endif}$$

where *Cond*, the guard, is a term representing a boolean condition. $Rule_{then}$ and $Rule_{else}$ are transition rules.

Many types of faults may occur on the guards of conditional rules [18]. Some of these faults include Literal Negation fault (LNF), Expression Negation fault (ENF), Missing Literal fault (MLF), Associative Shift fault (ASF), Operator Reference fault (ORF), Relational Operator fault (ROF), Stuck at 0(true)/1(false) fault (STF). Table III illustrates the mutation operators addressing the above fault classes. Furthermore, we define three additional conditional rule mutation operators:

- *Then Rule Replacement Operator (TRRO)*: replaces the rule $Rule_{then}$ by another rule.
- *Else Rule Replacement Operator (ERRO)*: replaces the rule $Rule_{else}$ by another rule.
- *Then Else Rule Permutation Operator (TERPEO)*: permutes the $Rule_{then}$ and the $Rule_{else}$ rules.

Table III
EXAMPLES OF CONDITIONAL RULE MUTATION OPERATORS FOR *CoreASM* [16]

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
LNO	**if** (a **and** b)	**if** (**not** a **and** b)
ENO	**if** (a **and** b)	**if not** (a **and** b)
MLO	**if** (a **and** b)	**if** (b)
ASO	**if** (a **and** (b **or** a))	**if** ((a **and** b) **or** a)
ORO	**if** (a **and** b)	**if** (a **or** b)
ROO	**if** (x >= c)	**if** (x <= c)
STO	**if** (a **and** b)	**if** (*true*)
TRRO	**if** a **then** *R1* **else** *R2*	**if** a **then** *R3* **else** *R2*
ERRO	**if** a **then** *R1* **else** *R2*	**if** a **then** *R1* **else** *R3*
TERPEO	**if** a **then** *R1* **else** *R2*	**if** a **then** *R2* **else** *R1*

2) Parallel and Sequence Rule Mutation Operators::
Parallel Constructor: If a set of ASM transition rules have to be executed simultaneously, a parallel rule is used:

$$\textbf{par } Rule_1 \ \ldots \ Rule_n \textbf{ endpar}$$

The update generated by this rule is the union of all the updates generated by $Rule_1$ to $Rule_n$.

Sequence Constructor: The sequence rule aims at executing rules/function updates in sequence:

$$\textbf{seq } Rule_1,\ \ldots,\ Rule_n$$

The resulting update set is a sequential composition of the updates generated by $Rule_1 \ \ldots \ Rule_n$.

We define the following mutation operators for both *Parallel* and *Sequence* constructors:

- *Add Rule Operator (ARO)*: adds a new rule to the parallel/sequence of rules.

- *Delete Rule Operator (DRO)*: deletes a rule from the parallel/sequence of rules.
- *Replace Rule Operator (RRO)*: replaces one of the rules in the parallel/sequence by another rule.
- *Permute Rule Operator (PRO)*: changes the order of the parallel/sequence rules by permuting two rules.

Table IV
EXAMPLES OF THE PARALLEL/SEQUENCE RULE MUTATION OPERATORS FOR *CoreASM* [16]

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
ARO	**seqblock** *R1 R2* **endseqblock**	**seqblock** *R1 R2 R3* **endseqblock**
ARO	**par** *R1 R2* **endpar**	**par** *R1 R2 R3* **endpar**
DRO	**seqblock** *R1 R2 R3* **endseqblock**	**seqblock** *R1 R3* **endseqblock**
DRO	**par** *R1 R2 R3* **endpar**	**par** *R1 R3* **endpar**
RRO	**seqblock** *R1 R2* **endseqblock**	**seqblock** *R1 R3* **endseqblock**
RRO	**par** *R1 R2* **endpar**	**par** *R1 R3* **endpar**
PRO	**seqblock** *R1 R2* **endseqblock**	**seqblock** *R2 R1* **endseqblock**
PRO	**par** *R1 R2* **endpar**	**par** *R2 R1* **endpar**
SPEO	**seqblock** *R1 R2* **endseqblock**	**par** *R1 R2* **endpar**
SPEO	**par** *R1 R2* **endpar**	**seqblock** *R1 R2* **endseqblock**

In addition to these rules, we define the *Sequence-Parallel Exchange Operator (SPEO)* to exchange a sequence rule with a parallel rule and vice versa. Table IV illustrates the Parallel/Sequence rule mutation operators.

