Verification Approach for Refactoring Transformation Rules of State-Based Models

Nada Almasri, Bogdan Korel, and Luay Tahat

Abstract—With the increased adoption of Model-Driven Engineering (MDE), where models are being used as the primary artifact of software, it is apparent that greater attention to the quality of the models is necessary. Traditionally, refactoring is used to enhance the quality of software systems at the source-code level; however, applying refactoring at the model level will have a more significant improvement on the system. After refactoring a model, proving that it still preserves its original behavior is crucial. In this paper, we present a process for applying refactoring transformations to the Extended Finite State Machine (EFSM) models using verified transformation rules that have been proven to preserve the model's original behavior. We provide a simplified three-step verification approach that can be used to prove that a transformation rule will generate a transformed model that is semantically equivalent to the original model. To do this, we formally define semantical equivalence at three different levels of granularity: models, sub-models, and transitions. Additionally, we introduce five model transformation rules and we demonstrate how our verification approach is used to prove the correctness of these rules. Finally, we present two case studies where we apply the proposed transformation process which adopts the five verified transformation rules. Using model testing, we show that applying a sequence of transformations using the verified transformation rules will keep both the original and the transformed model semantically equivalent. Additionally, the case studies show that model transformation can be used to enhance certain pre-defined model characteristics.

Index Terms—Extended finite state machine, model refactoring, refactoring transformation rules, verification of transformations, observable behavior, semantic equivalence of models

1 Introduction

System modeling allows representing the different aspects of a system, such as its structure, behavior, and external environment. Different types of models are used to represent these different aspects. In this paper, we focus on the behavioral aspect of the system.

System models for state-based systems describe the system behavior by a set of states and transitions between these states. The most popular formal description languages used for modeling state-based systems are Extended Finite State Machines (EFSMs), Specification Description Language (SDL), and Unified Modeling Language (UML) State Charts [1], [2], [3], [4], [5]. These languages are often graphical, which makes them easy to comprehend and utilize. They have received wide industry acceptance, especially in the fields of telecommunications, embedded systems, aerospace, healthcare, insurance, biology, and computer networking, where state-based systems are prevalent [1], [5], [6].

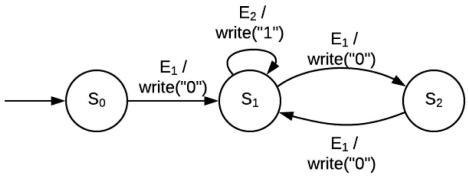
It is frequently the case that these models need to be modified in order to improve their quality characteristics such as performance, readability, maintainability, complexity, or size. This practice is commonly known as "*Refactoring*." Modifying models manually, however, is very expensive and unreliable. Therefore, to increase the confidence in the revised models, it is expected that they should be modified by automated tools using predefined transformation rules.

Transformation and refactoring are two related concepts that have originated for two different purposes. Transformation is a central concept in Model-Driven Engineering (MDE), which allows changing one model to a different model or generating the software source-code from its models. Different transformation types have different intents [22], and they are supported by different transformation languages, tools, and platforms [24], [30], [33], [34], [43], [44], [51]. The main purpose of the transformation activity is to maximize the automation of the different activities at the different stages of the software development life-cycle. Refactoring, on the other hand, was initially introduced at the source-code level to enhance its quality, reduce its complexity, and facilitate its maintenance. The main purpose of refactoring is to improve the software system's internal structure without changing its behavior [25].

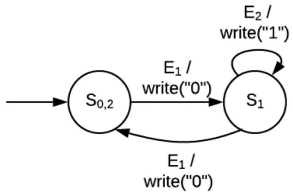
With the wide acceptance of MDE, different software engineering activities started shifting from the source-code level to the model level including software slicing [12], [18], testing [1], [2], [3], [4], [7], [8], [9], [10], [11], [50], change impact analysis [5], dependency analysis [2], [18], [23], [29], and refactoring [13], [17], [20], [21] only to name a few. With respect to refactoring, it is now being regarded by many researchers as a specific type of transformation where the source and target models are of the same model type and at the same level of abstraction [14], [21], [28], and they are being categorized as horizontal endogenous transformations [17].

- *Nada Almasri and Luay Tahat are with the MIS Department, Gulf University for Science and Technology, Hawally 32093, Kuwait. E-mail: {almasri.n, tahat.l}@gust.edu.kw.*
- *Bogdan Korel is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616 USA. E-mail: korel@iit.edu.*

Manuscript received 22 October 2020; revised 29 June 2021; accepted 15 August 2021. Date of publication 20 August 2021; date of current version 17 October 2022.
(Corresponding Author: Nada Almasri.)
Recommended for acceptance by W. Visser.
Digital Object Identifier no. 10.1109/TSE.2021.3106589



(a) Model M_1



(b) Model M_2

Fig. 1. Example of two semantically equivalent models.

When this type of transformation is applied to a source model, it is expected to generate a target model that exhibits the same observable behavior as the source model. When two models exhibit the same behavior, we call them *semantically equivalent models*. For example, assuming that a developer originally drafted the model in Fig. 1a, an automated tool (using a predefined set of model transformation rules) may detect that the model can be reduced in size if S_0 and S_2 are merged in one state. When the rule is applied to the source model of Fig. 1a, the *transformed model* of Fig. 1b is generated. Despite that the two models have different structures, they generate the same output for the same sequence of events, which makes the two models *semantically equivalent*.

Correct refactoring transformation rules guarantee that both the original and the transformed models are semantically equivalent by preserving the externally observable behavior of the original model. As new model transformation rules are being developed, developers need to have confidence in the correctness of these transformation rules. Typically, the transformed models are validated by extensive testing, which is an expensive process. The major motivation for this paper is to present an approach for the development and verification of model transformation rules. Using verified transformation rules guarantees generating transformed models that are semantically equivalent to the original models, hence, eliminating the need to re-validate the models after the transformation; which greatly saves time and efforts for the system modelers.

The approach presented in this paper targets Extended Finite State Machine (EFSM) models, but it can be generalized to other state-based models. Our approach of defining and verifying transformation rules is presented within the context of a larger process for transforming models. In this process, the transformation rule is applied to a portion of the model, referred to as a *sub-model*. The sub-model contains the elements of the model that are changed by the transformation (Section 3.1 introduces the details of the sub-models). Applying the transformation rule to a sub-model, SM, generates a transformed sub-model, SM_T. The original sub-model, SM, is then replaced with the transformed sub-model, SM_T, resulting in the transformed model.

Verifying the correctness of model transformation rules is a crucial task [38], so in this work, we present an approach to verify the transformation rules that are applied on the sub-model level. Given that the transformation rules are applied to a portion of the model instead of the whole model, the complexity of the proofs is significantly reduced, resulting in verified transformation rules which are safe to use. Verifying the transformation rule is done by proving that for any given EFSM sub-model, applying the rule generates a semantically equivalent EFSM sub-model. Transforming the whole model can thus be done by applying a sequence of safe transformations resulting in a sound model transformation that does not require further validation.

The major contributions of this paper can be summarized as follows:

- We define the concept of "semantic-equivalence" at different levels of granularity for state-based models (Sections 3.2.2, 3.2.3, and 3.2.4).
- We present a simplified three-step verification approach that can be used by system modelers to verify new transformation rules (Section 3.2.5).
- We present a transformation process (illustrated in Fig. 3) to transform models by iteratively applying verified transformation rules to sub-models.
- We demonstrate the use of the verification approach by developing five transformation rules along with the theoretical proofs of their correctness (Section 4).
- We introduce a mechanism for the automation of the partial verification of the transformation rules (Section 5).
- We present two case studies demonstrating the application of model transformations in improving different characteristics of EFSM models (Section 6).

The rest of the paper is organized as follows: Section 2 provides an overview of EFSM and the formal notation used in the paper. Section 3 presents the model transformation process and the proposed verification approach. In Section 4, five transformation rules are presented and verified using the proposed three-step verification approach. Section 5 discusses the automation of the presented approach. Two case studies are presented in Section 6, while the related work and conclusion are presented in Sections 7 and 8, respectively.

2 Preliminaries

2.1 EFSM Model Notation

In this paper, we address model transformation on the Extended Finite State Machine (EFSM). EFSM models consist of states (including a start state and an exit state) and transitions between states. The following elements are associated with each transition: an event, a condition, and a sequence of actions. A transition is triggered at its originating state when an event occurs (i.e., an input is received), and an enabling condition (i.e., a Boolean expression) associated with the transition is satisfied. When the transition is triggered, a sequence of actions may be performed (which may manipulate variables and produce an output), and the system is transferred to the target state of the transition.

An EFSM M is expressed formally as a seven tuple: $M = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$, where:

Σ, or Σ(M), is a set of events that the model M reacts to, where an event is denoted as ***E***

Q, or Q(M), is a set of all the "internal" states in the model M, i.e., non-entry and non-exit states, where a state is denoted as S

Q_{en}, or Q_{en}(M), is a set with one entry state

Q_{ex}, or Q_{ex}(M), is a set of exit states // Q_{ex}(M) may be empty (no exit state)

V, or V(M), is a finite set of model variables in the model M

O, or O(M), is the set of actions defined in the model M, and $O = \{O_E \cup O_I\}$ where:

O_E is the set of external actions which are visible to an outside observer

O_I is the set of internal actions that are invisible to an outside observer

R, or R(M), is the set of transitions in the model M, where a transition is denoted as ***T***

A transition ***T*** in the model M is represented by a five-tuple: $T = (E, C, A, S_b, S_e)$, where:

E is the event triggering the transition T in the model M, and $E \in \Sigma(M)$. E is also represented as E(args[]) where args is an array of arguments that can be used only locally by the condition and actions of the transition T.

C, also denoted as C(T), is an enabling condition for the transition T, and it is defined over V(M) and the parameters agrs[] of the event E,

S_b is the transition's originating state. S_b can also be denoted as S_b(T), and $S_b \in \{Q(M) \cup Q_{en}(M)\}$

S_e is the transition's terminating state. S_e can also be denoted as S_e(T), and $S_e \in \{ Q(M) \cup Q_{ex}(M)\}$

A, also denoted as A(T), is a sequence of actions performed when the transition T is triggered, A is also represented as $A(T) = \langle a_1, a_2, \ldots, a_j \rangle$, where $a_i \in O(M)$. An action could manipulate variables, display output, or perform similar tasks. When an action **a** manipulates variables, we use U(a) and D(a) to represent the sets of manipulated variables, where:

U(a) represents the set of variables used/read by the action.

D(a) represents the set of variables defined/written by the action.

Additionally, we define A_E, also denoted as A_E(T), as the sequence of external actions performed when the transition T is triggered. A_E(T) is also represented as $A_E(T) = \langle a_{e1}, a_{e2}, \ldots, a_{ek} \rangle$, where $a_{ei} \in O_E$.

Any state ***S*** within a model has three different sets of transitions, and they are denoted as follows:

$T_r(S)$: is the set of self-looping transitions of the state S, more formally:

$$T_r(S) = \{ T \in R(M) : S_b(T) = S, \ S_e(T) = S\}$$

T_{in}(S) is the set of all incoming transitions with respect to state S, more formally:

$$T_{in}(S) = \{T \in R(M) : \ S_b(T) \neq S , \ S_e = S\}$$

T_{out}(S) is the set of all outgoing transitions with respect to state S, more formally:

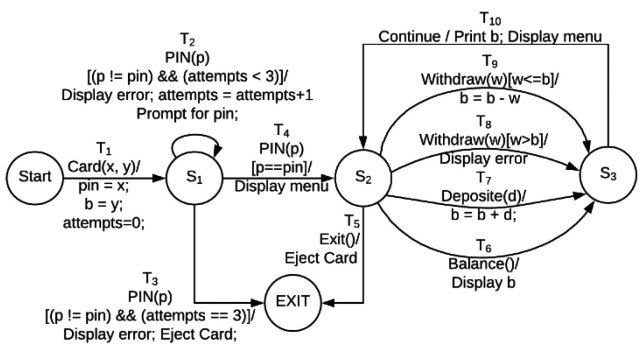


Fig. 2. EFSM example of an ATM.

$$T_{out}(S) = \{ T \in R(M) : \ S_b(T) = S , \ S_e(T) \neq S\}$$

An example of an EFSM is presented in Fig. 2. This model represents a simplified Automated Teller Machine (ATM) where the user starts by inserting the bank card, and upon entering the correct pin for the card, the user will be allowed to withdraw or deposit money.

2.2 EFSM Model Execution

In this paper, we assume that EFSM models are executable, i.e., enough details are provided in the model so that the model executor can execute the model based on the model's specifications.

An input to the EFSM is a sequence of events with values for arguments associated with the events. We represent the input as:

$$ES = \langle E_1, E_2, \ldots, E_j \rangle, \ where \ E_i \in \Sigma(M)$$

The execution result of the input can be expressed as the sequence of executed transitions, or it can be expressed at a finer granularity showing the particular actions executed. We represent the execution of the machine as a function of the sequence of events and the model on which the events are executed. We use TS when looking at the execution results in terms of the executed transitions, while we use AS when looking at the execution results in terms of actions. TS and AS are expressed as follows:

$$TS(M, ES) = \langle T_1, T_2, \ldots T_j \rangle, \ where \ T_i \in R(M)$$

$$AS(M, ES) = \langle A(T_1), A(T_2), \ldots, A(T_i) \rangle, \ where:$$

$$A(T_i) = \langle a_1, a_2, \ldots, a_n \rangle, \ where \ a_i \in O(M)$$

Finally, we use AS_E to express the sequence of external actions that are visible to an outside observer when the sequence of events ES is executed on the model M. AS_E is expressed as follows:

$$AS_E(M, ES) = \langle A_E(T_1), A_E(T_2), \ldots, A_E(T_j) \rangle, \ where:$$

$$A_E(T_i) = \langle a_{e1}, a_{e2}, \ldots, a_{ek} \rangle, \ where \ a_{ei} \in O_E(M)$$

For example, consider the following sequence of events for the ATM system of Fig. 2:

ES = <Card(1234, 200), PIN(1234), Deposit(20), Continue (), Exit()>. When the model M of Fig. 2 is executed on the sequence of events ES, the following sequence of transitions are executed:

$$TS(M, ES) = \langle T_1, T_4, T_7, T_{10}, T_5 \rangle$$

When narrowing down the granularity to the action level, the execution causes the following sequence of actions to take place:

$$\mathrm{AS(M, ES)} = <\mathrm{A(T_1)},\ \mathrm{A(T_4)},\ \mathrm{A(T_7)},\ \mathrm{A(T_{10})},\ \mathrm{A(T_5)}>$$

Given that $\mathrm{A(T_1)}$ = <pin = x, b = y, attempts = 0>, $\mathrm{A(T_4)}$ = <Display menu>, $\mathrm{A(T_7)}$ = <b = b+d>, $\mathrm{A(T_{10})}$ = <Print b, Display menu>, and $\mathrm{A(T_5)}$ = <Eject car>, then AS can be written: AS(M, ES) = <pin = x, b = y, attempts = 0, Display menu, b = b + d, Print b, Display menu, Eject card>.

Additionally, we can narrow the execution results down by considering the external actions, $\mathrm{a} \in \mathrm{O_E}$, which are visible to the outside observer:

$$AS_E(M, ES) = \langle A_E(T_1), A_E(T_4), A_E(T_7), A_E(T_{10}), A_E(T_5) \rangle$$

Given that $\mathrm{A_E(T_1)}$ = <>, $\mathrm{A_E(T_4)}$ = <Display menu>, $\mathrm{A_E(T_7)}$ = <>, $\mathrm{A_E(T_{10})}$ = <Print b, Display menu>, and $\mathrm{A_E(T_5)}$ = <Eject card>, then $\mathrm{AS_E}$ can be written as follows:

$\mathrm{AS_E}$(M, ES) = < Display menu, Print b, Display menu, Eject card>.

Finally, the actual visible result/values for the above external actions are represented as : $Res(M, ES) = \langle menu,\ 220,\ card\ ejected \rangle$

3 Model Transformation and Verification

Our transformation approach proposes transforming models through a transformation process which iteratively applies verified transformation rules. This transformation approach has two aspects to it; the transformation process which applies verified transformation rules, and a mechanism to verify the transformation rules prior to adopting them in the process.

In each iteration of the transformation process, only a single verified transformation rule is applied to a portion of the model, referred to as *sub-model, SM*. Limiting the scope of an iteration to a single transformation rule and a single sub-model helps in ensuring that the transformed model preserves its original behavior. This is mainly because the verification approach suggested in Section 3.2 targets proving that a transformation rule is semantic-preserving when applied to a sub-model while keeping the remaining parts of the model unchanged.

In the following Section 3.1 we introduce the transformation process along with the notations and algorithms used to support this process. In Section 3.2, we introduce our proposed mechanism to verify new transformation rules prior to using them in model transformations.

3.1 Overview of the Transformation Process

The proposed model transformation process is diagramed in Fig. 3. The process is applied iteratively where in each iteration the original model M goes through the full cycle as depicted in steps 1-7 of Fig. 3.

Given the original model M, the sub-model to be transformed can be identified using the expert analysis of a developer or using a search algorithm targeting the improvement of certain model characteristics [13]. Having identified the sub-model, *SM*, to be transformed (step 1 in Fig. 3), the next step is to detach it from the model M.

