Design and Classification of Mutation Operators for Abstract State Machines

Jameleddine Hassine
Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, KSA
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 is the most critical activity towards conducting effective mutation testing and analysis. Mutation operators are well defined for a number of programming (e.g., C, Java, etc.) and specification (e.g., FSM, Petri Nets, etc.) languages. In this paper, we design and classify mutation operators for the Abstract State Machines (ASM) formalism. The designed operators are defined based on the types of faults that may occur in ASM specifications and can be classified into three categories: (1) Domain operators, (2) function update operators, and (3) transition rules operators. Furthermore, a prototype mutation tool for the *CoreASM* language, has been built to automatically generate mutants and check their validity. We illustrate our approach using a simple *CoreASM* implementation of the *Fibonacci* series. Finally, an empirical comparison of the designed operators is presented and discussed.**

Keywords*—*Mutation testing; specification; mutation operator; Abstract State Machines (ASM); domain operators; function update operators; transition rules operators; CoreASM.

I. INTRODUCTION

In this article, we describe an extension of our work on designing Abstract State Machines mutation operators published in [1].

Fault based testing strategies aim at finding prescribed faults in a program [2]. Mutation testing [3] 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 modifications, or mutations, into the software artifact (i.e., source code or specification) under test. Mutation operators are classified by the language constructs they are created to alter. Given the fact that a program/specification being mutated is syntactically correct, a mutation operator must produce a mutant that is also syntactically correct. The objective is then to select test cases that are capable to distinguish the behavior of the mutants from the behavior of the original artifact. Such test cases are said to *kill* the mutants. However, it may also be that the mutant keeps the program's semantics unchanged-and thus cannot be detected by any test case. Such mutants are called *equivalent* mutants. The detection of equivalent mutants is, in general, one of biggest obstacles for practical usage of mutation testing. The effort needed to check if mutants are equivalent or not, can be very high even for small programs [4].

Since the number of possible faults for a given program or specification can be large, mutation-based testing strategies are based on the following two principles: (1) the Competent Programmer Hypothesis [3], which states that competent programmers tend to write programs that are *close* to being correct. In other words, a program written by a competent programmer may be incorrect but it is generally likely close to being correct (containing relatively simple faults) (2) the Coupling Effect [3], which states that a test data set that catches all simple faults in a program is so sensitive that it will also catch more complex faults. Analogously to the Competent Programmer Hypothesis [3], Ammann and Black [5] have proposed the *Competent Specifier Hypothesis* stating that analysts write specifications which are likely to be close to what is desired.

In a recent survey on the development of mutation testing, Jia and Harman [4] have stated that more than 50% of the mutation related publications have been applied to Java [6], [7], Fortran [8], [9] and C [10]. Although mutation testing has mostly been applied at the source code level, it has also been applied at the specification and design level [11], [4]. Formal specification languages to which mutation testing has been applied include Finite State Machines [12], [13], [14], Statecharts [15], Petri Nets [16], and Estelle [17].

Fabbri et al. [12] 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. [15] have defined mutation operators for Statecharts, an extension of FSM formalism, while Batth et al. [18] have applied mutation testing to Extended Finite State Machines (EFSM) formalism. Hierons and Merayo [14] have investigated the application of mutation testing to Probabilistic (PFSMs) or stochastic time (PSFSMs) Finite State Machines. The authors 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

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

testing (vs. scenario-based testing) to Abstract State Machines (ASM) [19] specifications. In this paper, we extend our previous work [1] on designing ASM-based mutation operators by:

- Extending the set of operators, introduced in [1], by adding the *Call Rule Operators*, the *Pick Rule Operators*, and the *Extend Rule Operators*.
- Refining the classification of the proposed ASM-based mutation operators. The resulting ASM-based operators can be classified using three categories: (1) ASM domain operators, (2) ASM function update operators, and (3) ASM transition rules operators.
- Presenting *CoreASM* [20], an ASM-based language, illustrative examples of the proposed mutation operators.
- Presenting an enhanced version of our *CoreASM* [20] mutation prototype tool, for automatic generation, validation, and execution of *CoreASM* mutants.
- Analyzing the generated mutants using an illustrative example of a *CoreASM* specification of *Fibonacci series*.
- Presenting an empirical comparison of the proposed *CoreASM* mutation operators using three *CoreASM* specifications: *Dining Philosophers*, *Vending Machine*, and *Rail Road Crossing*.

The remainder of this paper is organized as follows. The next section provides an overview of the Abstract State Machines (ASM) [19] formalism and the *CoreASM* language. In Section III, we define and classify a collection of mutation operators for *CoreASM* language. An analysis of the generated mutants is presented in Section IV. Section V describes the *CoreASM* Mutation toolkit. To demonstrate the applicability of the proposed approach, Section VI describes the application of *CoreASM* mutation operators to Fibonacci specification. An empirical comparison of *CoreASM*-based mutation operators is presented in Section VII. Finally, conclusions are drawn in Section VIII.

II. Abstract State Machines

Abstract State Machines (ASMs), originally known as *Evolving Algebras*, were first introduced by Yuri Gurevich [21], [19] in an attempt to improve on Turing's thesis [22] so that:

> "*Every algorithm is an ASM as far as the behavior is concerned. In particular the given algorithm can be step-for-step simulated by an appropriate ASM [23]." (The ASM Thesis)*

This means that an activity that is conceptually done in one step can be executed in the model in one step. This is in contrast to Turing machines, where simple operations might need any finite number of steps.

Abstract State Machines have been used to capture sequential, parallel and distributed algorithms. ASMs combine two fundamental concepts of transition systems: (1) transitions to model the dynamic aspects of a system, and (2) abstract states to model the static aspects at any desired level of abstraction. Börger and Stärk [24] further developed ASMs into a system engineering method that guides the development of software from requirements capture to implementation.