It is worth noting that:

- Applying *SPEO* operator may result into mutants that are syntactically correct but containing inconsistent updates. Table V shows a simple *coreASM* sequence rule and its corresponding mutant after applying *SPEO* operator. The execution of the produced mutant leads to an inconsistent update of variable *a* (i.e., the computation of the rule does not yield a next state).

Table V
APPLYING SPEO OPERATOR THAT LEADS TO AN INCONSISTENT UPDATE

Original Spec S	Mutant Spec S'
rule Main = seqblock a := a + 1 a := b endseqblock	rule Main = **par** a := a + 1 a := b **endpar**

- Applying *SPEO* operator may produce a mutant that is equivalent to the original specification. Indeed, such a case may take place when the rules enclosed within the parallel/sequence blocks do not interfere. Table VI shows a specifications S and its mutant S'. Both specifications are equivalents from an input/output perspective since variables *a* and *b* are updated independently. However, the original specification S produces 2 states (i.e., one *a:= a + 1* and one for *b := b +1*) whereas its mutant S' produces only one single state (i.e., *a:= a +1* and *b := b + 1* are executed in one single step).

Table VI
APPLYING SPEO OPERATOR PRODUCES A MUTANT THAT IS EQUIVALENT TO THE ORIGINAL SPEC

Original Spec S	Mutant Spec S'
rule Main = **seqblock** a := a + 1 b := b + 1 **endseqblock**	rule Main = **par** a := a + 1 b := b + 1 **endpar**

3) Choose Rule Mutation Operators:: The choose rule consists on selecting elements (non deterministically) from specified domains which satisfy guards φ, then evaluates $Rule_1$. If no such elements exist, then evaluates $Rule_2$.

choose x_1 **in** $D_1, \ldots, x_n$ **in** D_n **with** $\varphi\ (x_1, \ldots, x_n)$ **do** $Rule_{do}$ **ifnone** $Rule_{ifnone}$

The **with** and **ifnone** blocks are optional. The guard φ may be a simple boolean expression of predicate logic expressions.

To cover the *choose* rule, we define the following mutation operators:

- *Choose Domain Replacement Operator (CDRO)*: replaces a variable domain with another compatible domain.
- *Choose Guard Modification Operator (CGMO)*: alters the guard φ. In this paper, we consider simple boolean expressions as guards. Predicate logic expressions such as *exists* are left for future work.
- *Choose DoRule Replacement Operator (CDoRO)*: replaces the rule $Rule_{do}$ by another rule.
- *Choose IfNoneRule Replacement Operator (CIRO)*: replaces the rule $Rule_{ifnone}$ by another rule.
- *Choose Rule Exchange Operator (CREO)*: replaces the rule $Rule_{ifnone}$ by another rule.

Table VII
EXAMPLE OF THE CHOOSE RULE MUTATION OPERATORS FOR *CoreASM* [16]

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
CDRO	**choose** x **in** Set1 **with** (x >= 0)	**choose** x **in** Set2 **with** (x >= 0)
CGMO	**choose** x **in** Set1 **with** (x >= 0)	**choose** x **in** Set1 **with** (x <= 0)
CDoRO	**choose** x **in** Set1 **do** Rule1	**choose** x **in** Set1 **do** Rule2
CIRO	**choose** x **in** Set1 **do** Rule1 **ifnone** Rule2	**choose** x **in** Set1 **do** Rule1 **ifnone** Rule3
CREO	**choose** x **in** Set1 **do** Rule1 **ifnone** Rule2	**choose** x **in** Set1 **do** Rule2 **ifnone** Rule1

 ISBN: 978-1-61208-230-1

4) Forall Rule Mutation Operators:: The synchronous parallelism is expressed by a *forall* rule which has the following form:

$$\textbf{forall } x_1 \text{ in } D_1, \ldots, x_n \text{ in } D_n \textbf{ with } \varphi \textbf{ do } Rule_{do}$$

where $x_1, \ldots, x_n$ are variables, $D_1, \ldots, D_n$ are the domains where x_i take their value, φ is a boolean condition, $Rule_{do}$ is a transition rule containing occurrences of the variables x_i bound by the quantifier.