The result of this step consists of two parts, the sub-model to be transformed (*SM*), and the unchanged parts of the original model (M – *SM*). A sub-model is identified by a sub-set of states of M, referred to as *internal states* of SM (as described in 3.1.1). For example, the internal state of *SM* in Fig. 3 are states F, G, and H. Based on these internal states, *entry* and *exit* states to/from the sub-model are determined (in Fig. 3, D, and E are entry states to SM, and I and J are exit states from SM). Entry and exit states are common states between the sub-model (SM) and the remaining part of the model (M-SM). After detaching the sub-model SM from model M, the transformation rule is then applied to the stand-alone sub-model SM (step 4 in Fig. 3), resulting in the transformed sub-model $\mathrm{SM_T}$ which has the internal states F', H', and N'. Finally, in the last step of this transformation cycle, the transformed sub-model $\mathrm{SM_T}$ is attached (re-connected) with the remaining part of the model (M-SM) through the common entry and exit states (as described in 3.1.2). This final step results in a transformed version of the model, $\mathrm{M_T}$.

The above cycle (steps 1-7) can then be repeated several times where each time a single verified transformation rule is applied to the model. Given that the suggested process applies only verified transformation rules, any model derived from the original model using a sequence of correct/semantic-preserving transformation rules is semantically equivalent to the original model.

In the remainder of this section, we present the formal definition of a sub-model along with an algorithm to compute the sub-model elements (3.1.1). The algorithm to replace the original sub-model with the transformed one is presented in 3.1.2.

3.1.1 Sub-Model Notation and Computation

Let M be an EFSM model. A sub-model SM of M with respect to a subset of internal states of M is represented as $SM = (\Sigma,\ Q, Q_{en},\ Q_{ex},\ V, O,\ R)$, where:

Σ is a set of events that the sub-model SM reacts to, where: $\Sigma \subseteq \Sigma(\mathrm{M})$, and

$$\Sigma = \{\mathrm{E} \in \Sigma(\mathrm{M}) :\ \exists \text{ a transition } \mathrm{T} \in \mathrm{R} \text{ where } \mathrm{E} \in \mathrm{E(T)}\}$$

Q, also denoted as Q(SM) or $\mathrm{Q_{SM}}$, is the set of "internal" sub-model states, where $\mathrm{Q(SM)} \subset \mathrm{Q(M)}$

$\mathrm{Q_{en}}$ is a set of entry states to SM, where: $\mathrm{Q_{en}} \subset \mathrm{Q(M)} \cup \mathrm{Q_{en}(M)}$, and

$$\mathrm{Q_{en}} = \{\mathrm{S} \in (\mathrm{Q(M)} \cup \mathrm{Q_{en}(M)}) :\ \mathrm{S} \notin \mathrm{Q}, \exists\ \mathrm{T} \in \mathrm{R(M)}, \text{ where } \mathrm{S_b(T)} = \mathrm{S} \text{ and } S_e(\mathrm{T}) \in \mathrm{Q}\}$$

$\mathrm{Q_{ex}}$ is a set of exit states from SM, where: $\mathrm{Q_{ex}} \subset \mathrm{Q(M)} \cup \mathrm{Q_{ex}(M)}$, and

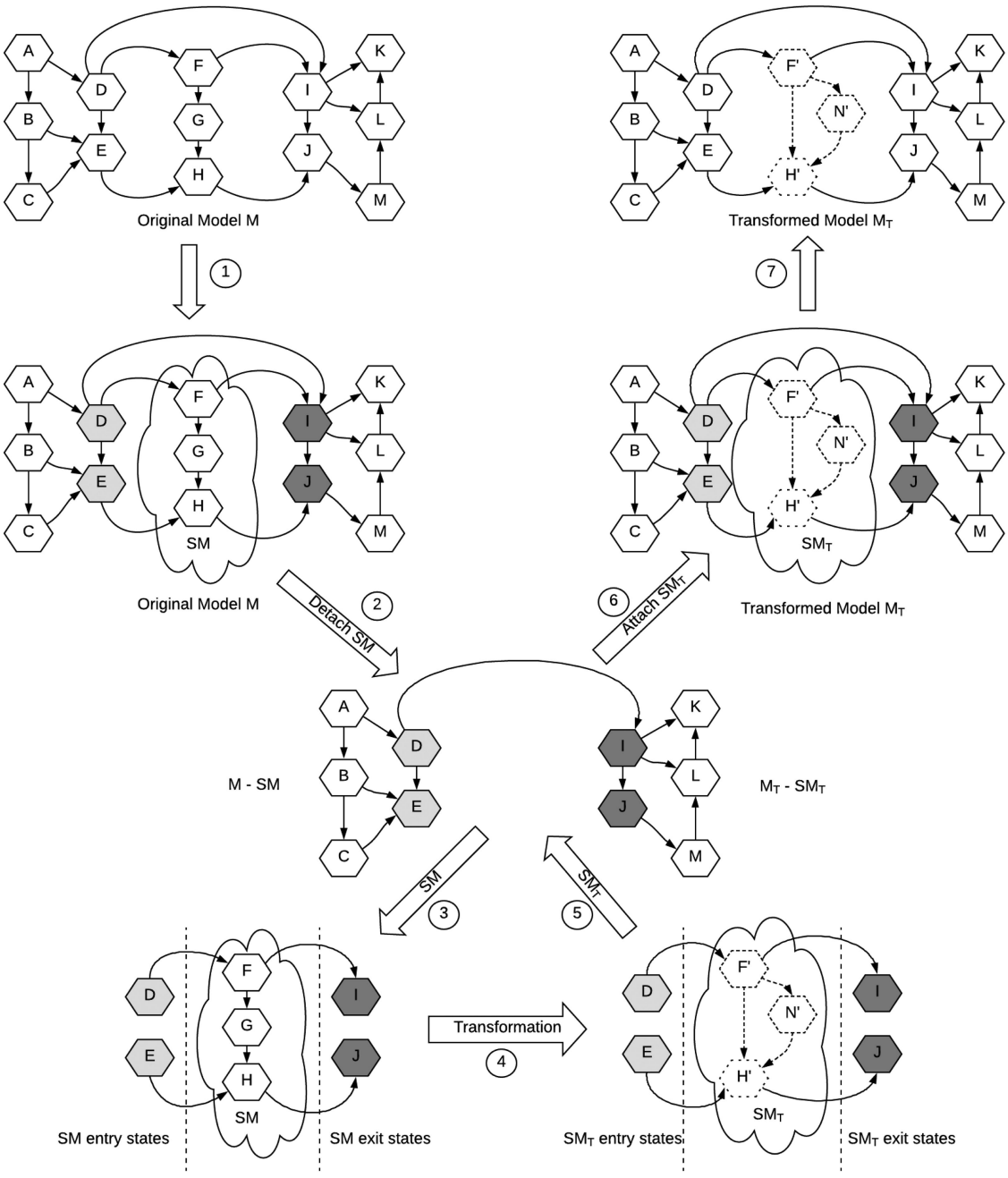


Fig. 3. Model transformation process.

$Q_{ex} = \{S \in Q(M) \cup Q_{ex}(M) : S \notin Q, \exists T \in R(M),$ where $S_b(T) \in Q$ and $S_e(T) = S\}$

V is a set of sub-model variables, where V ⊆ V(M), and

$V = \{v \in V(M): \exists$ a transition $T \in R$ where $v \in V(T)\}$

O is a set of sub-model actions, where O ⊆ O(M), and

$O = \{a \in O(M): \exists$ a transition $T \in R$ where $a \in O(T)\}$

R is the set of transitions of SM, where: R ⊂ R(M), and

$R = T_r \cup T_{in} \cup T_{out}$, where:

T_r is a set of "internal" transitions (i.e., transitions between internal states of SM), $T_r \subset R(M)$, and $T_r = \{T \in R(M): S_b(T) \in Q, S_e(T) \in Q\}$

T_{in} is a set of "incoming" transitions from entry states to internal states of SM, $T_{in} \subset R(M)$, and $T_{in} = \{T \in R(M): S_b(T) \in Q_{en}, S_e(T) \in Q\}$

T_{out} is a set of "outgoing/exiting" transitions from internal states to exit states of SM, $T_{out} \subset R(M)$, and $T_{out} = \{T \in R(M): S_b(T) \in Q, S_e(T) \in Q_{ex}\}$

Notice that for a sub-model SM, its components can also be referred to as Q(SM), R(SM), V(SM), O(SM), and Σ(SM).

Algorithm 1 shows how to compute a sub-model SM of model M with respect to a set of internal states Q(SM) ⊂ Q (M). The input to the algorithm is model M and subset Q (SM) of internal states of M. This set of internal states could be identified by a developer or using a search-algorithm [13], [15]. In step 1, the set of internal states Q of the sub-model is assigned the input subset Q(SM). In steps 2-6, the sub-model entry states, Q_{en}, and exist states, Q_{en}, are computed. Any state $S \notin Q(SM)$ that has an outgoing transition to an internal state of SM is identified as an entry state, and

any state $S \notin Q(SM)$ that has an incoming transition from an internal state of SM is identified as an exit state. In steps 7-12, the transitions of the sub-model are computed. Internal transitions, T_r, are between internal states of SM. Incoming transitions, T_{in}, are between the entry states of Q_{en} and the internal states of SM. Outgoing transitions, T_{out}, are between the internal states of SM and the exit states of Q_{ex}. In steps 14-19, the sub-model events, Σ, variables, V, and actions, O, are computed. Any model event, model variable, or model action that is a part of at least one sub-model transition is also a sub-model event, sub-model variable, or sub-model action.

Algorithm 1. ComputeSM(M, Q_{SM})

Input

M // model
$Q_{SM} \subset Q(M)$ // a set of internal states of a sub-model

Output

//sub-model of M with respect to Q(SM)
$SM = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$

Algorithm

1 $Q = Q_{SM}$
2 $Q_{en} = \emptyset$; $Q_{ex} = \emptyset$
3 **for every** $T \in R(M)$ **do**
4 **if** ($S_b(T) \notin Q$ and $S_e(T) \in Q$) **then** $Q_{en} = Q_{en} \cup \{S_b(T)\}$
5 **if** ($S_b(T) \in Q$ and $S_e(T) \notin Q$) **then** $Q_{ex} = Q_{ex} \cup \{S_e(T)\}$
6 **endfor**
7 $T_r = \emptyset$; $T_{in} = \emptyset$; $T_{out} = \emptyset$
8 **for every** $T \in R(M)$ **do**
9 **if** ($S_b(T) \in Q$ and $S_e(T) \in Q$ **then** $T_r = T_r \cup \{T\}$
10 **if** ($S_b(T) \in Q_{en}$ and $S_e(T) \in Q$) **then** $T_{in} = T_{in} \cup \{T\}$
11 **if** ($S_b(T) \in Q$ and $S_e(T) \in Q_{ex}$) **then** $T_{out} = T_{out} \cup \{T\}$
12 **endfor**
13 $R = T_r \cup T_{in} \cup T_{out}$
14 $\Sigma = \emptyset$; $V = \emptyset$; $O = \emptyset$
15 **for every** $T \in R(M)$ **do**
16 $\Sigma = \Sigma \cup \{E(T)\}$
17 $V = V \cup \{V(T)\}$ //V(T) a set of model variables
18 $O = O \cup \{O(T)\}$
19 **endfor**
20 $SM = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$
21 **return** SM

In Fig. 4, sample sub-models of the model of Fig. 2 are shown. The sub-model of Fig. 4a is identified by two internal states: S_2 and S_3. This sub-model has one entry state, S_1, and one exit state, EXIT. In addition, it has one entry transition, T_4, and one exit transition, T_5. The sample sub-model of Fig. 4b is defined by internal state S_3, where state S_2 is the entry and the exit state at the same time. This sub-model also has four entry transitions (T_6, T_7, T_8, T_9) and one exit transition T_{10}.

3.1.2 Replacing Two Sub-Models

After identifying the sub-model to be transformed and transforming it by following a certain verified transformation rule (examples of transformation rules are presented in Section 4), the last step of the model transformation process is to replace the original sub-model with the transformed sub-model. Algorithm 2 shows how one sub-model is replaced with another. The input to the algorithm is the original model M, the sub-model SM, and the transformed sub-model SM_T. The output is the transformed model M_T. In step 1, all events of M are assigned to M_T. In step 2, internal states of SM are removed from M, and internal states of SM_T are added. In steps 3 and 4, entry and exit states of the sub-models are included in M_T. In step 5, variables of SM are removed from M, and variables of SM_T are added. Similarly, actions of SM are removed and replaced with actions of SM_T in step 6. Finally, in step 7, transitions of SM are removed and replaced with transitions of SM_T.

Algorithm 2. Replace sub-model SM with transformed sub-model SM_T

Input

Model: M
Sub-model: SM
Transformed sub-model: SM_T

Output

Transformed model: $M_T = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$

Algorithm

1 $\Sigma = \Sigma(M)$
2 $Q = (Q(M) - Q(SM)) \cup Q(SM_T)$
3 $Q_{en} = Q_{en}(M)$
4 $Q_{ex} = Q_{ex}(M)$
5 $V = (V(M) - V(SM)) \cup V(SM_T)$
6 $O = (O(M) - O(SM)) \cup O(SM_T)$
7 $R = (R(M) - R(SM)) \cup R(SM_T)$
8 $M_T = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$

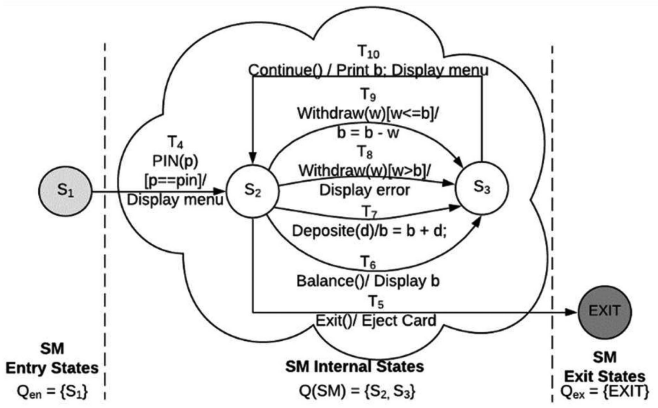
It is worth noting here that replacing SM with a semantically equivalent SM_T generates a transformed model M_T, which is also semantically equivalent to the original model M. The semantical equivalence is guaranteed given the assumption that the transformation process uses only verified transformation rules. Consequently, the process eliminates the need to re-validate the model after the transformation.

The next section introduces the verification mechanism that can be used to prove the correctness of the transformation rules prior to using them in this transformation process.

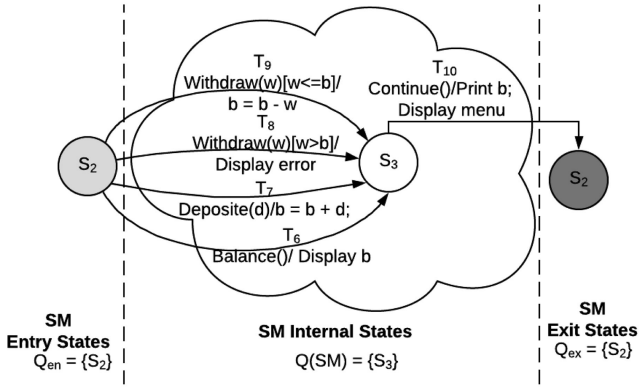
3.2 Verifying Transformation Rules

The crucial requirement for any refactoring transformation rule is that it should preserve a behavioral property, referred to as a *semantic equivalence*, of the original and the transformed models. Our proposed verification approach provides a mechanism to show that applying the rule to any given model will generate a semantically equivalent transformed model.

In this proposed verification approach, we introduce three main steps to prove that a rule is semantic-preserving (the three steps are detailed in 3.2.5). To come up with these three steps, we start by formally defining the concept of semantical equivalence between two models; then, we focus on defining the semantical equivalence of two sub-models given that the remaining parts of their respective models are identical. Finally, we show that it is possible to prove the semantical equivalence of two sub-models by looking at



(a) Sample sub-model having Q(SM) = {S_2, S_3}



(b) Sample sub-model having Q(SM) = {S_3}

Fig. 4. Sample sub-models of the model of Fig. 2.

the semantical equivalence of the individual transitions involved in the execution of a sequence of events between any pair of entry-exist states of the sub-models, and for any give sequence of events. The notation used for the executability of sub-models is presented in Section 3.2.1. In Sections 3.2.2, 3.2.3, and 3.2.4, we formally define semantical equivalence at the model level, sub-model level, and transition level. Finally, in Section 3.2.5, we present a simple three-step approach that can be followed to prove the correctness of new transformation rules.

3.2.1 Sub-Model Execution

Let $\mathrm{V(M)} = \{\mathrm{v_1}, \mathrm{v_2}, \ldots, \mathrm{v_m}\}$ be a set of model variables and d(v) be a domain of model variable v∈V(M), i.e., a set of all possible values of v. Let $\mathrm{d(M)} = \mathrm{d(v1)} \times \mathrm{d(v2)} \times \ldots \times \mathrm{d(vm)}$ be a model domain of variables, i.e., a set of all possible combinations of values of model variables. Let $Z \in$ d(M), referred to as a *state-of-model-variables*, represents the values of the model variables at some point during model execution.

Let SM be a sub-model of model M. Let $\mathrm{ES(S_i},\ Z_i,\ S_o) = <E_1, \ldots, E_k>$ be an event sequence on which SM is executed from entry state $\mathrm{S_i} \in \mathrm{Q_{en}(SM)}$ to exit state $\mathrm{S_o} \in \mathrm{Q_{ex}(SM)}$, where the sub-model is executed as follows:

1) SM is in entry state $\mathrm{S_i}$ with state-of-model-variables Z_i.
2) On event $\mathrm{E_1}$, the execution enters SM from Si.
3) On all events in ES($\mathrm{S_i}$, $\mathrm{Z_i}$, $\mathrm{S_o}$) between $\mathrm{E_1}$ and $\mathrm{E_k}$, SM is executed inside of the sub-model, i.e., the execution does not reach any exit state of SM.
4) On event $\mathrm{E_k}$, the execution exits SM by entering exit state $\mathrm{S_o}$.