Widely recognized applications of ASMs include semantic foundations of a wide variety of programming languages like C++ [25], C# [26], and Java [27], logic programming languages such as Prolog [28] and its variants, hardware languages such as VHDL [29], system design languages like the ITU-T standard for SDL [30], [31], Web service description languages [32], design of distributed systems [33], [34], etc.

A. ASM Program

Abstract State Machines (ASM) [19] 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:

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

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.

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 the form:

$$f(t_1,t_2,\ldots,t_n) := t$$

where t_1, t_2, ..., t_n, and t are first order terms.

The set of function updates 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*.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Example 1: 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 location updated with two different values, i.e., no two elements *(loc,v)* and *(loc,v')* with $v \neq v'$. In the case of inconsistency, the computation does not yield a next state.

Example 2: The following update set {(x, 1), (y, 2), (x, 2), (y, 2)}, is inconsistent due to the conflicting updates for x (i.e., x is updated with different values 1 and 2). It is worth noting that even though y is updated twice, it does not lead to an inconsistent update since it has been updated with the same value 2.

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

For a detailed description and a rigorous mathematical definition of the semantic foundations of Abstract State Machines, the reader is invited to consult [19], [24], [35], [36].

B. CoreASM Language

The *CoreASM* project [37] focuses on the design of a lean executable ASM language, in combination with a supporting tool environment for high-level design, experimental validation and, where appropriate, formal verification of abstract system models [20]. The *CoreASM* engine, implemented in *Java*, consists of a parser, an interpreter, a scheduler, and an abstract storage. The interpreter, the scheduler, and the abstract storage work together to simulate an ASM run. For a detailed description of *CoreASM* architecture, reader is invited to consult [20].

CoreASM is designed with extensibility in mind, supporting the extension of both the specification language and the execution engine's behavior through plug-ins (e.g., *Standard*, *KernelExtensions*, *Abstraction*, *TurboASM*, etc.).

Figure 1 shows a typical structure of a *CoreASM* specification. Every specification starts with the keyword **CoreASM** followed by the name of the specification. Plugins that are required are then listed with the keyword **use** followed by the name of the plugin (e.g., **use** Standard). The Header block is where various definitions take place (e.g., Declaration of an enumeration type). The **init** rule (the rule that creates the initial state) is defined by the keyword **init** followed by a rule name. This would be the rule that initializes the state of the ASM machine. The body of the init rule must be declared in the rule declaration block along with other user defined rules.

To run a *CoreASM* specification, two user interfaces are available:

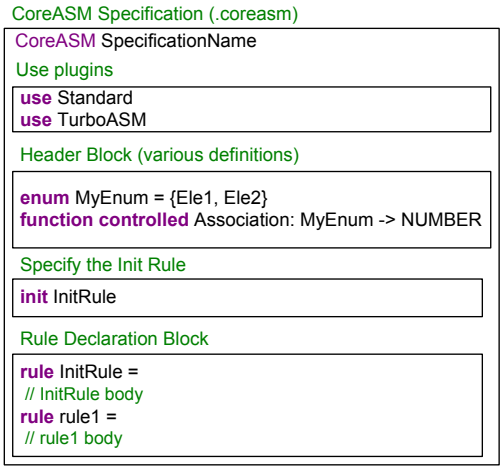


Fig. 1. Typical Structure of a CoreASM Specification

- A comprehensive command-line user interface called *Carma*, which accepts the name of the specification file and optional termination conditions (e.g., --steps 10 and/or --empty-updates) as arguments. For example, the following command runs *Spec.coreasm* using *Carma* and stops after 10 steps, or after a step that generates empty updates.

  carma --steps 10 --empty-updates Spec.coreasm

- A graphical interactive development environment in the *Eclipse* platform, known as the *CoreASM Eclipse Plugin*.

In what follows, we define and classify mutation operators for Abstract State Machines.

III. Abstract State Machines Mutation Operators

In order to formulate mutation operators for ASM formalism, we use the following guiding principles, introduced in [38]:

- Mutation categories should model potential faults.
- Only simple, first order mutants (i.e., a single change to an artifact) should be generated.
- Only syntactically correct mutants should be generated.

A. Categories of ASM Mutation Operators

There exist several aspects of an ASM specification that can be subject to faults. These aspects can be classified into three main categories of mutation operators, each category contains many mutation operators, one per a fault class:

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

Although the proposed categorization yields few generic categories that can be applied to any ASM-based language, the operators themselves are dependent on the syntax of the ASM-based language. Indeed, given that a specification being mutated is syntactically correct, a mutation operator must produce a mutant that is also syntactically correct. To do so,

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

it is required that a valid syntactic construct be mapped to another syntactic construct in the same language. In addition, peculiarities of language syntax have an effect on the kind of mistakes that a modeler could make. For instance, aspects such as procedural (e.g., *CoreASM* [20] language) versus object oriented (e.g., *AsmL* [39] language) are captured in the language syntax. In this paper, we target the *CoreASM* [20] language.

B. ASM Domain Mutation Operators

A domain (called also *universe* or *background*) 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* [20] code defines a new enumeration background *PRODUCT* having three elements (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:

- 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.

These mutation operators can be applied to enumeration (See Table I), universes, collections, the set background, the list background, and the map background.

TABLE I. EXAMPLES OF ASM DOMAIN MUTATION OPERATORS FOR *CoreASM*

Domain 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 = {}

C. 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 [8], [40] such as Absolute Value Insertion (ABS), Arithmetic Operator Replacement (AOR), Logical Operator Replacement (LOR), Statement Deletion (SDL), Scalar Variable Replacement (SVR), and Unary Operator Insertion (UOI) can be applied. In addition to these traditional mutation operators, we define *Function Parameter Replacement (FPR)* operator, where parameters of a function are replaced by other parameters of a compatible type. Two Types are compatible if values of one type can appear wherever values of the other type are expected, and vice versa.