We define the following mutation operators for the *forall* rule that are quite similar to the ones of the *choose* rule :

- *Forall Domain Replacement Operator (FDRO)*: replaces a variable domain with another compatible domain.
- *Forall Guard Modification Operator (FGMO)*: alters the guard φ using the set of operators introduced in Table III.
- *Forall DoRule Replacement Operator (FDoRO)*: replaces the rule $Rule_{do}$ by any other rule.

Table VIII
EXAMPLES OF THE FORALL RULE MUTATION OPERATORS FOR *CoreASM* [16]

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
FDRO	**forall** x **in** Set1 **with** (x = 0) **do** R1	**forall** x **in** Set2 **with** (x >= 0) **do** R1
FGMO	**forall** x **in** Set1 **with** (x = 0) **do** R1	**forall** x **in** Set1 **with** (x <= 0) **do** R1
FDoRO	**forall** x **in** Set1 **do** R1	**forall** x **in** Set1 **do** R2

In addition to the proposed *forall* rule mutation operators illustrated in Table VIII, we define the *Choose-Forall Exchange Operator (CFEO)* to exchange a *choose* rule with a *forall* rule and vice versa (See Table IX).

Table IX
EXAMPLES OF THE CHOOSE-FORALL EXCHANGE OPERATOR FOR *CoreASM* [16]

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
CFEO	**forall** x **in** Set1 **do** R1	**choose** x **in** Set1 **do** R1
CFEO	**choose** x **in** Set1 **do** R1	**forall** x **in** Set1 **do** R1

5) Let Rule:: The *let* rule assigns a value of a term *t* to the variable *x* and then execute the rule *Rule* which contains occurrences of the variable *x*. The syntax of a Let rule is:

$$\textbf{let } (x = t) \textbf{ in } Rule \textbf{ endlet}$$

We define the following mutation operators (see Table X):

- *Let Variable Assignment Operator (LVAO)*: assigns a different value to *x*, other than *t*, of a compatible type.
- *Let Rule Replacement Operator (LRRO)*: replaces the rule *Rule* by another rule that has occurrences of *x*.
- *Let Rule Variable Replacement (LRVR)*: replaces the variable *x* by another variable.

Table X
EXAMPLES OF THE LET RULE OPERATORS FOR *CoreASM* [16]

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
LVAO	**let** x = 1 **in** R1	**let** x = 2 **in** R1
LRRO	**let** x = 1 **in** R1	**let** x = 1 **in** R2
LRVR	**let** x = 1 **in** R1	**let** y = 1 **in** R1

Other ASM rules such as Case rule, iterate rule, etc. are not covered in this work due to the lack of space.

IV. ASM MUTANTS GENERATION

Figure 1 illustrates the *ASM Mutation Tool* user interface. The user may select one or multiple operators from the three operator categories. The produced mutants are then stored in separate files and run using *carma*, a comprehensive command-line to run *CoreASM* specification, to check their validity. Figure 2 shows an example of the output produced from the execution of *carma*, from the command line, on a syntactically incorrect specification (i.e., the output shows 'Engine Error' and the error location). Note that only 1 execution step is needed to detect syntax errors (i.e., carma –steps 1 MySpec.casm).

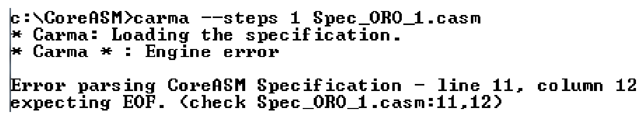

Figure 2. Checking the Validity of the Generated Mutant

It is worth noting that the mutation operator EDO (Extend Domain Operator) requires manual definition of the added element.

V. CONCLUSION AND FUTURE WORK

In this paper, we have introduced mutation operators for Abstract State Machines (ASM) formalism. The proposed set of mutation operators are classified into three main categories: ASM domain operators, ASM function update operators, and ASM transition rules operators. Mutants are generated automatically and their syntax are checked for correctness. As a future work, we are planning to conduct an empirical evaluation of the designed operators and to assess their effectiveness and the number of mutants they produce.

REFERENCES

[1] A. P. Mathur, *Mutation Testing*. John Wiley & Sons, Inc., 2002.

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649 –678, Sept.-Oct. 2011.

[3] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, pp. 97–133, June 2005.