We refer to such an event sequence as a *sub-model entry-exit event sequence*. The following example shows the use of this notation.

Let SM be the sub-model of Fig. 4*a having the following:*

$Q(SM) = \{S_2,\ S_3\},\ Q_{en}(SM) = \{S_1\},\ Q_{ex}(SM) = \{EXIT\}$
$R(SM) = \{T_6,\ T_7,\ T_8,\ T_9,\ T_10,\ T_4,\ T_5\},\ where$
$T_r(SM) = \{T_6,\ T_7,\ T_8,\ T_9,\ T_{10}\},$
$T_{in}(SM) = \{T_4\},\ and\ T_{out}(SM) = \{T_5\}$
$\Sigma(SM) = \{PIN(),\ Deposit(),\ Withdraw(),\ Balance(),\ Exit()\}$
$V(SM) = \{pin,\ b\}$
O(SM) = {Display Menu, b = b-w, b = b+d; Display b, Display error, Print b, Eject Card}, where
$O_E(SM)$ *= {Display Menu, Display b, Display error, Print b, Eject Card},* $O_I(SM)$ *= {b = b-w, b = b+d }*
Let SM be in entry state S_1 *with state-of-model-variables* Z *= {(pin = 1234), (b = 200), (attempts = 1)}.*
Let ES(S_1*, Z, EXIT) = <PIN(1234), Deposit(20), Continue (), Exit()> be a sub-model entry-exit event sequence.*

On the above entry-exit event sequence, SM is executed from entry state $\mathrm{S_1}$ to the exit state, EXIT, with the following sequences of transitions, actions, and observable outputs:

Sequence of transitions: TS(SM, ES) = $<T_4, T_7, T_{10}, T_5>$
Sequence of actions: AS(SM, ES) = <Display menu, b = b+d, Print b, Display menu, Eject card>
Sequence of external actions: AS_E*(SM, ES) = <Display menu, Print b, Display menu, Eject card>*
Sequence of Observable outputs: Res(SM,ES) = <menu, 220, menu, card ejected>

At EXIT state, the following is the *state-of-model-variables* Z_o *= {pin = 1234, b = 220, attempts = 1}.*

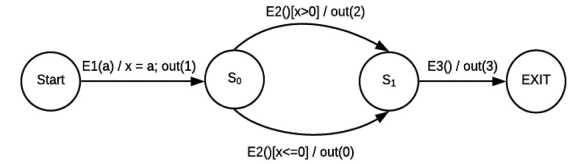
The concept of a sub-model *entry-exit event sequence* is used in Section 3.2.3 to define the semantical equivalence of sub-models.

3.2.2 Semantical Equivalence of Models

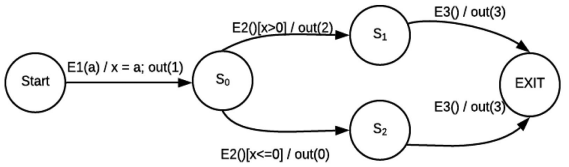
We define the concept of semantic equivalence of two models with respect to their behavior. In general, for two models to be semantically equivalent, we expect them to have the same observable output when executed under the identical inputs. Informally, two models M and model $\mathrm{M_T}$ are semantically equivalent, or behaviorally equivalent, if for any sequence of events, the "external observer" does not see any difference in the observable output of actions during the execution of model M and model $\mathrm{M_T}$.

More formally, we say that models M and $\mathrm{M_T}$ are semantically equivalent if during the execution of model M and $\mathrm{M_T}$ on any sequence of events ES = <$\mathrm{E_1}$, $\mathrm{E_2}$, . . ., $\mathrm{E_k}$>:

1) The sequence of external actions $\mathrm{AS_E(M, ES)}$ performed during execution of M on ES is identical to the sequence of external actions $\mathrm{AS_E(M_T, ES)}$ performed during execution of $\mathrm{M_T}$ on ES, and



(a) Model M



(b) Model M_T

Fig. 5. Example of semantically equivalent models.

2) The externally observable output produced by $AS_E(M, ES)$ is identical to the externally observable output produced by $AS_E(M_T, ES)$.

For example, it is not difficult to see that the two models of Fig. 5 are semantically (behaviorally) equivalent, where action "out()" is an external action, i.e., the result of its execution is visible to the external observer. On the other hand, action "x = a" is an internal action, which is not visible by the external observer. For these models, there is only one sequence of events $ES = <E_1(a), E_2(), E_3()>$ with different values of the parameter *a*. The following sequences of external actions are performed in both models on ES depending on the value of *a*:

a>0: out(1), out(2), out(3) resulting in the observable output: 1, 2, 3
a≤0: out(1), out(0), out(3) resulting in the observable output: 1, 0, 3

Since both models perform the same sequence of actions and produce the same external output for any sequence of events, ES, these models are semantically equivalent.

In general, it may be difficult or impossible to determine that two models are semantically equivalent, for all possible sequences of events. Consequently, in the next section, we present the concept of semantical equivalence of two models through the semantical equivalence of their sub-models, knowing that the transformation process described in Fig. 3 applies transformations on sub-models.

3.2.3 Semantical Equivalency of Sub-Models

Let M and M_T be two models. Let SM be a sub-model of M and SM_T be a sub-model of M_T, where,

1. SM and SM_T have the same entry states, exit states, external actions, and events:
   a. $Q_{en}(SM) = Q_{en}(SM_T)$
   b. $Qex(SM) = Q_{ex}(SM_T)$
   c. $O_E(SM) = O_E(SM_T)$
   d. $\Sigma(SM) = \Sigma(SM_T)$
2. M and M_T are identical "outside" of SM and SM_T:
   a. $Q(M) - Q(SM) = Q(M_T) - Q(SM_T)$
   b. $Q_{en}(M) = Q_{en}(M_T)$
   c. $Q_{ex}(M) = Q_ex(M_T)$
   d. $R(M) - R(SM) = R(M_T) - R(SM_T)$
   e. $V(M) - V(SM) = V(M_T) - V(SM_T)$
   f. $\Sigma(M) = \Sigma(M_T)$

In order for models M and M_T to be semantically equivalent, sub-models SM and SM_T must be semantically equivalent.

Sub-models SM and SM_T are semantically equivalent with respect to entry state S_i and exit state S_o of SM and SM_T if for any state-of-model-variables Z_i in S_o and for any entry-exit event sequence $ES(S_i, Z_i, S_o)$ of SM (or SM_T) the following holds:

1. $ES(S_i, Z_i, S_o)$ is also an entry-exit event sequence of SM_T (or SM).
2. The sequence of external actions performed during execution of SM and SM_T on $ES(S_i,Z_i,S_o)$ are identical: $\mathrm{AS_E(SM, ES(S_i, Z_i, S_o))} = \mathrm{AS_E(SM_T, ES(S_i, Z_i, S_o))}$
3. The externally observable output Res(SM, $ES(S_i, Z_i, S_o)$) produced by SM is identical to the externally observable output Res(SM_T, $ES(S_i, Z_i, S_o)$) produced by SM_T: $\mathrm{Res(SM, ES(S_i, Z_i, S_o))} = \mathrm{Res(SM_T, ES}(S_i, Z_i, S_o))$
4. At the exit state, S_o, state-of-model-variables, Z_o, is the same in SM and SM_T.

Sub-models SM and SM_T are *semantically equivalent* if for any entry-exit state pair, $(S_i, S_o) \in Q_{en}(SM) \times Q_{ex}(SM)$, SM, and SM_T are semantically equivalent with respect to states S_i and S_o. Semantical equivalency of two sub-models can be further simplified when considering the semantical equivalence of the transitions executed on both sub-models for the same sequence of events, as explained in the following sub-sections.

3.2.4 Semantical Equivalency of Transitions

Assuming that during the execution of models M and M_T on any sequence of events $ES = <E_1, E_2, \ldots, E_k>$, a sequence of transitions $TS(M, ES) = <T_1, \ldots, T_j, \ldots, T_k>$ is executed in M and $TS(M_T, ES) = <T'_1, \ldots, T'_j, \ldots, T'_k>$ in M_T. We say that transitions $T_j = (E_j, C_j, A_j, S_{bj}, S_{ej})$ and $T'_j = (E'_j, C'_j, A'_j, S'_{bj}, S'_{ej})$ are *semantically equivalent* if

- $E_j = E_j'$
- The sequence of external actions of T_j and T'_j (denoted as $A_E(T_j)$ and $A_E(T'_j)$ respectively) are the same, i.e., $A_E(T_j) = A_E(T'_j)$, and they produce the same externally observable output.

For example, two identical transitions $T_j = (E, C, A, S_b, S_e)$ and $T'_j = (E, C, A, S_b, S_e)$ are semantically equivalent. In addition, it is not difficult to see that two transitions $T_j = (E_j, C_j, A_j, S_{bj}, S_{ej})$ and $T'_j = (E'_j, C'_j, A'_j, S'_{bj}, S'_{ej})$ for which $E_j = E'_j$, $C_j = C'_j$ and $A_j = A'_j$ are semantically equivalent since the sequence of actions is identical in both transitions. Other examples of semantically equivalent transitions will be discussed later in the paper.

3.2.5 Verification Steps

In order to prove that two sub-models are semantically equivalent, it is sufficient to demonstrate that for any entry-exit state pair, $(S_i, S_o) \in Q_{en}(SM) \times Q_{ex}(SM)$, and for any sub-model entry-exit event sequence, $ES(S_i, Z_i, S_o) = <E_1, \ldots E_j, \ldots, E_k>$, semantically equivalent transitions are executed in SM and SM_T. For example, let's assume that during the execution of two sub-models, SM and SM_T, on any entry-exit event sequence $ES(S_i, Z_i, S_o) = <E_1, \ldots, E_k>$, a

sequence of transitions TS(SM, ES) = <T_1, T_2, . . .,T_k> is executed in SM and TS(SM_T, ES) = <T'_1, T'_2, . . ., T'_k> in SM_T. If the corresponding transitions T_j and T'_j that are executed on event E_j are semantically equivalent, then they must execute the same sequence of external actions and produce the same externally observable output for SM and SM_T, which guarantees that the two sub-models are semantically equivalent. Verifying the correctness of a transformation rule requires proving that upon applying the rule on any sub-model SM, a semantically equivalent sub-model, SM_T, is produced.

This verification consists of 3 major steps for the execution of SM and SM_T on ES(S_i, Z_i, S_o):

1. Entering sub-models on E_1 from entry state S_i with the following transitions:

   $SM: \quad T_1 = (E_1,\ C_1,\ A_1,\ S_i,\ S_{e1})$
   $SM_T: \ T'_1 = (E_1,\ C'_1,\ A'_1,\ S_i,\ S'_{e1})$
   Semantical equivalence between T_1 and T'_1 on E_1 needs to be established.

2. Executing the sub-models on any E_j between E_1 and E_k with the following transitions:

   $SM: \quad T_j = (E_j, C_j, A_j, S_{bj}, S_{ej})$
   $SM_T: \ T'_j = (E_j, C'_j, A'_j, S'_{bj}, S'_{ej})$
   Semantical equivalence between T_j and T'_j on E_j needs to be established.

3. Exiting the sub-models on E_k to exit state S_o with the following transitions:

   $SM: \quad T_k = (E_k,\ C_k,\ A_k,\ S_{bk},\ S_o)$
   $SM_T: \ T'_k = (E_k, C'_k, A'_k,\ S'_{bk}, S_o)$
   a. Semantical equivalence between T_k and T'_k on E_k needs to be established.
   b. It must be shown that the state-of-model-variables Z_o is the same in both sub-models in the exit state S_o.

In summary, to verify a transformation rule is correct, we follow the above steps to prove that applying the rule to any sub-model will generate a transformed sub-model that is semantically equivalent to the original sub-model. Assuming that the model has not changed outside the scope of the sub-model, the above steps prove that the entire transformed model is also semantically equivalent to the original model. Once the proof is established, the transformation rule can be adopted in the transformation process without the need to re-validate the resulting models.

In the next section, we present five model transformation rules, and for each rule, we demonstrate how the above steps can be followed to verify the rule.

4 EFSM Transformation Rules

In this section, we introduce five transformation rules that can be safely applied to EFSM models. When applied to a model, these rules keep the semantics of the model the same; however, they may change the structure of the model by changing the number of states or the number of transitions. For each of the transformation rules, we provide an algorithm, followed by a proof to show that the rule will generate a transformed model semantically equivalent to the original model.

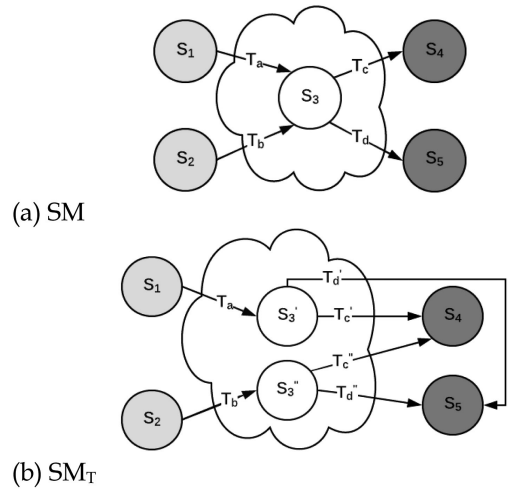


Fig. 6. State-Splitting example.

All of the presented algorithms follow the same structure. They receive as input a model M, and the main model elements that will undergo the transformation. Based on the given input, the initial sub-model SM is generated. Each algorithm defines the pre-conditions that should be satisfied in order for the transformation to take place. All the algorithms produce as an output a transformed sub-model SM_T which is generated based on the initial sub-model SM. All model elements outside the sub-models SM stay unchanged by the algorithms, while SM_T contains the elements that have been changed by the transformation. All of the algorithms follow five major steps to apply the transformation on SM and generate SM_T. In the first step, SM is generated from the provided input by calling Algorithm 1 presented in Section 3.1.1. In the second step, the set of incoming transitions to SM_T, $T_{in}(SM_T)$, is computed by visiting each transition in $T_{in}(SM)$ and applying the required changes according to the specified transformation. In the third step, the set of internal transitions, $T_r(SM_T)$, is computed from $T_r(SM)$, and in the fourth step, $T_{out}(SM_T)$ is computed based on $T_{out}(SM)$. Finally, in the last step, all the other elements of the transformed sub-model are computed from the matching elements in the initial sub-model.

4.1 State Splitting Transformation Rule

This transformation rule states that any state that has two or more incoming transitions can be split into two states, without changing the semantics of the model.

Generally, if a state S has k incoming transitions where $k = m + n$, and $m,n >= 1$, then state S can be split into two states S′ having n incoming transitions, and S" having m incoming transitions. Both states will have the same outgoing transitions as the original state, as demonstrated in Fig. 6. The two models are semantically equivalent given that each time T_a is triggered in sub-model SM, it can be followed by T_c or T_d, and similarly in sub-model SM_T, whenever T_a is triggered, it can be followed by T_c' or T_d'. The same thing applies to T_b in SM and SM_T.

Algorithm 3. State-Splitting Algorithm

Input

Model: M
State: $S \in Q(M)$
Subset of incoming transitions to S: $T_S \subset T_{in}(S)$

Pre-Condition

$|T_{in}(S)| > 1$ //at least two incoming transitions to S

Output

Transformed sub-model: $SM_T = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$

Algorithm

1 // 1. Compute submodel SM of M with respect to {S}
2 $Q_{SM} = \{S\}$
3 SM = call computeSM(M, Q_{SM})
4 // 2. Compute $T_{in}(SM_T)$
5 Set $T_{in}(SM_T)$ to empty
6 **for every** $T \in T_{in}(SM)$ **do**
7 **if** $T \in T_S$ **then** $S_e = S'$ **else** $S_e = S''$
8 $T' = (E(T), C(T), A(T), S_b(T), S_e)$
9 Add T′ to $\mathbf{T_{in}(SM_T)}$
10 **endfor**
11 // 3. Compute $T_r(SM_T)$
12 Set $T_r(SM_T)$ to empty
13 **for every** $T \in T_r(SM)$ **do**
14 $T' = (E(T), C(T), A(T), S', S')$
15 Add T′ to $\mathbf{T_r(SM_T)}$
16 $T' = (E(T), C(T), A(T), S'', S'')$
17 Add T′ to $T_r(SM_T)$
18 **endfor**
19 // 4. Compute $T_{out}(SM_T)$
20 Set $T_{out}(SM_T)$ to empty
21 **for every** $T \in T_{out}(SM)$ **do**
22 $T' = (E(T), C(T), A(T), S', S_e(T))$
23 Add T′ to $\mathbf{T_{out}(SM_T)}$
24 $T' = (E(T), C(T), A(T), S'', S_e(T))$
25 Add T′ to $T_{out}(SM_T)$
26 **endfor**
27 // 5. Compute SM_T
28 $Q = \{S', S''\}$ where $S', S'' \notin \{Q(M) \cup Q_{en}(M) \cup Q_{ex}(M)\}$
29 $R = T_r(SM_T) \cup T_{in}(SM_T) \cup T_{out}(SM_T)$
30 $SM_T = (\Sigma(SM), Q, Q_{en}(SM), Q_{ex}(SM), V(SM), O(SM), R)$

In both sub-models, when T_b is triggered, it will be followed by either T_c or T_d. Consequently, we can see that the two models are semantically equivalent. The proof is provided in 4.1.2.