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

Table II illustrates some examples of the proposed function update mutation operators.

TABLE II. EXAMPLES OF FUNCTION UPDATE MUTATION OPERATORS FOR *CoreASM*

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
ABS	x := a + b	x := a + abs(b)
AOR	x := a + b	x := a - b
LOR	y := m and n	y := m or n
SDL	x := a + b	skip
SVR	x := a * b	a := a * b
UOI	x := 3 * a	x := 3 * -a
FPR	price(Soda):=70	price(Candy):=70
FPP	packaging(Soda, Candy):=1	packaging(Candy, Soda):=1

D. 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:

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

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 [41]. 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 existing rule.
- *Else Rule Replacement Operator (ERRO)*: replaces the rule $Rule_{else}$ by another existing rule.
- *Then Else Rule Permutation Operator (TERPEO)*: permutes the $Rule_{then}$ and the $Rule_{else}$ rules. It is worth noting that operators *TERPEO* and *ENO* would produce syntactically different but semantically equivalent mutants.

2) Sequence Rule Mutation Operators: The sequence rule aims at executing rules/function updates in sequence. *TurboASM* plugin offers two forms of sequential rules:

$$\textbf{seq } Rule_1 \textbf{ next } Rule_2 \quad (1)$$

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE III. EXAMPLES OF CONDITIONAL RULE MUTATION OPERATORS FOR *CoreASM*

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
LNO (Literal Negation)	**if** (a **and** b)	**if** (**not** a **and** b)
ENO (Expression Negation)	**if** (a **and** b)	**if not** (a **and** b)
MLO (Missing Literal)	**if** (a **and** b)	**if** (b)
ASO (Associative Shift)	**if** (a **and** (b **or** a))	**if** ((a **and** b) **or** a)
ORO (Operator Reference)	**if** (a **and** b)	**if** (a **or** b)
ROO (Relational Operator)	**if** (x >= c)	**if** (x <= c)
STO (Stuck at 0/1)	**if** (a **and** b)	**if** (*true*)
TRRO (Then Rule Replacement)	**if** a **then** *R1* **else** *R2*	**if** a **then** *R3* **else** *R2*
ERRO (Else Rule Replacement)	**if** a **then** *R1* **else** *R2*	**if** a **then** *R1* **else** *R3*
TERPEO (Then Else Rule Permutation)	**if** a **then** *R1* **else** *R2*	**if** a **then** *R2* **else** *R1*

Evaluates $Rule_1$, applies the generated updates in a virtual state, and evaluates $Rule_2$ in that state. The resulting update set is a sequential composition of the updates generated by $Rule_1$ and $Rule_2$.

$$\textbf{seqblock}\ Rule_1\ \ldots Rule_n\ \textbf{endseqblock} \quad (2)$$

Similar to the **seq** rule (above), this rule form executes the listed rules in sequence. 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 the sequence rule:

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

Table IV illustrates examples of the sequence rule mutation operators.

TABLE IV. EXAMPLES OF THE SEQUENCE RULE MUTATION OPERATORS FOR *CoreASM*

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

3) Block Rule Mutation Operators: If a set of ASM transition rules have to be executed simultaneously, a block rule (included in the BlockRule plugin) 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 $Rule_1 \ldots Rule_n$. The sequence rule operators (i.e., ARO, DRO, and RRO defined in Section III-D2) can be applied to the block rule. Table V illustrates the sequence-block exchange mutation operator.

Applying PRO to a block rule, will produce an equivalent specification (i.e., **par** *R1 R2* **endpar** is equivalent to **par** *R2 R1* **endpar**). Section IV-B discusses equivalent mutants.

TABLE V. EXAMPLES OF THE BLOCK RULE MUTATION OPERATORS FOR *CoreASM*

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

4) Sequence-Block Exchange Operator: In addition to the sequence and block mutation operators, we define the *Sequence-Block Exchange Operator (SBEO)* to exchange a sequence rule with a block rule and vice versa. Table VI illustrates the sequence-block exchange mutation operator.

TABLE VI. EXAMPLES OF THE SEQUENCE-BLOCK RULE MUTATION OPERATOR

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
SBEO	**seqblock** *R1 R2* **endseqblock**	**par** *R1 R2* **endpar**
SBEO	**par** *R1 R2* **endpar**	**seqblock** *R1 R2* **endseqblock**

5) Choose Rule Mutation Operators: The choose rule consists on selecting elements (non deterministically) from specified domains that satisfy guards φ, then evaluates $Rule_{do}$. If no such elements exist, then evaluates $Rule_{ifnone}$.

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

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 using the operators described in Table III. 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 $Rule_{do}$ rule by $Rule_{ifnone}$ rule.

Table VII illustrates the choose rule mutation operators.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE VII. EXAMPLE OF THE CHOOSE RULE MUTATION OPERATORS FOR *CoreASM*

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** R1	**choose** x **in** Set1 **do** R2
CIRO	**choose** x **in** Set1 **do** R1 **ifnone** R2	**choose** x **in** Set1 **do** R1 **ifnone** R3
CREO	**choose** x **in** Set1 **do** R1 **ifnone** R2	**choose** x **in** Set1 **do** R2 **ifnone** R1

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

forall x_1 in D_1, ..., x_n in D_n **with** φ **do** $Rule_{do}$

where x_1, ..., x_n are variables, D_1, ..., D_n are the domains where x_i take their value, φ is a boolean condition, $Rule_{do}$ is a transition rule containing occurences 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*

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

7) Choose-Forall Exchange Operator: In addition to the proposed *forall* and *choose* rule mutation operators illustrated in Tables VIII and VII, 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*

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

8) Let Rule Mutation Operators: The *let* rule, included in the *LetRule* plugin, 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:

let (x = t) **in** $Rule$

We define the following *Let* rule 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 of same type.