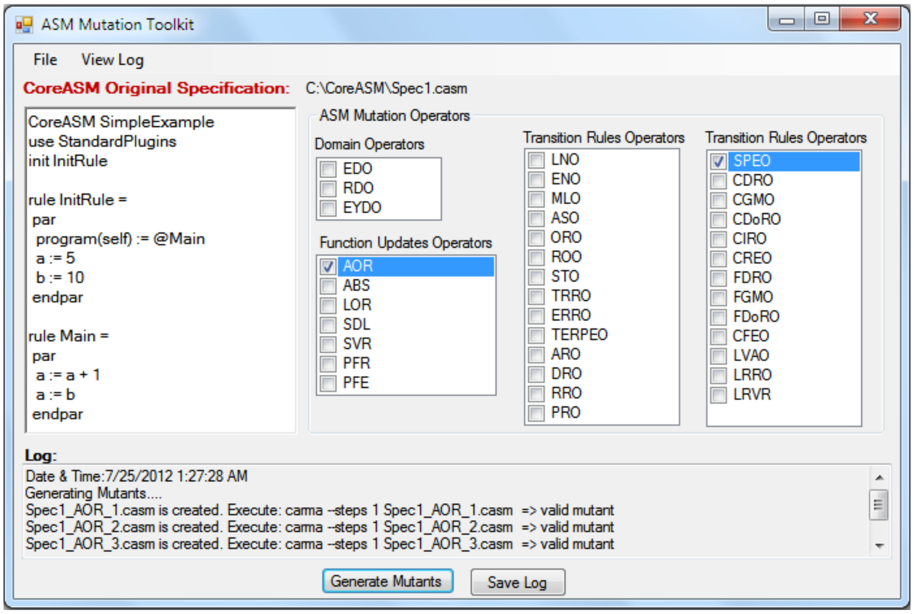

Figure 1. ASM Mutation Toolkit

[4] A. J. Offutt, VI and K. N. King, "A fortran 77 interpreter for mutation analysis," *SIGPLAN Not.*, vol. 22, pp. 177–188, July 1987.

[5] H. Agrawal, "Design of mutant operators for the C programming language," Software Engineering Research Center/Purdue University, Tech. Rep., 1989.

[6] P. E. Black, V. Okun, and Y. Yesha, "Mutation operators for specifications," in *Proceedings of the 15th IEEE international conference on Automated software engineering*, ser. ASE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 81–88.

[7] S. Pinto Ferraz Fabbri, M. Delamaro, J. Maldonado, and P. Masiero, "Mutation analysis testing for finite state machines," in *Proceedings of the 5th International Symposium on Software Reliability Engineering*, November 1994, pp. 220 –229.

[8] J.-h. Li, G.-x. Dai, and H.-h. Li, "Mutation analysis for testing finite state machines," in *Proceedings of the 2009 Second International Symposium on Electronic Commerce and Security - Volume 01*, ser. ISECS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 620–624.

[9] R. M. Hierons and M. G. Merayo, "Mutation testing from probabilistic and stochastic finite state machines," *J. Syst. Softw.*, vol. 82, pp. 1804–1818, November 2009.

[10] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero, "Mutation testing applied to validate specifications based on statecharts," in *Proceedings of the 10th International Symposium on Software Reliability Engineering*, ser. ISSRE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 210–219.

[11] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. Wong, "Mutation testing applied to validate specifications based on petri nets," in *Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII*. London, UK, UK: Chapman & Hall, Ltd., 1996, pp. 329–337.

[12] S. D. R. S. De Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. L. De Souza, "Mutation testing applied to estelle specifications," *Software Quality Control*, vol. 8, pp. 285–301, December 1999.

[13] S. S. Batth, E. R. Vieira, A. Cavalli, and M. U. Uyar, "Specification of timed efsm fault models in sdl," in *Proceedings of the 27th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, ser. FORTE '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 50–65.

[14] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide," in *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1995, pp. 9–36.

[15] E. Börger and R. F. Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003.

[16] R. Farahbod, V. Gervasi, and U. Glässer, "CoreASM: An Extensible ASM Execution Engine," *Fundamenta Informaticae*, vol. 77, pp. 71–103, January 2007.

[17] M. Woodward, "Errors in algebraic specifications and an experimental mutation testing tool," *Software Engineering Journal*, vol. 8, no. 4, pp. 211–224, Jul 1993.

[18] M. F. Lau and Y. T. Yu, "An extended fault class hierarchy for specification-based testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 247–276, July 2005.