4.1.1 State Splitting Algorithm

The state-splitting algorithm receives as inputs a model M, a state S to be split into two states S′ and S″, and a subset of incoming transitions to state S, denoted as Ts.

The algorithm generates as output a transformed sub-model, SM_T, having two sub-states S′ and S″ replacing state S. The transitions of T_S become incoming transitions to S′ in SM_T, and the remaining incoming transitions of S become the incoming transitions to S″ in SM_T. The algorithm starts by computing a sub-model SM of the model M where the sub-model contains only the state S to be split into two states (lines 1-3). This is done by calling Algorithm 1.

When computing $T_{in}(SM_T)$, all incoming transitions to SM that are also part of T_S are associated with S′ as a target state. All other transitions are associated with S″ as their target state (lines 5-10). In the next step of the algorithm, $T_r(SM_T)$ is computed. Since SM consists of only one state, S, all internal transitions of SM, denoted as $T_r(SM)$, are self-looping transitions, and hence they are expressed in the equivalent transformed sub-model as self-looping transitions for both S′ and S″ (lines 12-18). All outgoing transitions of the sub-model SM, are associated as outgoing transitions for both states S′ and S″ (lines 20-26). Finally, to generate SM_T, all the elements of the sub-model SM_T are generate based on SM. Q, representing the internal states of SM_T, is computed as the two states S′ and S″ replacing state S (line 28). R, the set of all transitions of SM_T is calculated as the union of the internal, incoming, and outgoing transitions (line 29). In line 30, the seven-tuple defining SM_T is computed. We can notice that this seven-tuple has five elements in common with the initial sub-model, SM, namely, the set of events, $\Sigma(SM)$, the set of entry states, $Q_{en}(SM)$, the set of exit states, $Q_{ex}(SM)$, the set of variables, V(SM), and the set of actions, O(SM).

4.1.2 Proof

In this section, we apply the three-step verification approach presented in 3.2.5 to prove that the state splitting transformation rule is semantics-preserving. To do that, we show that, on the execution of any sequence of events on any pair or entry-exit states of the sub-models SM and SM_T, semantically equivalent transitions are executed in SM and SM_T. The formal proof proceeds as follows:

Let S_i be an entry state and S_o be an exit state of SM and SM_T, where SM_T is generated as a result of applying the state splitting transformation rule (Algorithm 3).

Let $ES(S_i, Z_i, S_o) = <E_1, \ldots E_j, \ldots, E_k>$ be a sequence of events on which SM and SM_T are executed from entry state S_i to exit state S_o, where Z_i is the *state-of-model-variables* prior to executing ES at sate S_i. The execution of SM and SM_T on $ES(S_i, Z_i, S_o)$ proceeds as follows:

1 ***Entering the sub-models on E_1 from S_i with Z_i***
On event E_1, the following transitions are executed:
In SM, $T_1 = (E_1, C_1, A_1, S_i, S)$ is executed.
In SM_T, either T_1' or T_1'' is executed (based on steps 5-10 of Algorithm 3), where:

if $T_1 \in T_S$, then $T_1' = (E_1, C_1, A_1, S_i, S')$ is executed
if $T_1 \notin T_S$, then $T_1'' = (E_1, C_1, A_1, S_i, S'')$ is executed

T_1' and T_1'' are semantically equivalent to T_1 given that they have the same event, E_1, and the same sequence of actions, A_1. In addition, since the same sequence of all actions, A_1, is executed in SM and SM_T, the state-of-model-variables is the same in both sub-models.

2 ***Executing the sub-models internally on any event E_j between E_1 and E_k***
On event E_j, the following transitions are executed:

In SM, $T_j = (E_j, C_j, A_j, S, S)$ is executed.
In SM_T, either T_j' or T_j'' is executed (based on steps 12-18 of Algorithm 3), where:

if SM_T is in S′, then $T_j' = (E_j, C_j, A_j, S', S')$ is executed
if SM_T is in S″, then $T_j'' = (E_j, C_j, A_j, S'', S'')$ is executed

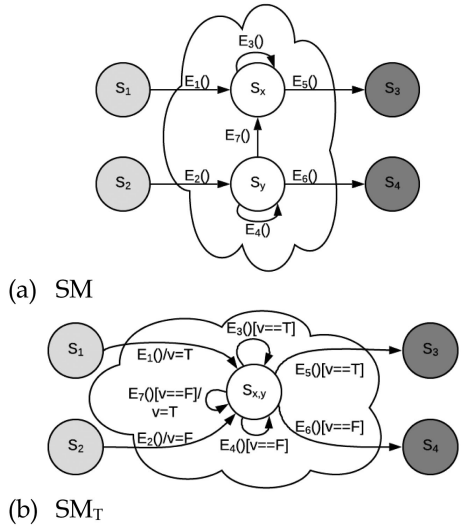


Fig. 7. State-merging example.

T_j' and T_j'' are semantically equivalent to T_j given that they have the same event $\mathbf{E_j}$ and the same sequence of actions $\mathbf{A_j}$. Consequently, the state-of-model-variables is the same in both sub-models.

*3 **Exiting the sub-models on E_k***

On event E_k, the following transitions are executed:

In SM, $T_k = (E_k, C_k, A_k, S, S_o)$ is executed.

In SM_T, either T_k' or T_k'' is executed (based on steps 20-26 of Algorithm 3), where:

if SM_T is in S', then $T_k' = (E_k, C_k, A_k, S', S_o)$ is executed

if SM_T is in S'', $T_k'' = (E_k, C_k, A_k, S'', S_o)$ is executed

T_k' and T_k'' are semantically equivalent transitions to T_k, and the state-of-model-variables Z_o is the same in both sub-models in S_o.

In conclusion, the above three steps prove that semantically equivalent transitions are executed in SM and SM_T for any sequence of events, ES, and for any pair of entry-exit states (S_i, S_o). Consequently, any sub-model, SM_T, generated by applying the state splitting rule is semantically equivalent to the original sub-model SM.

4.2 State Merging Transformation Rule

This transformation rule states that any two internal states S_x and S_y of any model, M, can be merged into one state S_{xy} while preserving the semantic of the model M.

As demonstrated in Fig. 7, this can be done by introducing a new boolean variable, v, to the model to distinguish between the two merged states. All incoming transitions to S_x and S_y become incoming transitions to S_{xy}, all outgoing transitions of S_x and S_y become outgoing for S_{xy}, and all transitions between S_x and S_y become self-looping transitions to S_{xy}. Additionally, an action, setting the value of the boolean variable, is added to each incoming transition to state S_{xy}. The value is set to False for transitions that were originally incoming transitions to state S_x in SM while it is set to True for all incoming transitions to state S_y in SM. Similarly, a condition checking the value of the boolean variable is added for each outgoing transition to state S_{xy}. The condition checks if the value is False for all transitions that were originally outgoing from state S_x, while it checks that the value is True for transitions that were originally outgoing from state S_y. Self-looping transitions in S_{xy} may have both an action and a condition added to them. The detailed algorithm is given in the next subsection.

4.2.1 State Merging Algorithm

Algorithm 4 starts by computing the sub-model, SM, having the two states to be merged; S_x and S_y (lines 1-2).

Algorithm 4. State Merging Algorithm

Input

Model: M
States: $S_x, S_y \in Q(M)$

Pre-Condition

True

Output

Transformed sub-model: $SM_T = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$

Algorithm

```
1   // 1. Compute submodel SM of M with respect to {Sx,Sy}
2   QSM = {Sx, Sy}
3   SM = call computeSM(M, QSM)
4   // 2. Compute Tin(SMT)
5   Set Tin(SMT) to empty
6   for every T∈Tin(SM) do
7     if Se(T) = Sx then A = <A(T), v = False>
8                   else A = <A(T), v = True>
9     T' = (E(T), C(T), A, Sb(T), Sxy)
10    Add T' to Tin(SMT)
11  endfor
12  // 3. Compute Tr(SMT)
13  Set Tr(SMT) to empty
14  for every T∈Tr(SM) do
15    A = A(T)
16    if Sb(T) = Sx then
17      C = [(v == False) and C(T)]
18      if Se(T) = Sy then
19        A = <A, v = True>
20    else
21      C = [(v == True) and C(T)]
22      if Se(T) = Sx then
23        A = <A, v = False>
24    endif
25    T' = (E(T), C, A, Sxy, Sxy)
26    Add T' to Tr(SMT)
27  endfor
28  // 4. Compute Tout(SMT)
29  Set Tout(SMT) to empty
30  for every T∈Tout(SM) do
31    if Sb(T) = Sx then C = [(v == False) and C(T)]
32                  else C = [(v == True) and C(T)]
33    T' = (E(T), C, A(T), Sxy, Se(T))
34    Add T' to Tout(SMT)
35  endfor
36  // 5. Compute SMT
37  Q = {Sxy} where Sxy ∉ {Q(M) ∪ Qen(M) Qex(M)}
38  V = V(SM)∪{v} where v ∉ V(M)
39  O = O(SM)∪{v = False, v = True}
40  R = Tr(SMT) ∪ Tin(SMT) ∪ Tout(SMT)
41  SMT = (Σ(SM), Q, Qen(SM), Qex(SM), V, O, R)
```

The algorithm then calculates for SM_T the three sets of transitions $T_{in}(SM_T)$, $T_r(SM_T)$, and T_{out} (SM_T) in lines 5-33. For all incoming transitions in SM, the value of the variable v is set to *True* when the destination state of the transition is S_y; otherwise, it is set to *False* (lines 7-8). Transitions between the two states S_x and S_y, as well as self-looping transitions in S_x and S_y, are added as internal transitions in the transformed sub-model after adding an action and/or a condition to preserve the semantic of the original sub-model (lines 13-27). For all outgoing transitions of the sub-model SM, the algorithm associates a condition to guarantee that the transition will be triggered only when it is preceded by an incoming transition of the matching state. So, if the state of origin of an outgoing transition is S_y, the condition added to the transition would be [v == *True*]; otherwise, the condition would be [v == *False*] (lines 29-35).

Finally, the algorithm computes the seven-tuple that defines SM_T by computing its different elements (lines 37-41). It sets the value of Q, the internal states of SM_T, to the single merged state S_{xy} (line 37). It computes V, the set of variables in SM_T, by adding the variable v to V(SM) (line 38). Similarly, it sets the actions of SM_T to the set of actions in SM in addition to two actions representing setting the value of the variable v to *True* or *False* (line 39). R, the set of all transitions in SM_T, is computed as the union of T_{in}, T_r, and T_{out}, which were computed earlier in the algorithm (line 40). In line 41, the seven-tuple defining SM_T is computed based on the different elements which were computed previously in the algorithm. The three elements that have not changed in both SM and SM_T are Σ(SM), Q_{en}(SM), and Q_{ex}(SM).

4.2.2 Proof

Let S_i be an entry state and S_o be an exit state of SM and SM_T, where SM_T is generated as a result of applying the state merging transformation rule (Algorithm 4).

Let $ES(S_i, Z_i, S_o) = <E_1, \ldots E_j, \ldots, E_k>$ be a sequence of events on which SM and SM_T are executed from entry state S_i to exit state S_o, where Z_i is the *state-of-model-variables* prior to executing ES at sate S_i. The execution of SM and SM_T on $ES(S_i, Z_i, S_o)$ proceeds as follows:

1 Entering the sub-models from S_i with Z_i.

On event E_1, the following transitions are executed:

a. Assuming that the target state of T_1 in SM is S_x
SM: $T_1 = (E_1, C_1, A_1, S_i, S_x)$, where SM is now in state S_x.

SM_T: $T_1' = (E_1, C_1, <A_1, v = False>, S_i, S_{xy})$, SM_T is in state S_{xy} with the value of v = *False*, where v is an internal variable of SM_T.
or

b. Assuming that the target state of T_1 in SM is S_y
SM: $T_1 = (E_1, C_1, A1, S_i, Sy)$, where SM is now in state S_y.

SM_T: $T_1'' = (E_1, C_1, <A_1, v = True>, S_i, S_{xy})$, SM_T is in state S_{xy} with value of v = *True*.

In both cases, T_1' and T_1'' are semantically equivalent transitions to T_1 since the same sequence of actions $\mathbf{A_1}$ is executed in SM and SM_T, and assignment statements to v are not external actions. In addition, the state-of model-variables in both sub-models are the same, except the value of the internal variable in SM_T. This semantic equivalence is guaranteed by steps 5-11 in the transformation.

2 Executing the sub-models internally on any event E_j between E_1 and E_k.

On E_j, the following transitions are executed:

a. SM is in state S_x and SM_T is in state S_{xy} with v = *False*:

SM: $T_j = (E_j, C_j, A_j, S_x, S_x)$
SM_T: $T_j' = (E_j, [v == False \text{ and } C_j], A_j, S_{xy}, S_{xy})$
$= (E_j, C_j, A_j, S_{xy}, S_{xy})$
or
SM:
$T_j = (E_j, C_j, A_j, S_x, S_y)$, where SM is now in state S_y.
SM_T:
$T_j'' = (E_j, [v == False \text{ and } C_j], <A_j, v = True>, S_{xy}, S_{xy})$
$= (E_j, C_j, <A_j, v = True>, S_{xy}, S_{xy})$, where the value *True* is now assigned to *v*.

b. SM is in state S_y and SM_T is in state S_{xy} with v = *True*:

SM: $T_j = (E_j, C_j, A_j, S_y, S_y)$
SM_T: $T_j' = (E_j, [v == True \text{ and } C_j], A_j, S_{xy}, S_{xy})$
$= (E_j, C_j, A_j, S_{xy}, S_{xy})$
or
SM: $T_j = (E_j, C_j, A_j, S_y, S_x)$, where SM is now in state S_x.
SM_T:
$T_j'' = (E_j, [v == True \text{ and } C_j], <A_j, v = False>, S_{xy}, S_{xy})$
$= (E_j, C_j, <A_j, v = False>, S_{xy}, S_{xy})$, where the value *False* is now assigned to *v*.

In all cases, T_j' and T_j'' are semantically equivalent transitions to T_j since the same sequence of actions $\mathbf{A_j}$ is executed in SM and SM_T, and assignment statements to v are not external actions. In addition, the state-of model-variables in both sub-models is the same, except the value of internal variable v in SM_T. This semantic equivalence is guaranteed by steps 13-25 in the transformation algorithm.

3 Exiting the sub-models on E_k.

The following transitions are executed:

a. SM is in state S_x, and SM_T is in state S_{xy} with v = *False*:

SM: $T_k = (E_k, C_k, A_k, S_x, S_o)$
SM_T: $T_k' = (E_k, [v == False \text{ and } C_k], A_k, S_{xy}, S_o)$
$= (E_k, C_k, A_k, S_{xy}, S_o)$

b. SM is in state S_y, and SM_T is in state S_{xy} with v = *True*:

SM: $T_k = (E_k, C_k, A_k, S_y, S_o)$
SM_T: $T_k'' = (E_k, [v == True \text{ and } C_k], A_k, S_{xy}, S_o)$
$= (E_k, C_k, A_k, S_{xy}, S_o)$

In both cases, T_k' and T_k'' are semantically equivalent transitions to T_k, and the state-of-model-variables Z_o is the same in both sub-models in S_o. This semantic equivalence is guaranteed by steps 27-33 in the transformation.

In conclusion, we have shown that SM and SM_T are semantically equivalent sub-models with respect to any pair (S_i, S_o) of entry and exit states. Therefore, the state merging transformation is correct for any model and its states.

4.3 Transition Merging Transformation Rule

This transformation rule states that two transitions T_x, T_y having two different enabling conditions can be merged into one transition if they have the same event, sequence of actions, origin state, and destination state. Merging the two transitions can be done by combining the two different conditions of the transitions into one condition with an OR operator, as demonstrated in Fig. 8.

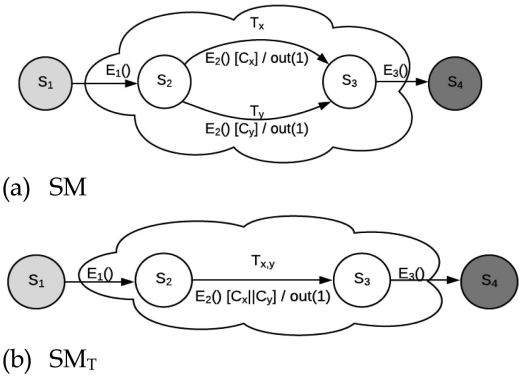


Fig. 8. Transition-merging example.

Algorithm 5. Transition Merging Algorithm

Input

Model: M
Transitions: T_x, $T_y \in R(M)$

Precondition

$T_x = (E, C_x, A, S_b, S_e)$
$T_y = (E, C_y, A, S_b, S_e)$

Output

Transformed sub-model: $SM_T = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$

Algorithm

```
1   // 1. Compute submodel SM of M with respect to {Sb,Se}
2   QSM = {Sb, Se}
3   SM = call computeSM(M, QSM)
4   // 2. Compute Tin(SMT)
5   Tin(SMT) = Tin(SM)
6   // 3. Compute Tr(SMT)
7   Set Tr(SMT) to empty
8   for every T∈Tr(SM) do
9     T′ = T
10    if T = Tx or T = Ty then
11      Cxy = [Cx || Cy]
12      T′ = (E, Cxy, A, Sb, Se)
13    Add T′ to Tr(SMT)
14  endfor
15  // 4. Compute Tout(SM)
16  Tout(SMT) = Tout(SM)
17  // 5. Compute SMT
18  R = Tr(SMT) ∪ Tin(SMT) ∪ Tout(SMT)
19  SMT = (Σ(SM), Q(SM), Qen(SM), Qex(SM), V(SM), O(SM),
    R)
```

4.3.1 Transition Merging Algorithm

Algorithm 5 is self-explanatory. According to the pre-condition, all the elements of the two transitions must be identical except for the condition in order for the transformation to take place. The algorithm starts by calculating the sub-model, SM, that has the source and target states of the transitions to be merged (lines 2-3).