TABLE X. EXAMPLES OF THE LET RULE OPERATORS FOR *CoreASM*

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

9) Call Rule Mutation Operators: The call rule executes the previously defined transition rule R with the given parameters. Parameters are passed in a call-by-name fashion; i.e., they are passed unevaluated. The syntax of a *Call* rule is:

R(a_1, ..., a_n)

We define the following *Call* rule mutation operators (see Table XI):

- *Call Rule Parameter Replacement (CRPR)*: replaces the actual rule parameter by another parameter of the same type.
- *Call Rule Parameter Exchange (CRPE)*: permutes actual parameters if they are of the same type.

TABLE XI. EXAMPLE OF CALL RULE MUTATION OPERATOR

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
CRPR	rule Add(a, b)= return a+b rule Main = addition := Add(x,y)	rule Add(a, b)= return a+b rule Main = addition := Add (x,**z**)
CRPE	rule Add(a, b)= return a+b rule Main = addition := Add(x,y)	rule Add(a, b)= return a+b rule Main = addition := Add (**y**,**x**)

10)Pick Rule Mutation Operators: The pick rule, part of the *ChooseRule* plugin, provides another way of pick non-deterministically a value that satisfies a given condition from an enumerable. Its syntax is as follows:

pick *x* **in** *D* **with** *guard*

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

- *Pick Domain Replacement Operator (PDRO)*: replaces the domain D with another compatible domain.
- *Pick Guard Modification Operator (PGMO)*: alters the guard φ using the operators described in Table III.

Table XII illustrates the pick rule mutation operators.

11)Extend Rule Mutation Operators: The extend rule, part of *ExtendRule* plugin, is used to construct new elements and add them to a specific domain. The resulting update set is the updates generated by $Rule$.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE XII. EXAMPLE OF THE PICK RULE MUTATION OPERATORS FOR *CoreASM*

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
PDRO	**pick** x **in** D1 **with** (x >= 0)	**pick** x **in** D2 **with** (x >= 0)
PGMO	**pick** x **in** D1 **with** (x >= 0)	**pick** x **in** D1 **with** (x <= 0)

extend *D* **with** *id* **do** *Rule*

We define the following *Extend* rule mutation operators (see Table XIII):

- *Extend Domain Replacement Operator (EDRO)*: replaces the domain by another compatible domain.
- *Extend Rule Replacement Operator (ERRO)*: replaces the rule $Rule$ by another one.
- *Extend Id Replacement Operator (EIRO)*: extends the domain with another element of a compatible type (e.g., extend the domain D1 with id2 instead of id1). All occurrences of the *id* are replaced in *Rule*.

TABLE XIII. EXAMPLES OF THE EXTEND RULE OPERATORS FOR *CoreASM*

Mutation Operator	CoreASM Spec S	CoreASM Mutant S'
EDRO	**extend** D1 **with** id1 **do** R1	**extend** D2 **with** id1 **do** R1
ERRO	**extend** D1 **with** id1 **do** R1	**extend** D1 **with** id1 **do** R2
EIRO	**extend** D1 **with** id1 **do** R1	**extend** D1 **with** id2 **do** R1

Other CoreASM-specific rules and constructs such as *Case* rule, add/remove *List* constructs, enqueue/dequeue *Queue* constructs, etc. are not covered in this paper.

E. CoreASM*: What is not Mutated*

Some mutation operators may have a possible infinite domain on which they operate. For instance, given the fact that the set of types might be infinite, it is difficult to determine how the declaration of a variable of one specific type may be mutated. This is applicable to libraries, functions names, etc. For the *CoreASM* language, the following entities are not mutated:

- Variable declarations.
- Format of strings in I/O functions.
- The *init* rule declaration (i.e., *init* InitRule)
- Plugin names introduced using the **use** keyword.
- Rule declarations
- Rule names indicating a call to a rule. Note that the actual parameters in a *Call* rule are mutated (e.g., *CRPR* and *CRPE* operators) but the rule names are not.

IV. ANALYSIS OF THE GENERATED MUTANTS

A. Inconsistent Updates

Applying *SBEO* operator may result into mutants that are syntactically correct but containing inconsistent updates. Therefore, the computation does not yield a next state. Table XIV shows a simple *CoreASM* sequence rule and its corresponding mutant after applying *SBEO* operator. The execution of the produced mutant may lead to an inconsistent update of variable a (i.e., in case variable a is updated twice simultaneously with different values (a+1 $\neq$ b)).

TABLE XIV. APPLYING SBEO OPERATOR THAT LEADS TO AN INCONSISTENT UPDATE

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

B. Equivalent Mutants

In many cases, applying *CoreASM* mutation operator produces a specification that is equivalent to the original specification. For instance, the application of the *PRO* (Permute Rule Operator) to a block rule (e.g., **par** R1 R2 **endpar**), would produce a mutant (e.g., **par** R2 R1 **endpar**) that is equivalent to the original specification.

Similarly, applying *SBEO* operator may produce a mutant that is equivalent to the original specification. This might be the case when the rules enclosed within the parallel/sequence blocks are independent (i.e., with different functions updates). Table XV shows a specifications *S* and its mutant *S*'. Rules "a:=a+1" and "b:=b+1" are independent (i.e., Variables a and b are updated independently). Hence, no test cases would kill mutant *S*'. 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 XV. APPLYING SBEO OPERATOR PRODUCES A MUTANT THAT IS EQUIVALENT TO THE ORIGINAL SPEC

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

In general, like traditional programming languages, detecting *CoreASM* equivalent mutants is an undecidable problem [40].

V. COREASM MUTATION TOOLKIT

Figures 2, 3, and 4 illustrate the Microsoft .NET C#-based, *CoreASM Mutation Toolkit* GUI. The GUI is composed of four tab pages: (1) Mutants Generator tab (Figure 2), (2) Mutants Viewer tab (Figure 3), (3) Test Execution tab (Figure 4), and (4) Help tab. The user starts with loading a *CoreASM* specification, then he/she selects one or multiple operators from the three operator categories. The produced mutants are created and stored in separate files in a separate directory.

In Section III, we have stated that only syntactically correct mutants are generated, as a result of applying the mutation

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

operators. This guiding principle is further enforced by checking the validity of the produced mutants using the *Carma* command line. The invalid mutants, if any, are then discarded. The error output for syntactically invalid mutants is stored in a log file.

The generated mutants can be viewed using the second tab page (see Figure 3). Statistics about the type and the number of produced valid mutants are listed in the log section.

The test execution GUI (Figure 4) allows for the execution of test cases against the generated mutants. The test case definition include a sequence of inputs that the specification requires the user to enter, a sequence of expected outputs (one per line), and a sequence of strings from which the output will be extracted (one per line).

VI. Illustrative Example: Fibonacci Series

In this section, we apply mutation testing to a *CoreASM* specification that produces *Fibonacci* numbers. The *Fibonacci* numbers or Fibonacci series are the numbers in the following integer sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

By definition, the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two. In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

Figure 5 describes the *CoreASM* recursive implementation for producing *Fibonacci* numbers. The user is asked to enter a number from the standard input (the entered string is converted into a *Number* and stored in variable *n*), then the function *fibo_r* is invoked and the output is printed on the standard output using the *print* directive.

```
CoreASM Fibonacci
use Standard
init InitRule
rule InitRule =
  seqblock
      n := toNumber(input("Enter n now \n:"))
      print "Fibonacci(" + n + ") using pure recursion: " + fibo_r(n)
      program(self) := undef
  endseqblock
derived fibo_r(x) =
      local r in return r in
      if x < 0 then r := 0
      else if x < 2 then r := x
      else r := fibo_r(x-2) + fibo_r(x-1)
```

Fig. 5. *CoreASM* Fibonacci Recursive Specification

The input domain for the Fibonacci example can be partitioned into three blocks: (1) negative numbers, (2) zero, and (3) positive numbers. The refinement of the resulting three blocks lead to the creation of three test cases: (TC1) input:-1, expected output:0, (TC2) input:0, expected output:0, and