Incoming transitions to SM remain unchanged in SM_T (line 5). $Tr(SM_T)$ is generated in lines 7-14. Initially, for each internal transition T in SM, an equivalent transition T′ is generated (line 9). When the transition under consideration is Tx or Ty, then the condition of T′ is reset by joining both conditions of T_x and T_y with an OR operator (line 11). The outgoing transitions of SM remain unchanged in SM_T (line 16). Finally, R is calculated in line 18, and the seven-tuple defining SM_T is constructed in line 19, where all the elements are identical to SM except for R.

4.3.2 Proof

Let S_i be an entry state and S_o be an exit state of SM and SM_T, where SM_T is generated as a result of applying the transition merging transformation rule (Algorithm 5).

Let $ES(S_i, Z_i, S_o) = <E_1, \ldots E_j, \ldots, E_k>$ be a sequence of events on which SM and SM_T are executed from entry state S_i to exit state S_o, where Z_i is the *state-of-model-variables* prior to executing ES at sate S_i.

The execution of SM and SM_T on $ES(S_i, Z_i, S_o)$ proceeds as follows:

1 ***Entering the sub-models from S_i with Z_i***
On event E_1, the following transitions are executed:

SM: $T = (E_1, C_1, A_1, S_i, S_b)$
SM_T: $T' = (E_1, C_1, A_1, S_i, S_b)$ based on step 5 of the algorithm.

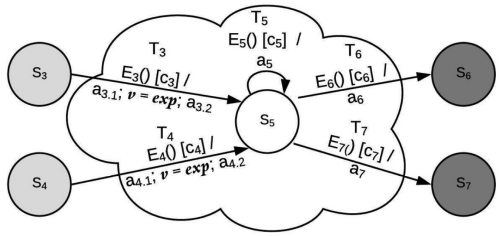
2 ***Executing the sub-models internally on any event E_j between E_1 and E_k.***
On event E_j, the following transitions are executed:

SM: $T = (E_j, C_j, A_j, S_1, S_2)$ where $T \neq T_x$ and $T \neq T_y$ and $S_1, S_2 \in \{S_b, S_e\}$
SM_T: $T' = (E_j, C_j, A_j, S_1, S_2)$ based on step 9 of the algorithm.
or
SM: $T = T_x = (E_j, C_x, A_j, S_b, S_e)$ or $T = T_y = (E_j, C_y, A_j, S_b, S_e)$
SM_T: $T' = T_{xy} = (E_j, [C_x \,||\, C_y], A_j, S_b, S_e)$ based on steps 9-13 of the transformation.

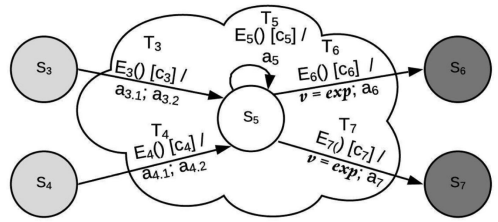
3 ***Exiting the sub-models on E_k.***
On event E_k, the following transitions are executed:

SM: $T = (E_k, C_k, A_k, S_e, S_o)$
SM_T: $T' = (E_k, C_k, A_k, S_e, S_o)$ based on step 16 of the algorithm.

In all cases, T and T′ are semantically equivalent transitions, and the state-of-model-variables is the same in both sub-models after the execution of these transitions. In conclusion, we have shown that SM and SM_T are semantically equivalent sub-models with respect to any pair (S_i, S_o) of entry and exit states. Therefore, the transition merging transformation is correct for any model and its state.



(a) SM



(b) SM_T

Fig. 9. Example of moving assignment (v=exp) forward.

4.4 Moving Assignment Action Forward

For a given state S in a model M, an assignment action that is defined in all incoming transitions can be moved to all outgoing transitions if certain conditions are met.

In its simplest form, this rule states that for a given state S, if all incoming transitions define the same action a ($v = expr$) as the last action in their sequence of actions, then it is possible to move the action a to be the first action in the sequence of actions associated with the outgoing transitions of the state S. This rule assumes that none of the outgoing transitions use the variable v in their enabling condition, and it also assumes that if self-looping transitions are present at state S, then they do not use or change variable v or any other variable used in the expression *expr*. Of course, in a more general context, the action to be moved forward with respect to a state could be anywhere in the middle of the sequence of actions.

4.4.1 Algorithm to Move Assignment Actions Forward

The algorithm lists two pre-conditions that should be satisfied in order for the transformation to take place. The first pre-condition states that the assignment action should be a definition of a variable, v, that belongs to the set of model variables, V (M). Additionally, it states that any other variable used in the action a should also belong to V(M). This condition excludes variables that are parameters of events. Events' parameters are not part of the model variables since their scope is limited to the transition itself, so they can be referenced only while the transition is being triggered. Consequently, expressions using these parameters cannot be moved to other transitions. The second precondition ensures that: the assignment action is defined in all incoming transitions, the variable v is not referenced along the different paths to any outgoing transition of state S, and variables used in its definition (if any) are not modified along the different paths as well.

For example, in Fig. 9a, the action ($v = expr$) is defined on both incoming transitions of state S_5, namely T_3 and T_4. To check if the first pre-condition is satisfied, x and all variables used in *expr* should be model variables, so not parameters of E_3 or E_4. As for the second pre-condition, we need to check its three parts. The first one is satisfied since the action ($v = expr$) is defined for all incoming transitions of S_5. To satisfy the second part (*ii*) of the pre-condition, the variable v should not be referenced by $a_{3.2}$, $a_{4.2}$, a_5, C_5, C_6, or C_7. Finally, to satisfy the third part (*iii*) of the pre-condition, any variable used in *expr* should not be modified by $a_{3.2}$, $a_{4.2}$, or a_5. Assuming that all preconditions are satisfied, the action can be moved forward to the outgoing transitions T_6 and T_7. The algorithm starts by computing the sub-model having S_5 as its internal state (lines 2-3). Then in lines 5-11, the algorithm removes the action for all incoming transitions, and in lines 15-20, the algorithm adds the action to all outgoing transitions. The internal transitions in SM remain unchanged (line 13). Finally, in line 23 the algorithm generates the transformed sub-model SM_T, which has the same model elements of SM except for R.

Algorithm 6. Moving Assignment Action Forward

Input

Model: M
State: $S \in Q(M)$
Action: $a \in O(M)$

Precondition

1) a is a definition of variable $v \in V(M)$ and a use of model variables, $U(a) \subseteq V(M)$
2) $\forall T \in T_{in}(S)$:
i. a is an action in $A(T)$,
ii. v is not referenced between a and the first action of any outgoing transition of S
iii $\forall w \in U(a)$: w is not modified between a and the first action of any outgoing transition of S

Output

Transformed sub-model: $SM_T = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$

Algorithm

```
1  // 1. Compute submodel SM of M with respect to {S}
2  Q_SM = {S}
3  SM = call computeSM(M, Q_SM)
4  // 2. Compute T_in(SM_T)
5  Set T_in(SM_T) to empty
6  for every T∈T_in(SM) do
7     A = A(T)
8     Remove a from A
9     T' = (E(T), C(T), A, S_b(T), S)
10    Add T' to T_in(SM_T)
11 endfor
12 // 3. Compute Tr(SM_T)
13 T_r(SM_T) = T_r(SM)
14 // 4. Compute T_out(SM_T)
15 Set T_out(SM_T) to empty
16 for every T∈T_out(SM) do
17    A = <a, A(T)>
18    T' = (E(T), C(T), A, S, S_e(T))
19    Add T' to T_out(SM_T)
20 endfor
21 // 5. Compute SM_T
22 R = T_r(SM_T) ∪ T_in(SM_T) ∪ T_out(SM_T)
23 SM_T = (Σ(SM), Q(SM), Q_en(SM), Q_ex(SM), V(SM), O(SM), R)
```

4.4.2 Proof

Let S_i be an entry state and S_o be an exit state of SM and SM_T, where SM_T is generated as a result of moving assignment actions forward following Algorithm 6.

Let ES(S_i, Z_i, S_o) = <$E_1, \ldots E_j, \ldots, E_k$> be a sequence of events on which SM and SM_T are executed from entry state S_i to exit state S_o, where Z_i is the *state-of-model-variables* prior to executing ES at sate S_i. The execution of SM and SM_T on ES(S_i, Z_i, S_o) proceeds as follows:

1 Entering the sub-models from S_i with Z_i.

On event E_1, the following transitions are executed:

SM: $T_1 = (E_1, C_1, A_1, S_i, S)$,
where $A_1 = < A_F, a, A_L >$. A_F and A_L are action sub-sequences of A_1. Action *a* is a definition of model variable $v = expr$ and is not an external action. Let us assume that before the execution of *a*, i.e., after execution A_F, the value of $v = v_b$ and after the execution of *a* value of v is v_a. Based on the precondition (2), all actions in A_L do not reference nor modify v and any variable used by *a*, i.e., variables of U(a). Therefore, the value of v assigned at *a* is not changed and is equal to v_a in state S. In addition, all variables of U(a) have the same values before and after the execution of A_L.

SM_T: $T'_1 = (E_1, C_1, A'_1, S_i, S)$,
where $A'_1 = < A_F, A_L >$. Since the value of v after the execution of A_F is equal to v_b and all actions in A_L do not reference nor modify v and any variable of U(a), the value of v equals to v_b in state S, and all variables of U(a) have the same values before and after the execution of A_L.

T_1' and T_1 are semantically equivalent transitions since the same sequences of external actions in $\mathbf{A_F}$ and $\mathbf{A_L}$ are executed in SM and SM_T producing the same externally observable output. In addition, the state-of model-variables in state S is the same in both sub-models, except the value of variable v, i.e., $v = v_a$ in SM and $v = v_b$ in SM_T. This semantic equivalence is guaranteed by steps 5-11 in the algorithm.

2 Executing the sub-models internally on any event E_j between E_1 and E_k

The following transitions are executed on E_j:

SM: $T_j = (E_j, C_j, A_j, S, S)$
SM_T: $T'_j = (E_j, C_j, A_j, S, S)$

Clearly, T_j' and T_j are semantically equivalent transitions. This equivalence is guaranteed by step 13 in the algorithm. In addition, based on Precondition (2), variables of U(a) and v are not used nor modified in C_j and A_j. Therefore, their values are the same before and after the execution of T_j and T'_j. Consequently, the state-of model-variables in S is the same in both sub-models, except the value of v ($v = v_a$ in SM and $v = v_b$ in SM_T).

3 Exiting the sub-models on E_k.

On E_k, the following transitions are executed:

SM: $T_k = (E_k, C_k, A_k, S, S_o)$
where the value of v before execution of A_k is equal to v_a.

SM_T: $T'_k = (E_k, C_k, A'_k, S, S_o)$
where $A'_k = <\mathbf{a}, \mathbf{A_k}>$. Note: C_k also evaluates to *true* in T'_k because it does not use variable v. Before execution of *a* in A'_k, value of $v = v_b$ and all variables in U(a) have the same values as before execution of *a* in $A_1 = <A_F, a, A_L>$ of the incoming transition to SM. As a result, *a* assigns value v_a to v.

T_k' and T_k are semantically equivalent transitions since the state-of model-variables is the same before the execution of $\mathbf{A_k}$ in SM and SM_T producing the same externally observable output, and the state-of model-variables in S_o is the same in both sub-models. This semantic equivalence is guaranteed by steps 15-20 in the algorithm.

In conclusion, we have shown that SM and SM_T are semantically equivalent sub-models with respect to any pair (S_i, S_o) of entry and exit states. Therefore, the transformation rule is correct for any given model.

4.5 Moving Assignment Action Backward

This transformation is the opposite of the previous transformation. The rule attempts to move an action defining a variable v backward in the model. For a given state S in an EFSM model M, an assignment action defined in all of the outgoing transitions can be moved to all incoming transitions of the same state if certain conditions are met.

Algorithm 7. Moving Assignment Action Backward

Input

Model: M
State: S ∈ Q(M)
Action: *a* ∈ O(M)

Precondition

1) *a* is a definition of variable v∈V(M) and a use of model variables, U(*a*) ⊆ V(M)
2) ∀ T∈T_{out}(S):
i. a is in the action sequence A(T),
ii. v is not referenced between the last action of any incoming transition to *S* and *a* in A(T),
iii. ∀ *w*∈U(a): *w* is not modified between the last action of any incoming transition to *S* and *a*.

Output

Transformed sub-model: $SM_T = (\Sigma, Q, Q_{en}, Q_{ex}, V, O, R)$

Algorithm

1 // 1. Compute submodel SM of M with respect to {S}
2 Q_{SM} = {S}
3 SM = call computeSM(M, Q_{SM})
4 // 2. Compute $T_{in}(SM_T)$
5 Set $T_{in}(SM_T)$ to empty
6 **for every** T∈T_{in}(SM) **do**
7 A = <A(T), *a* >
8 T′ = (E(T), C(T), A, S_b(T), S)
9 Add T′ to $T_{in}(SM_T)$
10 **endfor**
11 // 3. Compute $T_r(SM_T)$
12 $T_r(SM_T) = T_r(SM)$
13 // 4. Compute $T_{out}(SM_T)$
14 Set $T_{out}(SM_T)$ to empty
15 **for every** T∈T_{out}(SM) **do**
16 A = A(T)
17 Remove *a* from A
18 T′ = (E(T), C(T), A, S, S_e(T))
19 Add T′ $T_{out}(SM_T)$
20 **endfor**
21 // 5. Compute SM_T
22 $R = T_r(SM_T) \cup T_{in}(SM_T) \cup T_{out}(SM_T)$
23 $SM_T = (\Sigma(SM), Q(SM), Q_{en}(SM), Q_{ex}(SM), V(SM), O(SM), R)$

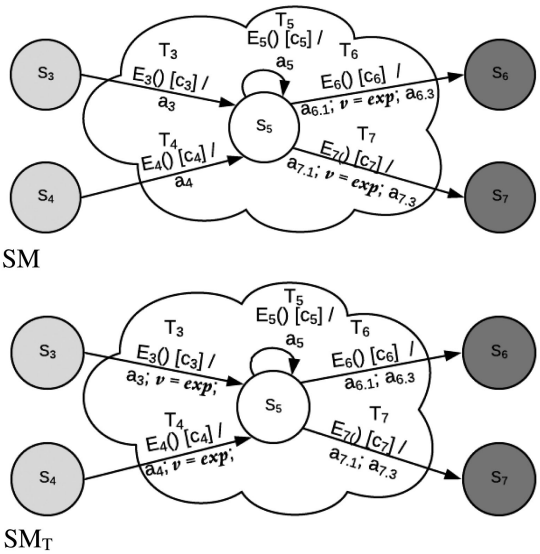


Fig. 10. Moving assignment action (v=exp) backward.

In its simplest form, this rule states that for a given state S, if all outgoing transitions define the same action a ($v = expr$) as the first action in their sequence of actions, then the action can be moved backward to be placed as the last action of all incoming transitions of the same state.

Of course, in a more general context, the state S may have self-looping transitions, the outgoing transitions may have enabling conditions, and the action defining the variable v could be anywhere in the middle of the sequence of actions.

4.5.1 Algorithm to Move Assignment Actions Backward

Similar to the previous algorithm, this algorithm has two pre-conditions as well. The first one guarantees that no event parameters are used in the action to be moved backward. The second pre-condition ensures three main things. First, it ensures that the action to be moved backward is defined by all outgoing transitions of the state. Second, it ensures that the variable v defined in the action is not referenced "backward". So if the action is not the first in a sequence of actions defined by an outgoing transition T, all actions preceding the action a in T should not reference v. Similarly, the enabling conditions of the outgoing transitions and all self-looping transitions should not reference v. The last part of the condition (item *iii*), ensures that the variables used in the expression of the action a (referred to as U(a)) are not modified "backward" up until the last action of any incoming transition.

For example, in Fig. 10a, we recognize that the action ($v = exp$) is defined by both outgoing transitions of the state S_5. This makes the action a good candidate to be moved backward if all pre-conditions are satisfied. The first pre-condition can be satisfied if the variable v and all other variables used in expr are model variables, so they are not parameters of the E_6 or E_7. The second pre-condition can be satisfied if all of its three parts are satisfied. Pre-condition (*i*) is already satisfied since the action is defined in all outgoing transitions of the state S_5. Pre-condition (*ii*) can be satisfied if the variable v is not referenced by $a_{6.1}$, $a_{7.1}$, a_5, C_5, C_6, or C_7. Finally, pre-condition (*iii*) can be satisfied if all variables used in *expr* are not modified by $a_{6.1}$, $a_{7.1}$, or a_5.

4.5.2 Proof

Let S_i be an entry state and S_o be an exit state of SM and SM_T, where SM_T is generated as a result of moving assignment actions backward following Algorithm 7.

Let $ES(S_i, Z_i, S_o) = <E_1, \ldots E_j, \ldots, E_k>$ be a sequence of events on which SM and SM_T are executed from entry state S_i to exit state S_o, where Z_i is the *state-of-model-variables* prior to executing ES at sate S_i.

The execution of SM and SM_T on $ES(S_i, Z_i, S_o)$ proceeds as follows:

1 ***Entering the sub-models from S_i with Z_i***

On event E_1, the following transitions are executed:

SM: $T_1 = (E_1, C_1, A_1, S_i, S)$,

Let's assume that after execution of A_1, the value of $v = v_b$.