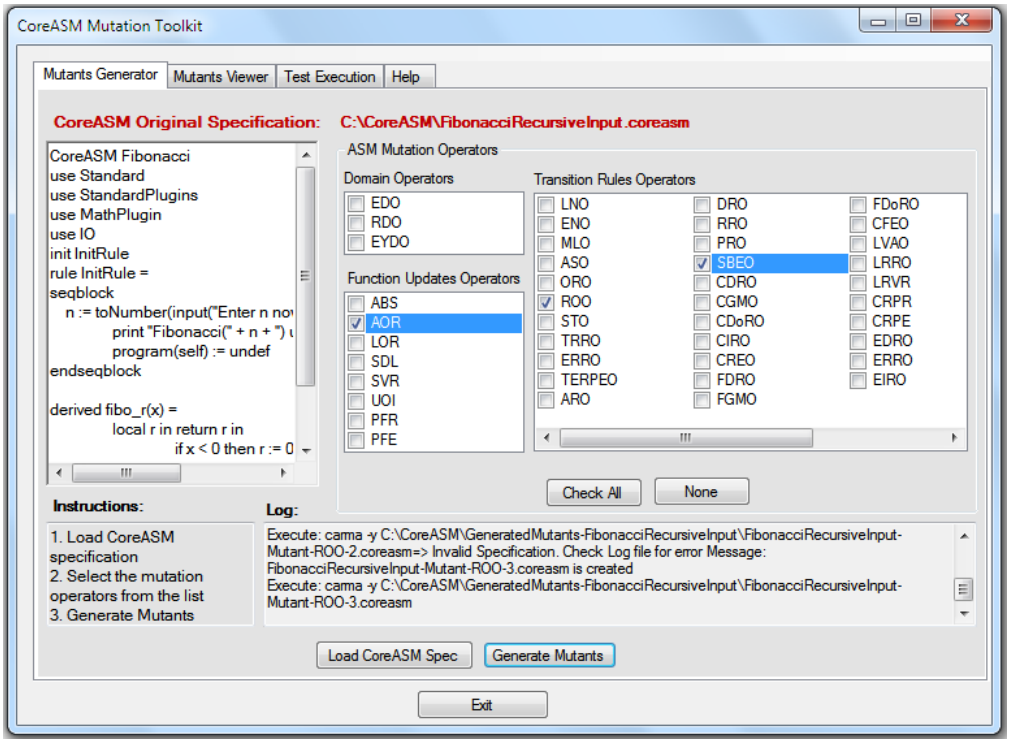


Fig. 2. CoreASM Mutation Toolkit: Mutants Generation GUI

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

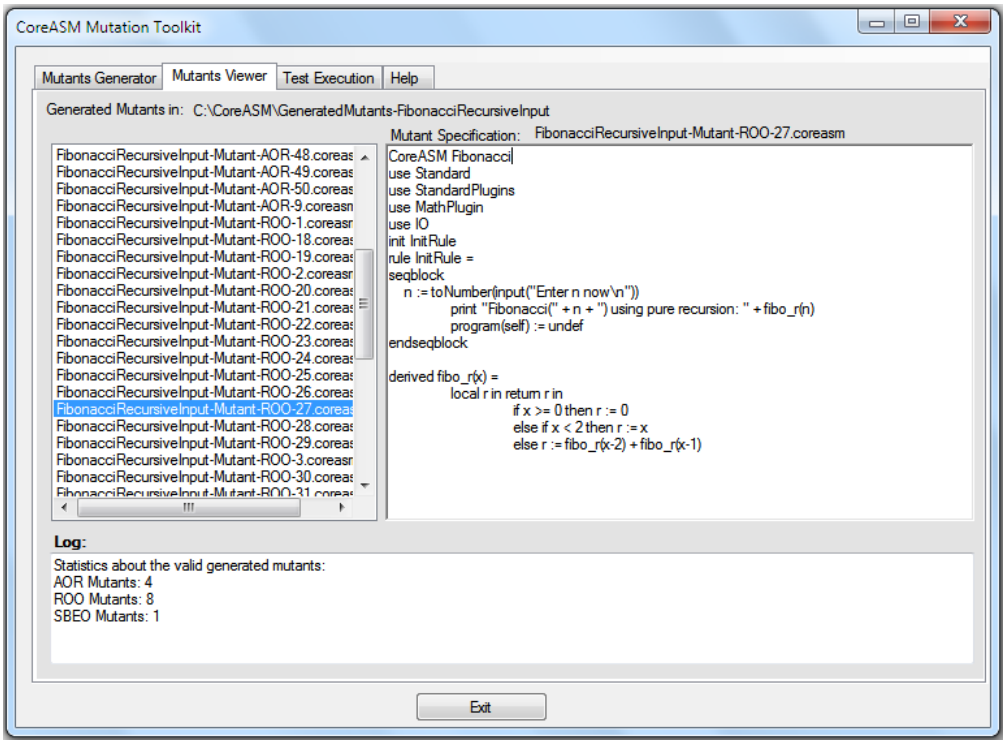


Fig. 3. *CoreASM* Mutation Toolkit: Mutants Viewer GUI

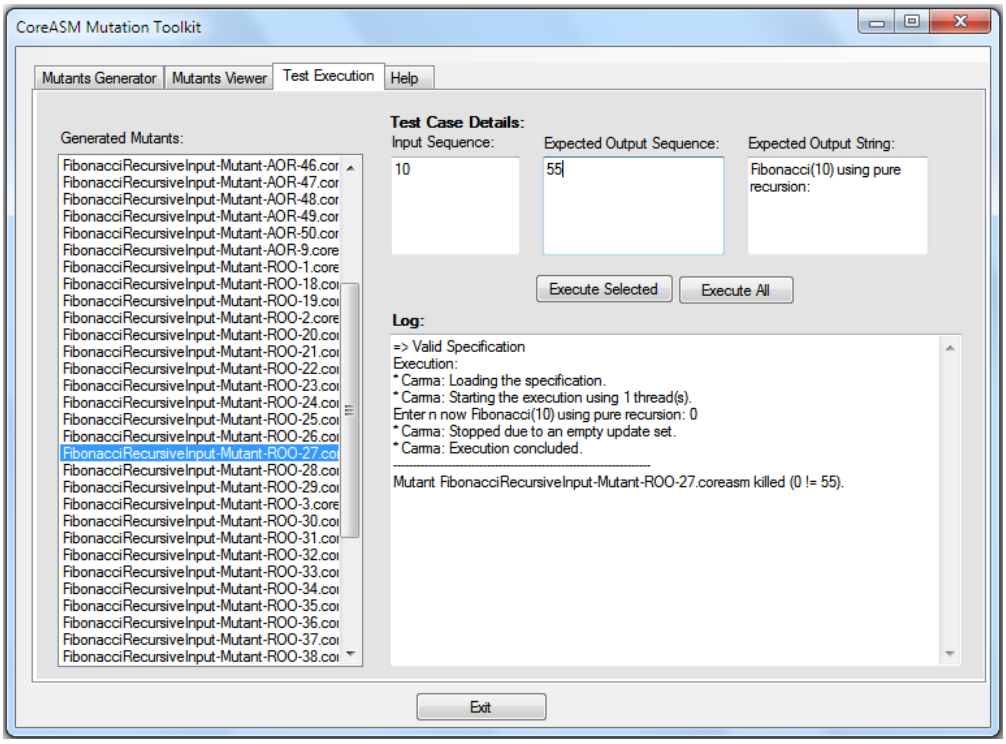


Fig. 4. *CoreASM* Mutation Toolkit: Mutants Executor GUI

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

(TC3) input:10, expected output:55.

Table XVI shows the distribution breakdown into separate operators of the 48 generated mutants (i.e, valid mutants only). The execution of a test case leads to one the following two outputs:

- A numeric output value. For example, the execution of the TC3 against mutant *FibonacciRecursiveInput-Mutant-ROO-27.coreasm* (Figure 4) produces an output equal to 0, which is different from the expected output 55. Hence the test case has killed the mutant. This mutant is said to be of type error revealing.
- A *null* output in case the execution is not conclusive. For example, the execution of TC3 against mutant *FibonacciRecursiveInput-Mutant-AOR-9.coreasm* which replaces *fibo_r(x-2)* with *fibo_r(x+2)* leads to a *null* output. Again, the test case has killed the mutant.

Fourty seven mutants have been killed by the proposed test suite. Mutant *FibonacciRecursiveInput-Mutant-ROO-1.coreasm* that replaces $x<0$ by $x<=0$) remains alive. This mutant is equivalent to the original specification and cannot be killed by any test case.

TABLE XVI. GENERATED MUTANTS STATISTICS FOR THE FIBONACCI EXAMPLE (FIGURE 5)

Mutation Operator	Number of Valid Mutants	Number of Killed Mutants
ABS	6	6
AOR	4	4
SDL	3	3
SVR	3	3
UOI	6	6
STO	4	4
ENO	2	2
ROO	8	7
TRRO	2	2
ERRO	2	2
TERPEO	1	1
CRPR	6	6
SBEO	1	1
Total	48	47

The test set effectiveness (TC_{eff}) (also called *adequacy score*) is computed by the following equation:

$$TC_{eff} = \frac{M_k}{M_t - M_e} \quad (3)$$

where M_k is the number of killed mutants, M_t is the total number of generated mutants, and M_e is the number of equivalent mutants.

A test set effectiveness score of 100% is acquired for the three proposed test cases.

VII. EMPIRICAL COMPARISON OF MUTATION OPERATORS

To empirically compare the proposed mutation operators, we ran experiments on three *CoreASM* specifications:

- Dining Philosophers [42] (98 LOC).
- Vending Machine [43] (208 LOC).
- Rail Road Crossing [44] (107 LOC).

Tables 6(a), 6(b), and 6(c) illustrate the number of resulting mutants for each mutation operators for each specification. We made the following observations:

- The number of mutants produced by domain operators is low (2, 3, and 4 respectively). Indeed, we were able to apply *EDO* only. Applying *RDO* and *EYDO* have produced syntactically incorrect mutants for the three specifications.
- The number of mutants produced by transition rules operators (e.g., *ROO*, *STO*, etc.) is the highest amongst the three categories. This is expected because the general schema of an ASM transition system appears as a set of guarded rules.
- The number of rules, the number of used variables, the number of conditions, the number of rule calls, etc. are important factors impacting the number of produced *CoreASM* mutants.
- ROO (relational operator) is the operator that have produced the largest number of mutants for the vending machine and the rail road crossing examples.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we have extended our previous work [1] on designing mutation operators for the Abstract State Machines (ASM) formalism. The developed operators are classified into three categories: (1) Domain operators, (2) function update operators, and (3) transition rules operators. Furthermore, a prototype mutation tool for the *CoreASM* language, has been built to automatically generate mutants and check their validity. We have illustrated our approach using a simple *CoreASM* implementation of the *Fibonacci* series. An initial empirical comparison of the number of generated mutants is presented and discussed.

As a future work, we are planning to enhance our empirical study by considering parameters such as the number of variables, the number of rules, etc, and by assessing the effectiveness of the defined mutation operators.

REFERENCES

[1] J. Hassine, "Absbtract state machines mutation operators," in *The Seventh International Conference on Software Engineering Advances (ICSEA 2012), Lisbon, November 18-23*, 2012, pp. 436–441.

[2] L. J. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, Aug. 1990.

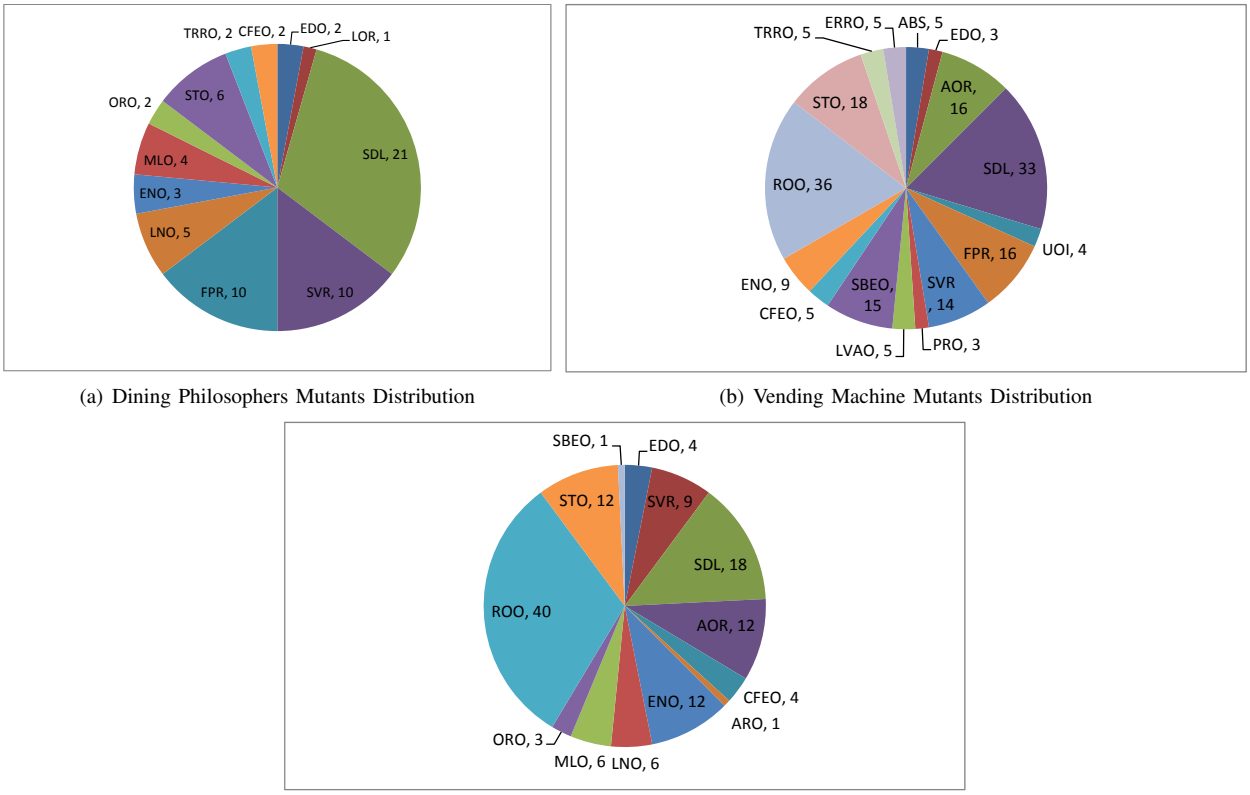
[3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978. [Online]. Available: http://dx.doi.org/10.1109/C-M.1978.218136

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

[5] P. Ammann and P. E. Black, "A specification-based coverage metric to evaluate test sets," in *The 4th IEEE International Symposium on High-Assurance Systems Engineering*, ser. HASE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 239–248.