SM_T: $T'_1 = (E_1, C_1, A'_1, S_i, S)$,

where $A'_1 = <\mathbf{A_1}, \mathbf{a}>$. Action a is a definition of model variable $v = expr$ and is not an external action. The value of v before execution of a, i.e., after the execution of A_1, is equal to v_b. Let's assume that after the execution of a, the value of v is v_a.

T_1' and T_1 are semantically equivalent transitions since the same sequence of external actions of $\mathbf{A_1}$ and $\mathbf{A'_1}$ is executed in SM and SM_T producing the same externally observable output. In addition, the state-of model-variables in state S is the same in both sub-models, except the value of variable v ($v = v_b$ in SM and $v = v_a$ in SM_T). This semantic equivalence is guaranteed by steps 5-10 in the algorithm.

2 ***Executing the sub-models internally on any event E_j between E_1 and E_k***

The following transitions are executed on E_j:

SM: $T_j = (E_j, C_j, A_j, S, S)$

SM_T: $T'_j = (E_j, C_j, A_j, S, S)$

Clearly, T_j' and T_j are semantically equivalent transitions. This semantic equivalence is guaranteed by step 12 in the algorithm. In addition, based on Precondition (2), v and variables of U(a) are not used nor modified in C_j and A_j. Therefore, their values are the same before and after the execution of T_j and T'_j. Consequently, the state-of model-variables in S is the same in both sub-models, except the value of v ($v = v_b$ in SM and $v = v_a$ in SM_T).

3 ***Exiting the sub-models on E_k***

On E_k, the following transitions are executed:

SM: $T_k = (E_k, C_k, A_k, S, S_o)$

where $A_k = <\mathbf{A_F}, a, \mathbf{A_L}>$ and value of v before execution of A_k is equal to v_b. Based on the pre-condition (2), all actions in A_F do not reference nor modify v and any variable of U(a). Before the execution of a, the value of $v = v_b$, and all variables in U(a) have the same values as before execution of a in $A'_1 = <A_1, a>$ of the incoming transition T'_1 to SM_T. As a result, before A_L is executed, a assigns the value v_a to v.

SM_T: $T'_k = (E_k, C_k, A'_k, S, S_o)$

where $A'_k = <\mathbf{A_F}, \mathbf{A_L}>$. Note: C_k also evaluates to *true* in T'_k because it does not use the variable v. Before the execution of A'_k the value of $v = v_a$, and this value is not changed in A_F before A_L is executed.

T_k' and T_k are semantically equivalent transitions since the same sequence of external actions of A_k and A'_k is executed producing the same externally observable output, and the state-of model-variables in S_o is the same in both sub-models. This semantic equivalence is guaranteed by steps 14-20 in the transformation.

In conclusion, we have shown that SM and SM_T are semantically equivalent sub-models with respect to any pair (S_i, S_o) of entry and exit states. Therefore, the transformation rule is correct for any given model.

5. Model Transformation Automation

The transformation process presented in this paper and depicted in Fig. 3 is supported by a tool developed in Java in order to ensure the automation of the process. The tool accepts as input the text-based specifications of an EFSM model, and it interactively suggests to an expert modeler a list of possible transformations that can be applied to the model. The tool applies the transformations as selected by the expert, and generates the transformed model. It is worth mentioning that the interaction with an expert can be replaced with an automated search algorithm guided by a fitness function to enhance certain quality metrics of the model (size, complexity, etc.) as presented in [13].

Currently, the tool applies only the five transformation rules which have been verified through the theoretical proofs presented in (4.1.2, 4.2.2, 4.3.2, 4.4.2 & 4.5.2). Limiting the tool to verified transformation rules guarantees generating output models that are semantically equivalent to the input models. In order to integrate a new transformation rule into the tool, a developer needs first to verify it manually by following the approach provided in 3.2.5.

Obviously, the automation of the verification approach of the transformation rules is not as crucial as automating the transformation process itself. Indeed, the theoretical proof of the correctness of the rule needs to be established only once for a new rule, and the rule can then be safely applied to an unlimited number of models without the need for further verifications. There exist, however, many techniques that support the automation of model transformations [35], [36], [37], [38], [39], [40], [41], and such techniques can be used to automate the verification of the transformation rules.

In the remainder of this section, we support the automation of our proposed rule verification approach by presenting an automated approach for the partial verification of new rules.

Typically, to automate model validation, transformed model properties are identified, and automated methods are used to verify these properties. For our rule verification approach, properties related to the equivalence of transitions in the original sub-model and the transformed one can be identified. For example, for any transition in the original sub-model SM there should exist the corresponding transition in the transformed sub-model SM_T for which the sequence of external actions in both transitions is identical. We refer to this property as a *partial semantic equivalence of transitions*. This property can be defined separately for the incoming transitions, internal transitions, and outgoing transitions. These properties are formally defined below:

Property #1: Incoming transitions

$$\forall\ T \in T_{in}(SM) \exists\ T' \in T_{in}(SM_T)\ \textit{where}$$
$$S_b(T) = S_b(T')\ \textit{and}\ E(T) = E(T')\ \textit{and}\ A_E(T) = A_E(T')$$

Property #2: Internal transitions

$$\forall\ T \in T_r(SM) \exists\ T' \in T_r(SM_T)\ \textit{where}$$
$$E(T) = E(T')\ \textit{and}\ A_E(T) = A_E(T')$$

Property #3: Outgoing transitions

$$\forall\ T \in T_{out}(SM) \exists\ T' \in T_{out}(SM_T)\ \textit{where}$$
$$E(T) = E(T')\ \textit{and}\ A_E(T) = A_E(T')\ \textit{and}\ S_e(T) = S_e(T')$$

It is expected that any model transformation rule satisfies these properties. Automated methods can be developed to verify whether these properties are satisfied by any new model transformation rule. Automated symbolic execution can be used to verify these properties, as shown in algorithm 8. In this algorithm for all paths in the for-loops that compute the corresponding incoming, internal and outgoing transitions for the transformed sub-model, the properties 1-3 are checked for any violations.

Algorithm 8. Property Verification Algorithm

Input

Model Transformation algorithm

Output

True: All properties are satisfied
False: Properties are not satisfied

Property Verification Algorithm

```
1   Boolean verified = true
2   for every path P inside of a for-loop computing T_in(SM_T)
    or T_r(SM_T) or T_out(SM_T) do
3   symbolically execute path P to compute T'
4   At "Add T' to T_in(SM_T)" operation check:
5   if E(T)≠E(T') or A_E(T) ≠A_E(T') or S_b(T) ≠S_b(T') then
6   verified = false
7   At "Add T' to T_r(SM_T)" operation check:
8   if E(T) ≠E (T') or A_E(T) ≠A_E(T') then
9   verified = false
10  At "Add T' to T_out(SM_T)" operation check:
11  if E(T)≠E(T') or A_E(T) ≠A_E(T') or S_e(T) ≠S_e(T') then
12  verified = false
13  endfor
14  output(verified)
```

For example, when this algorithm is used for the state merging transformation rule (Algorithm 4) to compute internal transitions, four paths in the for-loop (14-26) are identified and symbolically executed:

P1: 14, 15, 16, 17, 18, 25, 26
P2: 14, 15, 16, 17, 18, 19, 25, 26
P3: 14, 15, 16, 21, 22, 25, 26
P4: 14, 15, 16, 21, 22, 23, 25, 26

Sample symbolic execution for path P2, which can be performed by the automated symbolic executor, is as follows:

14 $T \in T_r(SM)$
15 $A = A(T)$
16 $(S_b(T) = S_x)$ **True**
17 $C = [(v == \text{False}) \text{ and } C(T)]$
18 $(S_e(T) = S_y)$ **True**
19 $A = <A(T), v = \text{True}>$
25 $T' = (E(T), [(v == \text{False}) \text{ and } C(T)], <A(T), v = \text{True}>, S_{xy}, S_{xy})$
26 Add T′ to $T_r(SM_T)$

Clearly, Property #2 is satisfied for T and T′ at line 8 of the algorithm since:

$E(T) = E(T')$, and
$A_E(T) = A_E(T')$ given that $A(T') = <A(T)$,
$v=\text{True}>$and "v=True" is not an external action.

The automated symbolic execution can be used to show that Property #2 is also satisfied for all the remaining paths. Similarly, all three properties can be checked for any violations.

Clearly, algorithm 8 provides an automated approach for the partial verification of the rules, but it doesn't prove their correctness similar to following the three-step approach presented in Section 3.2.5. The algorithm can, however, be used to increase the developer's confidence in new transformation rules.

6 Case Studies

Our goal behind presenting the case studies in this section is to give the reader a higher confidence in the proposed transformation approach beyond the provided theoretical proofs. We set two objectives under this goal. The first one is to show through testing that applying a sequence of transformations using individually proven transformation rules does not change the overall behavior of the model. The second objective is to show that the presented transformation rules can be used to enhance certain model characteristics in order to enhance the overall quality of the model.

The two models that we consider in this section are the Fuel Pump model [13] and the Elevator Door Control model [29]. For each model, we apply a sequence of transformations targeting enhancing a pre-defined model metric. We also test each model before and after applying the transformations using a test suite composed of five test cases that satisfy the transition coverage criterion.

To address our first objective, we check that all of the five test cases give the same observable output before and after applying the transformations. To address the second objective, we measure the targeted model metric before and after the transformations to show the enhancement.

For the first case study, we apply a sequence of transformations targeting enhancing the model's average change impact (6.1.2), while for the second case study we apply a sequence of transformations targeting reducing the size of the model without increasing its complexity.

6.1 EFSM Model Metrics

Identifying model metrics helps in quantifying the quality of the models, and provides the basis on which the system modeler can select what transformations to apply and in what order. Model transformations may target reducing the size of the model, reducing its complexity, enhancing its readability, or enhancing its maintainability, to list only a few. In the following sub-sections, we describe three model metrics that we use in the two case studies to judge the improvement of the model quality.

6.1.1 Model Size

The size of an EFSM model can be expressed in terms of the number of states or the number of transitions, as these are the two main components of the EFSM. The states are the passive components of the model, while the transitions are the active components. Some models may have a high number of states, which will naturally lead to a high number of transitions since each state will have at least one incoming transition and one outgoing transition. In this paper, we measure the model size using two different metrics: the number of states and the number of transitions.

6.1.2 Model Average Change Impact

Impact analysis is the process of identifying the expected impact of a potential change applied to the model. An approach to identify the impact of a given change applied on EFSM models is proposed in [5], where for any given model modification (MF), two impact sets are identified; the starting impact set (SIS), which consists of the set of transitions directly impacted by the change, and the extended impact set (EIS) which consists of indirectly impacted transitions. In order to quantify the impact of a given change (denoted as MF), the number of transitions in the EIS can be used as a measure.

For any given model, it is possible to estimate the *average change impact* of *any potential change* applied to the model as a measure of maintainability. This measure gives a better understanding of the development team about how easy or difficult the maintenance of the model is. For example, if a model that consists of 100 transitions has an average change impact of 60 transitions, this indicates that on average, when a change is applied to the model, the developer should expect 60 transitions to be impacted by the change (so 60% of the model). Of course, the smaller this number is, the easier it is to maintain the model. The calculations of the model's average change impact is demonstrated in [13].

6.1.3 Transition Density

The model's *transition density* was first defined in [5] as the average number of transitions per state (#transitions/# states). This attribute helps approximating the complexity of the model. It can also be used in combination with the model size to ensure that reducing the model size doesn't increase its complexity.

6.2 Fuel Pump Model

According to the fuel pump model depicted in Fig. 11, when the pump is activated, the prices for regular fuel and super fuel are initialized. A customer using the pump has the choice to pay by credit or cash. If credit payment is chosen, the credit card is validated. Upon approval of the payment, the user will indicate his/her choice of fuel, and accordingly, the price to be paid by the user is initialized. When the person starts pumping fuel, the amount of gas pumped in is tracked. As an incentive for people paying cash, the price is adjusted to allow them to pump extra fuel

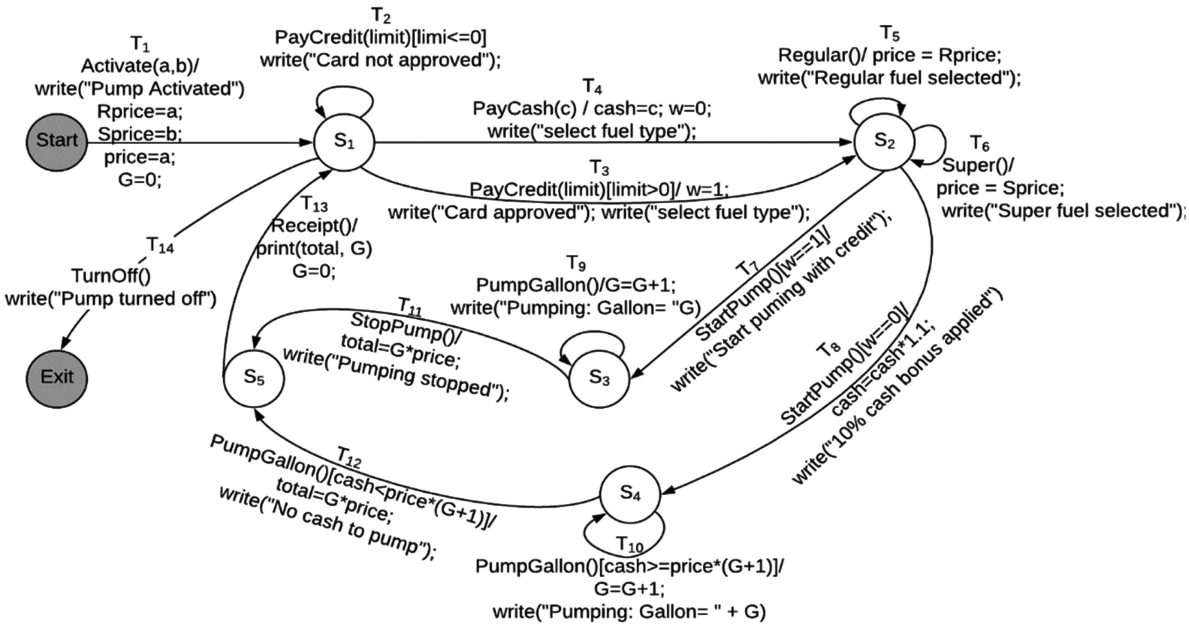
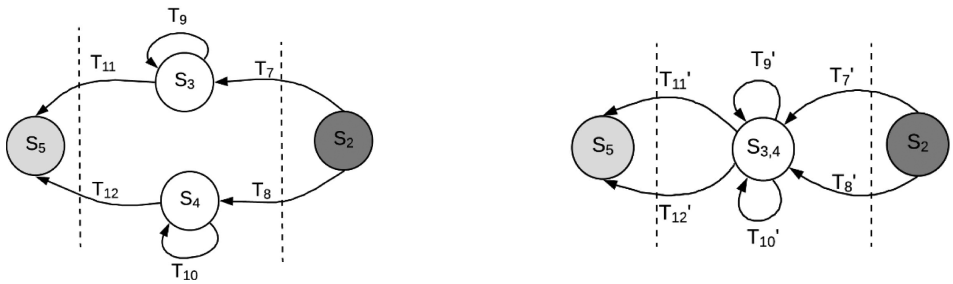


Fig. 11. Original fuel pump model.



T_7: StartPump()[w==1]/ write("Start pumping with credit");
T_8: StartPump()[w==0]/ cash=cash*1.1;
write("10% cash bonus applied")
T_9: PumpGallon()/G=G+1; write("Pumping: Gallon= "G)
T_{10}: PumpGallon()[cash>=price*(G+1)] /G=G+1;
write("Pumping: Gallon= "+G)
T_{11}: StopPump()/total=G*price; write("Pumping stopped");
T_{12}: PumpGallon()[cash>=price*(G+1)] /
total=G*price; write("Start pumping with credit");

(a) SM

T_7': StartPump()[w==1]/ ***v = True;*** write("Start pumping with credit");
T_8': StartPump()[w==0]/ ***v = False;*** cash=cash*1.1;
write("10% cash bonus applied")
T_9': PumpGallon()[***v == True***]/G=G+1; write("Pumping: Gallon= "+G)
T_{10}': PumpGallon()[***v == False*** && cash>=price*(G+1)] / G=G+1;
write("Pumping: Gallon= "G)
T_{11}': StopPump()[***v == True***]/total=G*price; write("Pumping stopped");
T_{12}': PumpGallon()[***v == False*** && cash>=price*(G+1)] /
total=G*price; write("Start pumping with credit");

(b) SM_T

Fig. 12. Transformation 1- merging states S3 and S4 in the model of Fig. 11.

equivalent to 10% of their money. The gas nozzle will keep pumping as long as the amount of cash has not been reached (for people paying cash), or as long as no explicit stop action has been detected (for people paying with a credit card). Finally, when the pump stops, the total price is calculated, and a receipt is printed out. At this stage, the pump can be used by the next customer turned off.

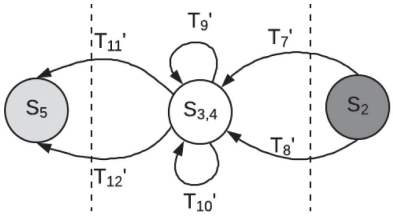
TABLE 1
Transformation Sequence for Fuel Pump Model

Transformation sequence	*Change Impact*
Tr1: Merging states S_3 and S_4	5.07
Tr2: Merging transitions T_9 and T_{10}	4.23
Tr3: Moving G=0 forward wrt S_1	3.46
Tr4: Moving G=0 forward wrt S_2	3.46
Tr5: Moving total=G*price forward	3.08

For this model, we apply transformations that target reducing the average change impact of the model. The average impact of a potential change applied to this model prior to the transformation is 5.07, as demonstrated in [13]. Using the search algorithm presented in [13], we identify a sequence of five transformations (Table 1) that enhance the model's average change impact from 5.07 to 3.08, a reduction of 39% in the change impact. When applying these transformation rules to the model in Fig. 11, the transformed model in Fig. 17 is generated.

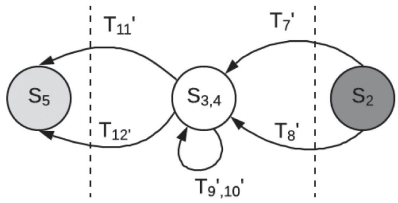
6.2.1 Merging States

The first transformation consists of merging states S_3 and S_4. Applying algorithm 4.2.1, the sub-model SM, which has



T9': PumpGallon()[*v* == ***True***]/G=G+1; write("Pumping: Gallon= "+G)
T10': PumpGallon()[*v* == ***False*** && cash
* All other transitions for this sub-model are presented in Fig. 16 (b)

(a) SM



T9', 10': PumpGallon()[*v*== ***True*** ‖*v* ==***'False*** && cash>=price*(G+1)]/
G=G+1; write("Pumping: Gallon= "G)
* All other transitions remain in SM_T remain the same as SM

(b) SM_T

Fig. 13. Transformation 2- merging transitions T9 and T10.

two internal states S_3 and S_4, becomes SM_T with a single combined state $S_{3,4}$, as depicted in Fig. 12.

The state-merging algorithm starts by calculating SM (according to algorithm 4). It then proceeds by computing the different components of the transformed sub-model, SM_T, based on the components of the original sub-model, namely, the set of internal states, Q, the set of sub-model variables, V, the set of sub-model actions, O, and the set of transitions, R. The remaining sub-model components, Σ, Q_{en} and Q_{ex}, remain unchanged for both sub-models.

In this example, each of the two transitions happens to have the same origin and target states as they are self-looping. Consequently, no action was added to them, as triggering these transitions does not reflect any change in the original model's state. However, whether these transitions should be triggered or not in SM_T depends on whether the preceding transition has S_3 or S_4 as a target state in SM. Consequently, T_9 in SM becomes T_9' in SM_T by adding the condition [v == True], and T_{10} in SM becomes T_{10}' in SMT by adding the condition [v == False] (algorithm lines 14-24). Notice that each time a transition undergoes any change, its name changes by keeping the same number and adding an apostrophe.

The algorithm calculates Q, the internal states of the transformed sub-model SM_T, by combining the two states of the original sub-model SM as one state, $S_{3,4}$ (algorithm line 2). V, the set of SM_T variables, consists of the same variables of SM in addition to a new variable, v, added to distinguish between the two merged states (algorithm line 3).

The set of SM_T actions, O, consists of the same actions defined in SM in addition to two new actions: "v = True" action to refer to state S_3 in SM, and "v = False" to refer to state S_4 in SM (algorithm line 4). R, the set of transitions of SM_T is calculated by separately calculating T_{in}, T_{out}, and T_r. The set of incoming transitions to SM is $T_{in} = \{T_7, T_8\}$. When the target state of the incoming transition is S_3 in SM, the action "v = True" is added to the transitions' sequence of actions in SM_T, while when the target state is S_4, the action "v = False" is added to the transitions sequence of actions in SM_T. Consequently, T_7 in SM becomes T_7' in SM_T by adding "v = True" to its actions, and T_8 in SM becomes T_8' in SM_T by adding "v = False" to its actions (algorithm lines 6-9).

The outgoing set transitions of SM is $T_{out} = \{T_{11}, T_{12}\}$. When the origin state of an outgoing transition is S_3 in SM, the condition "v == True" is added to the transition's conditions in SM_T. Similarly, when the origin state is S4, [v == False] is added. Consequently, T_{11} in SM becomes T_{11}' in SM_T by adding the condition [v == True], and T_{12} becomes T_{12}' by adding the condition [V == False] with an AND logical operator (algorithm lines 10-13). Finally, for all internal transitions in SM, $T_r = \{T_9, T_{10}\}$, both their origin and target states are checked.

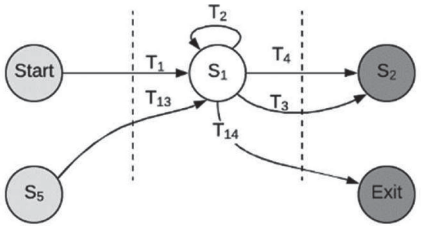
6.2.2 Merging Transitions

Having merged states S_3 and S_4, it is possible now to merge transitions T_9' and T_{10}' since the pre-condition of the transitions-merging algorithm is satisfied. The pre-conditions states that two transitions can be merged if they share the same event, actions, source state, and target state. The algorithm starts by computing the sub-model containing the source and target states of the transitions to be merged, in this case, Q(SM) = $\{S_{3,4}\}$, which produces the sub-model depicted in Fig. 13a. To generate the transformed sub-model, the algorithm starts by initializing R(SM_T) to R(SM) in line 2. Then, it removes transitions T_9' and T_{10}' from R (SM_T). It then generates a condition combining the two conditions of T_9 and T_{10} and joining them with an OR operator (line 3). This combined condition is used as the condition of the new transition, $T_{9',10}'$ replacing T_9' and T_{10}' in SM_T. This new transition has the same event, actions, source, and target states as the merged transitions. Transition $T_{9',10}'$ is added to R(SM_T), generating the new transformed sub-model, SM_T, as depicted in Fig. 13b.

6.2.3 Moving Assignment Action Forward

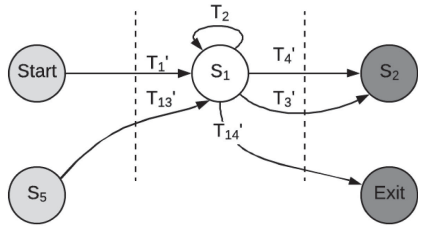
Looking at state S_1 in the fuel pump model, we can recognize that the assignment (G = 0) is declared on each incoming transition in T_{in}(S_1). In order to move the assignment forward with respect to S_1, the pre-conditions set in algorithm 4.4.1 should be satisfied. Indeed, all pre-conditions are satisfied for the assignment expression (G = 0) where G is not a parameter, and it does not use event parameters. It is defined for all transitions in $T_{in}(S_1) = \{T_1, T_{13}\}$, and it is not used along any of the paths to $T_{out}(S_1) = \{T_3, T_4, T_{14}\}$.

Having verified the pre-conditions, the algorithm then starts by computing the sub-model having S_1 as its single internal state. This model is depicted in Fig. 14a. T_{in} for SM_T is then calculated by removing the action (G = 0) from the incoming transitions $\{T_1, T_{13}\}$(algorithm lines 16-20). T_{out} for SM_T is calculated by adding the action (G = 0) to be the first action for each outgoing transition $\{T_3, T_4, T_{13}\}$ (algorithm lines 21-24). The remaining components of SM_T remain the



T_1: Activate(a,b)/write("Pump Activated"); Rprice = a; Sprice = b; price = a; G = 0;
T_2: PayCredit(limit)[limi<=0]/write("Card not approved");
T_3: PayCredit(limit)[limit>0]/ w=1;write("Card approved"); write("select fuel type");
T_4: PayCash(c) / cash=c; w=0;write("select fuel type");
T_{13}: Receipt()/print(total, G);G=0;
T_{14}: TurnOff()/write("Pump turned off");

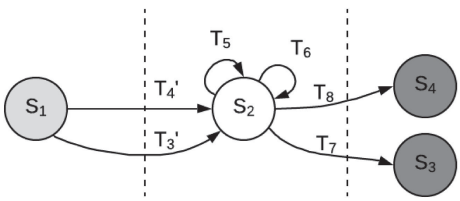
(a) SM



T_1': Activate(a,b)/write("Pump Activated"); Rprice = a; Sprice = b; price = a; ~~G = 0;~~
T_2: PayCredit(limit)[limi<=0]/write("Card not approved");
T_3': PayCredit(limit)[limit>0]/ w=1; **G = 0;** write("Card approved"); write("select fuel type");
T_4': PayCash(c) / cash=c; **G = 0;** w=0;write("select fuel type");
T_{13}': Receipt()/print(total, G);~~G=0;~~
T_{14}': TurnOff()/**G = 0;** write("Pump turned off");

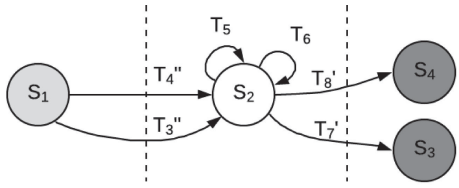
(b) SM_T

Fig. 14. Transformation 3- moving (G=0) forward with respect to S1.



T_3': PayCredit(limit)[limit>0]/ w=1; **G = 0;** write("Card approved"); write("select fuel type");
T_4': PayCash(c) / cash=c; **G = 0;** w=0;write("select fuel type");
T_5: Regular()/ price = Rprice; write("Regular fuel selected");
T_6: Super()/ price = Sprice; write("Super fuel selected");
T_7: StartPump()[w==1]/ write("Start pumping with credit");
T_8: StartPump()[w==0]/ cash=cash*1.1; write("10% cash bonus applied")

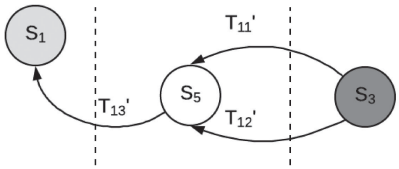
(a) SM



T_3": PayCredit(limit)[limit>0]/ w=1; ~~**G = 0;**~~ write("Card approved"); write("select fuel type");
T_4": PayCash(c) / cash=c;~~ **G = 0;**~~ w=0;write("select fuel type");
T_5: Regular()/ price = Rprice; write("Regular fuel selected");
T_6: Super()/ price = Sprice; write("Super fuel selected");
T_7': StartPump()[w==1]/ **G = 0;** write("Start pumping with credit");
T_8': StartPump()[w==0]/ **G = 0;** cash=cash*1.1; write("10% cash bonus applied")

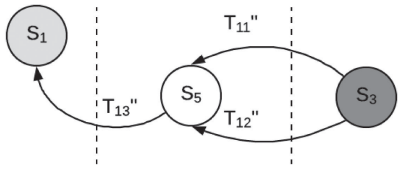
(b) SM_T

Fig. 15. Transformation 4 - moving (G=0) forward with respect to S_2.



T_{11}': StopPump()[***v* == *True***]/total=G*price; write("Pumping stopped");
T_{12}': PumpGallon()[***v* == *False*** && cash>=price*(G+1)] / total=G*price; write("Start pumping with credit");
T_{13}': Receipt()/print(total, G);

(a) SM



T_{11}": StopPump()[***v* == *True***]/~~**total=G*price;**~~ write("Pumping stopped");
T_{12}": PumpGallon()[***v* == *False*** && cash>=price*(G+1)] / ~~**total=G*price;**~~ write("Start pumping with credit");
T_{13}": Receipt()/**total=G*price;** print(total, G);

(b) SM_T

Fig. 16. Transformation 5 - moving (total = G * price) forward with respect to S_5.

same as the components of SM. The generated SM_T is depicted in Fig. 14b. Notice that each time a transition undergoes a change its name changes by keeping the same transition number and adding an apostrophe. Unchanged transitions keep their original name.

Having moved (G=0) forward with respect to S_1, we recognize that it can be moved forward even further with respect to state S_2. Indeed, S_2 has two incoming transitions T_4' and T_3', both defining the same action. Similar to the previous transformation, this action also satisfies all the pre-conditions of algorithm 4.4.2 with respect to state S_2. For this transformation, SM and SM_T are presented in Fig. 15.

Lastly, it is also possible to move assignment action (total = G * price) forward from T_{11}' and T_{12}' to T_{13}. Similarly, all pre-conditions for this assignment action are satisfied. The algorithm will compute the sub-model SM depicted in Fig. 16(a), generating the transformed model depicted in Fig. 16b. Finally, Fig. 17 shows the last version of the model after applying all of the transformations.

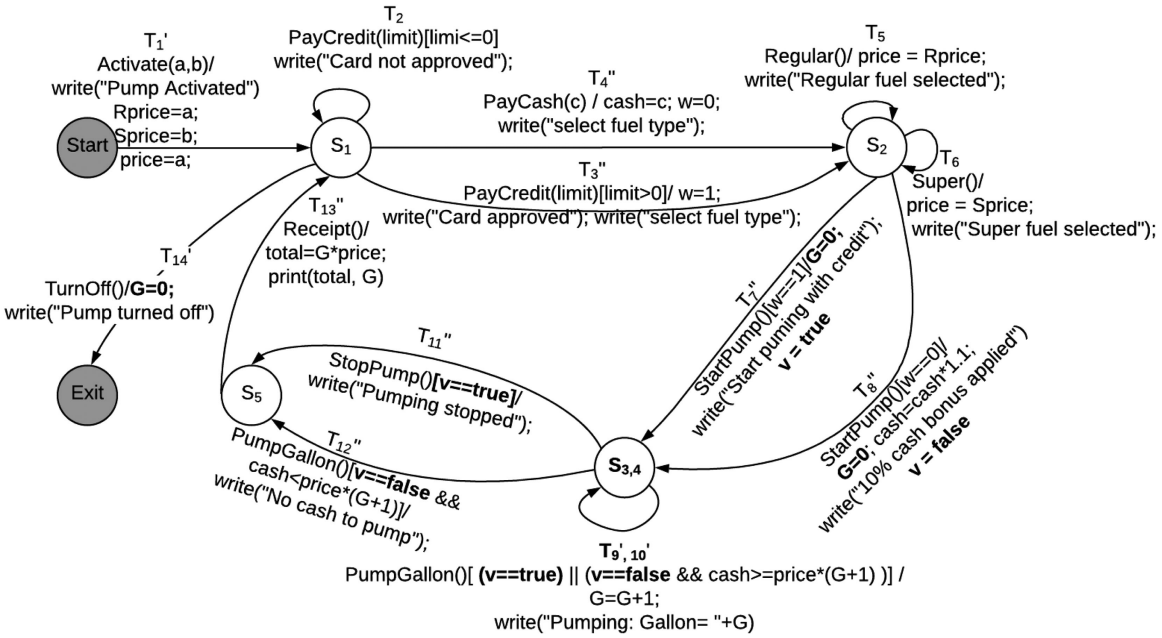


Fig. 17. Transformed fuel pump model M_T.

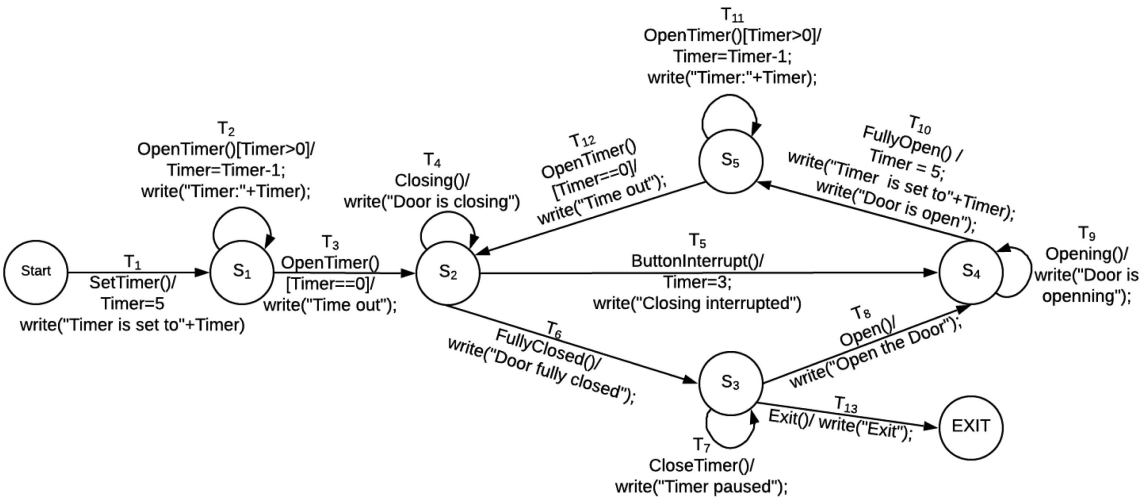


Fig. 18. Door control EFSM original model.

6.3 Door Control Model

In this section, we consider a slightly modified version of the elevator's door EFSM used in [29]. To account for external actions in this model, we added write statements on all transitions (Fig. 18). The purpose of these write statements is simply to represent any other external action that might take place (e.g., beeping when the door is opening or announcing that "the door is closing"). The model represents the main actions of the door: opening, waiting for passengers to move in or out, and then closing.

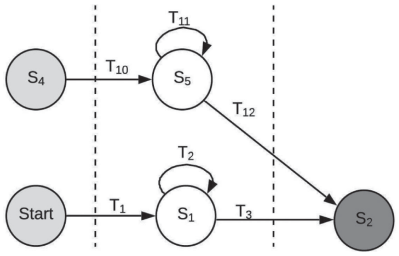
For this model, we apply transformations which target reducing the size of the model without increasing its density. The model initially has a total of 7 states and 13 transitions; its model density is hence $13/7 = 1.86$ transitions per state. In order to reduce the model size, a possible sequence of transformation is demonstrated in Table 2.

Figs. 19 and 20 show the sub-model SM and the transformed sub-model SM_T for the three transformations listed in Table 2. The transformed model after merging the states and the transitions is presented in Fig. 21.