[6] P. Chevalley and P. Thévenod-Fosse, "A mutation analysis tool for java programs," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 1, pp. 90–103, 2003.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



(a) Dining Philosophers Mutants Distribution

(b) Vending Machine Mutants Distribution

(c) Rail Road Crossing Mutants Distribution

Fig. 6. Number of Generated Mutants for Dining Philosophers, Vending Machine, and Rail Road Crossing *CoreASM* Specifications

[7] 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.

[8] A. J. Offutt, VI and K. N. King, "A fortran 77 interpreter for mutation analysis," in *Papers of the Symposium on Interpreters and interpretive techniques*, ser. SIGPLAN '87. New York, NY, USA: ACM, 1987, pp. 177–188. [Online]. Available: http://doi.acm.org/10.1145/29650.29669

[9] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software:Practice and Experience*, vol. 21, pp. 685–718, June 1991.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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–.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] Y. Gurevich, "Evolving Algebras. A Tutorial Introduction," *Bulletin of The European Association for Theoretical Computer Science*, vol. 43, pp. 264–284, 1991.

[22] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proc. London Math. Soc.*, vol. 2, no. 42, pp.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

230–265, 1936.

[23] Y. Gurevich, "Abstract state machines: An overview of the project," in *Foundations of Information and Knowledge Systems*, ser. Lecture Notes in Computer Science, D. Seipel and J. Turull-Torres, Eds. Springer Berlin Heidelberg, 2004, vol. 2942, pp. 6–13.

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

[25] C. Wallace, "The semantics of the C++ programming language," in *Specification and validation methods*. New York, NY, USA: Oxford University Press, Inc., 1995, pp. 131–164.

[26] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk, "A high-level modular definition of the semantics of c#," *Theor. Comput. Sci.*, vol. 336, no. 2-3, pp. 235–284, May 2005.

[27] E. Börger and W. Schulte, "Defining the java virtual machine as platform for provably correct java compilation," in *MFCS '98: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*. London, UK: Springer-Verlag, 1998, pp. 17–35.

[28] E. Börger and D. Rosenzweig, "A mathematical definition of full prolog," *Sci. Comput. Program.*, vol. 24, no. 3, pp. 249–286, 1995.

[29] U. Glässer, E. Börger, and W. Müller, "Formal definition of an abstract vhdl'93 simulator by ea-machines," in *Formal Semantics for VHDL*, C. Delgado Kloos and P. T. Breuer, Eds. Kluwer Academic Publishers, 1995.

[30] U. Glässer and R. Karges, "Abstract state machine semantics of SDL," *Journal of Universal Computer Science*, vol. 3, no. 12, pp. 1382–1414, 1997.

[31] R. Eschbach, U. Glässer, R. Gotzhein, M. von Löwis, and A. Prinz, "Formal definition of SDL-2000: Compiling and running SDL specifications as ASM models," *Journal of Universal Computer Science, Special Issue on Abstract State Machines - Theory and Applications*, 2001, springer-Verlag.

[32] R. Farahbod, U. Glsser, and M. Vajihollahi, "Specification and validation of the business process execution language for web services," in *Abstract State Machines 2004. Advances in Theory and Practice*, ser. Lecture Notes in Computer Science, W. Zimmermann and B. Thalheim, Eds. Springer Berlin / Heidelberg, 2004, vol. 3052, pp. 78–94.

[33] U. Glässer and Q.-P. Gu, "Formal description and analysis of a distributed location service for mobile ad hoc networks," *Theor. Comput. Sci.*, vol. 336, no. 2-3, pp. 285–309, May 2005.

[34] U. Glässer, Y. Gurevich, and M. Veanes, "Abstract communication model for distributed systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 7, pp. 458–472, Jul. 2004.

[35] Y. Gurevich, "Sequential abstract-state machines capture sequential algorithms," *ACM Trans. Comput. Logic*, vol. 1, no. 1, pp. 77–111, Jul. 2000.

[36] A. Blass and Y. Gurevich, "Abstract state machines capture parallel algorithms: Correction and extension," *ACM Trans. Comput. Logic*, vol. 9, no. 3, pp. 19:1–19:32, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1352582.1352587

[37] CoreASM, "The CoreASM Project," http://www.coreasm.org, 2012, last accessed, June 2013.

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

[39] AsmL, "Microsoft Research: The Abstract State Machine Language," http://research.microsoft.com/en-us/projects/asml/, 2006, last accessed, June 2013.

[40] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[41] 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.

[42] G. Ma and R. Farahbod, "Dining Philosphers: A Sample Specification in CoreASM," 2006, last accessed, June 2013. [Online]. Available: http://coreasm.svn.sourceforge.net/viewvc/coreasm/engine-carma/trunk/sampleSpecs/DiningPhilosophers.coreasm

[43] M. Vajihollahi and R. Farahbod, "Vending Machine CoreASM Spec," 2006, last accessed, June 2013. [Online]. Available: http://coreasm.svn.sourceforge.net/viewvc/coreasm/engine-carma/trunk/sampleSpecs/VendingMachine.coreasm

[44] R. Farahbod, "Rail Road Crossing CoreASM Spec," 2009, last accessed, June 2013. [Online]. Available: http://coreasm.svn.sourceforge.net/viewvc/coreasm/engine-carma/trunk/sampleSpecs/RailroadCrossing.coreasm

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org