Compared to the original model, the transformed model has two fewer transitions and one less state. This is a reduction of 14% in the model size considering the number of states (7 to 6), and a reduction of 15% considering the

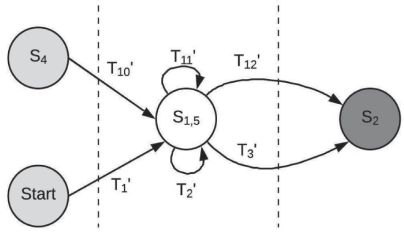
TABLE 2
Transformation Sequence for Door Control Model

Transformation sequence	*Size*		*Transition Density*
	Transitions	*States*	
Tr0: Original model	13	7	1.86
Tr1: Merging S_1 and S_5	13	6	2.17
Tr2: Merging T_2 and T_{11}	12	6	2
Tr3: Merging T_3 and T_{12}	11	6	1.83



T_{in}(SM)={T_1, T_{10}}
T_1: SetTimer()/Timer=5; write("Timer is set to"+Timer)
T_{10}: FullyOpen() / Timer = 5;write("Timer is set to"+Timer);
write("Door is open");
T_{out}={T_3, T_{12}}
T_3: OpenTimer()[Timer==0]/ write("Time out");
T_{12}: OpenTimer()[Timer==0]/ write("Time out");
T_r={T_2,T_{11}}
T_2: OpenTimer()[Timer>0]/ Timer=Timer-1; write("Timer:"+Timer);
T_{11}: OpenTimer()[Timer>0]/ Timer=Timer-1; write("Timer:"+Timer);

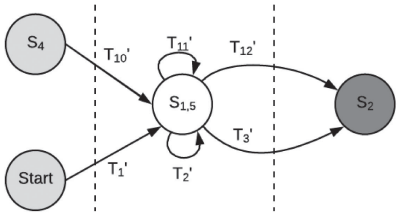
(a) SM



T_{in}={T_1, T_{10}}
T_1': SetTimer()/Timer=5; write("Timer is set to"+Timer); ***v* = *True***;
T_{10}': FullyOpen() / Timer = 5;write("Timer is set to"+Timer);
write("Door is open"); ***v* = *False***;
T_{out}={T_3, T_{12}}
T_3': OpenTimer()[***v* == *True*** && Timer==0]/ write("Time out");
T_{12}': OpenTimer()[***v* ==*False*** && Timer==0]/ write("Time out");
T_r={T_2,T_{11}}
T_2': OpenTimer()[***v* == *True*** && Timer>0]/ Timer=Timer-1;
write("Timer:"+Timer);
T_{11}': OpenTimer()[***v* == *False*** && Timer>0]/ Timer=Timer-1;
write("Timer:"+Timer);

(b) SM_T

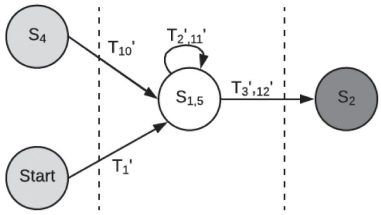
Fig. 19. Transformation 1: merging state S1 and S5.



T_2': OpenTimer()[***v* == *True*** && Timer>0]/Timer=Timer-1;
write("Timer:"+Timer);
T_{11}': OpenTimer()[***v* == *False*** && Timer>0]/Timer=Timer-1;
write("Timer:"+Timer);

T_3': OpenTimer()[***v* == *True*** && Timer==0]/ write("Time out");
T_{12}': OpenTimer()[***v* ==*False*** && Timer==0]/ write("Time out");

(a) SM



$T_{2',11'}$: OpenTimer()[***v* == *True*** && Timer>0 || ***v* == *False*** && Timer>0]/
Timer=Timer-1; write("Timer:"+Timer);
$T_{3',12'}$: OpenTimer()[***v* == *True*** && Timer==0 || ***v*== *False*** && Timer ==0]/
write("Time out");

* Notice that a developer or expression analyzer can then simplify the conditions used in the above two merged transitions as follows:

$T_{2',11'}$: OpenTimer()[Timer>0]/ Timer=Timer-1; write("Timer:"+Timer);
$T_{3',12'}$: OpenTimer()[Timer ==0]/ write("Time out");

(b) SM_T

Fig. 20. Merging transitions T2' & T11' and T3' & T12' in Fig. 19c.

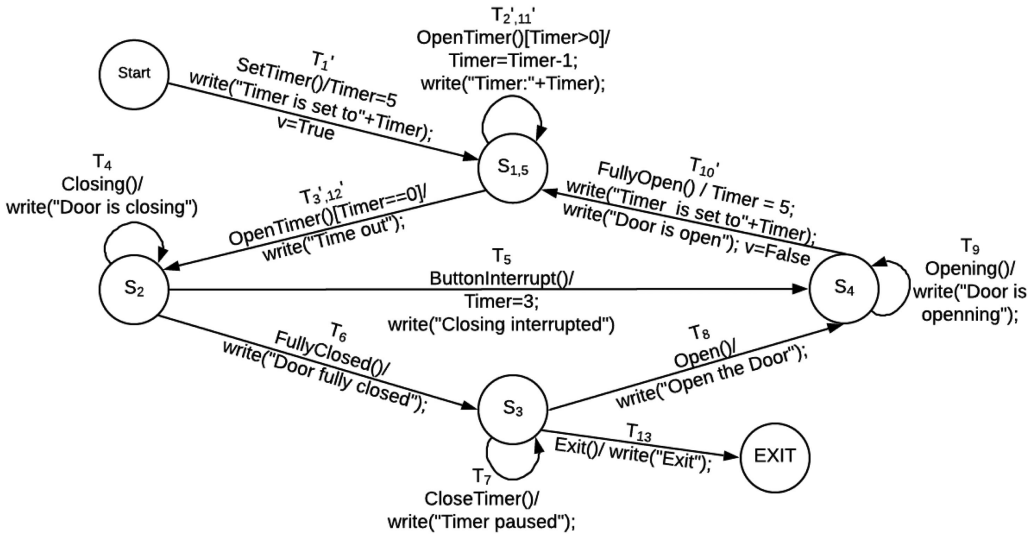


Fig. 21. Transformed door control model M_T.

TABLE 3
Five Test Cases for Fuel Pump Model

Test Case 1	
ES_1	<Activate(5,6), PayCash(10), Regular(), StartPump(), PumpGallon(), PumpGallon(), PumpGallon(), Receipt(), TurnOff()>
TS(M, ES_1)	<T_1, T_4, T_5, T_8, T_{10}, T_{10}, T_{12}, T_{13}, T_{14}>
TS(M_T,ES_1)	<T_1', T_4'', T_5, T_8'', **$T_{9,10}$**, **$T_{9,10}$**, T_{12}'', T_{13}'',T_{14}'>
AS(M,ES_1)	<write("Pump Activated"), Rpice=a, Sprice=b, price=a,G=0,write("select fuel type"),cash=c,w=0, price = Rprice, write("Regular fuel selected"), cash=cach*1.1, write("10% cash bonus applied"), G=G+1, write("Pumping: Gallon="+G), G=G+1, write("Pumping: Gallon="+G), total=G*price, write("No cash to pump"), print(total, G), G=0, Write("Pump turned off")>
AS(M_T,ES_1)	<write("Pump Activated"), Rpice=a, Sprice=b, price=a,write("select fuel type"),cash=c,w=0, price = Rprice, write("Regular fuel selected"),**G=0,**cash=cach*1.1, write("10% cash bonus applied"), **v= *false***, G=G+1, write("Pumping: Gallon="+G), G=G+1, write("Pumping: Gallon="+G), total=G*price, write("No cash to pump"), print(total, G), G=0, Write("Pump turned off")>
AS_E(M, ES_1)/ AS_E(M_T,ES_1)	<write("Pump Activated"), write("select fuel type"), write("Regular fuel selected"), write("10% cash bonus applied"), write("Pumping: Gallon="+G), write("Pumping: Gallon="+G), write("No cash to pump"), print(total,G),Write("Pump turned off")>
Res(M,ES_1)/ Res(M_T,ES_1)	<"Pump Activated", "select fuel type", "Regular fuel selected","10% cash bonus applied", "Pumping: Gallon =1", "Pumping: Gallon =2","No Cash to Pump", receipt printed out(total=10,G=2, "Pump turned off")>
Test Case 2*	
ES_2	< Activate(5, 6), PayCredit(50), Regular(), StartPump(), PumpGallon(), PumpGallon(), StopPump(), Receipt(), TurnOff()>
TS(M, ES_2)	< T_1, T_3, T_5, T_7, T_9, T_9, T_{11}, T_{13}, T_{14}>
TS(M_T,ES_2)	<T_1', T_3'', T_5, T_7'', **$T_{9,10}$**, **$T_{9,10}$**, T_{11}'', T_{13}'', T_{14}'>
Res(M,ES_2)/ Res(M_T,ES_2)	<"Pump activated", "Card approved", "Select fuel type", "Regular fuel selected", "Start pumping with credit", "Pumping: Gallon=1", "Pumping: Gallon=2", "Pumping Stop", receipt printed out(total=10,G=2, "Pump turned off")>
Test Case 3*	
ES_3	<Activate(5, 6), PayCash(12), Super(), StartPump(), PumpGallon(), PumpGallon(), PumpGallon(), Receipt(), TurnOff()>
TS(M, ES_3)	< T_1, T_4, T_6, T_8, T_{10}, T_{10}, T_{12}, T_{13}, T_{14}>
TS(M_T,ES_3)	< T_1', T_4'', T_6, T_8'', **$T_{9,10}$**, **$T_{9,10}$**, T_{12}'', T_{13}'', T_{14}'>
Res(M,ES_3)/ Res(M_T,ES_3)	<" Pump activated", "Select fuel type", "Super fuel selected", "10% cash bonus applied","Pumping: Gallon=1", "Pumping: Gallon=2", "No cash to pump", receipt printed out(total=12,G=2, "Pump turned off")
Test Case 4*	
ES_4	< Activate(5, 6), PayCredit(-100), PayCash(5), Super(), StartPump(), PumpGallon(), Receipt(), TurnOff()>
TS(M, ES_4)	< T_1, T_2, T_4, T_6, T_8, T_{12}, T_{13}, T_{14}>
TS(M_T,ES_4)	< T_1', T_2, T_4'', T_6, T_8'', T_{12}'', T_{13}'', T_{14}'>
Res(M,ES_4)/ Res(M_T,ES_4)	<"Pump activated", "Card not approved", "Select Fuel Type", "Super fuel selected", "10% cash bonus applied", "No Cash To Pump", receipt printed out(total=0,G=0, "Pump turned off")
Test Case 5*	
ES_5	< Activate(1, 2), PayCredit(200), Super(), StartPump(), PumpGallon(), StopPump(), Receipt(), TurnOff()>
TS(M, ES_5)	<T_1, T_3, T_6, T_7, T_9, T_{11}, T_{13}, T_{14}>
TS(M_T,ES_5)	<T_1', T_3'', T_6, T_7'', **$T_{9,10}$**, T_{11}'', T_{13}'', T_{14}'>
Res(M,ES_5)/ Res(M_T,ES_5)	< "Pump Activated", "Card approved", "Select fuel type", "Super fuel selected", "Start Pumping with credit", "Pumping: Gallon=1", "Pumping Stop", receipt printed out(total=2,G=1, "Pump turned off")

* *Because of space limitation, the sets AS(M,ES), AS(M_T,ES), AS_E(M, ES), and AS_E(M_T,ES) are not shown, but they can be easily generated similar to (Test Case 1) by looking at the original and the transformed Fuel Pump models.*

number of transitions (13 to 11). With this reduction, the model's transition density stayed almost the same (1.86 to 1.83), which overall demonstrates that the model size was reduced without negatively impacting its complexity.

6.4 Testing the Observable Behavior of the Models

For each of the two case studies, we generated five test cases, which satisfy the transition coverage criterion, to show that transformed models preserved the same observable behavior as the original models. Table 3 presents the test cases that are applied to the original fuel pump model (Fig. 11) and its corresponding transformed model (Fig. 17). Table 4 presents the test cases that are applied to the original door control model (Fig. 18) and its corresponding transformed model (Fig. 21).

A test case is defined in terms of a sequence of events taking place, and it is denoted as ES. The result of running the test cases is demonstrated at two levels, transitions and results of external actions. At the first level, we show the sequence of transitions executed as a result of running the test case. This sequence of transitions is denoted as TS. We show the sequence of transitions executed for both the original model M and the transformed model M_T, where TS(M, ES) refers to the sequence of transitions executed when running the test case ES on the original model M, while TS(M_T, ES) refers to the sequence of the transitions executed when running the test case ES on the transformed model M_T. At the second level of representing the result of the test case, we show the result of the external actions of the executed sequence of transitions; this result is denoted as Res(M, ES) for the original model, and Res(M_T, ES) for the transformed model. The notation for ES, TS, and Res is detailed in Section 2.2, and an example of sub-model execution is given in Section 3.1.3.

Considering the first test case in Table 3, ES_1, the first event (*Activate(56)*) activates the fuel pump and sets the regular and super fuel prices to 5 and 6, respectively. The user

TABLE 4
Five Test Cases for Door Control Model

Test Case 1:	
ES_1	<SetTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), Closing(), FullyClosed(), CloseTimer(), Open(), Opening(), FullyOpen(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), FullyClosed(), CloseTimer, Exit()>
$TS(M, ES_1)$	< T_1, T_2, T_2, T_2, T_2, T_2, T_3, T_4, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{11}, T_{11}, T_{11}, T_{11}, T_{12}, T_6, T_7, T_{13}>
$TS(M_T,ES_1)$	< T_1, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{3,12}}$, T_4, T_6, T_7, T_8, T_9, T_{10}, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{3,12}}$, T_6, T_7, T_{13} >
$Res(M,ES_1)$/ $Res(M_T,ES_1)$	< “Timer is set to:5”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Door is closing”, “Door fully closed”, “Timer paused”, “Open the Door”, “Door is opening”, “Timer is set to:5”, “Door is open”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Door fully closed”, “Timer paused”, “Exit”>
Test Case 2:	
ES_2	< SetTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), Closing(), Fully Closed(), Exit()>
$TS(M, ES_2)$	< T_1, T_2, T_2, T_2, T_2, T_2, T_3, T_4, T_6, T_{13}>
$TS(M_T,ES_2)$	< T_1, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{3,12}}$, T_4, T_6, T_{13}>
$Res(M,ES_2)$/ $Res(M_T,ES_2)$	<“Timer is set to:5”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Door is closing”, “Door fully closed”, “Exit”>
Test Case 3:	
ES_3	< SetTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), Closing(), ButtonInterrupt(), FullyOpen(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), Fully Closed(), Exit()>
$TS(M, ES_3)$	< T_1, T_2, T_2, T_2, T_2, T_2, T_3, T_4, T_5, T_{10}, T_{11}, T_{11}, T_{11}, T_{11}, T_{11}, T_{12}, T_6, T_{13}>
$TS(M_T,ES_3)$	< T_1, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{3,12}}$, T_4, T_5, T_{10}, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, T_{12}, T_6, T_{13}>
$Res(M,ES_3)$/ $Res(M_T,ES_3)$	<“Timer is set to:5”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Door is closing”, “Button Interrupted”, “Timer is set to:5”, “Door is open”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Door fully closed”, “Exit”>
Test Case 4:	
ES_4	< SetTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), ButtonInterrupt(), Opening(), FullyOpen(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), Fully Closed(), CloseTime(), Exit()>
$TS(M, ES_4)$	<T_1, T_2, T_2, T_2, T_2, T_2, T_3, T_5, T_9, T_{10}, T_{11}, T_{11}, T_{11}, T_{11}, T_{11}, T_{12}, T_6, T_7, T_{13}>
$TS(M_T,ES_4)$	<T_1, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{3,12}}$, $\mathbf{T_5}$, $\mathbf{T_9}$, $\mathbf{T_{10}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, T_{12}, T_6, T_7, T_{13}>
$Res(M,ES_4)$/ $Res(M_T,ES_4)$	< “Timer is set to:5”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Button Interrupted”, “Door is opening”, “Timer is set to:5”, “Door is open”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Door fully closed”, “Timer paused”, “Exit”>
Test Case 5:	
ES_5	<SetTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), ButtonInterrupt(), Opening(), FullyOpen(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), OpenTimer(), Fully Closed(), CloseTime(), CloseTimer(), CloseTimer(), Exit()>
$TS(M, ES_5)$	<T_1, T_2, T_2, T_2, T_2, T_2, T_3, T_5, T_9, T_{10}, T_{11}, T_{11}, T_{11}, T_{11}, T_{11}, T_{12}, T_6, T_7, T_7, T_7, T_{13}>
$TS(M_T,ES_5)$	<T_1, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{3,12}}$, $\mathbf{T_5}$, $\mathbf{T_9}$, $\mathbf{T_{10}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{2,11}}$, $\mathbf{T_{3,12}}$, T_6, T_7, T_7, T_7, T_{13}>
$Res(M,ES_5)$/ $Res(M_T,ES_5)$	< “Timer is set to:5”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Button Interrupted”, “Door is opening”, “Timer is set to:5”, “Door is open”, “Timer:4”, “Timer:3”, “Timer:2”, “Timer:1”, “Timer:0”, “Time out”, “Door fully closed”, “Timer paused”, “Timer paused”, “Timer paused”, “Exit”>

then decides to pay cash and indicates 10 as the maximum amount of cash to spend (*payCash(10)*) and chooses regular fuel (*regular()*). The user then starts pumping gas (pumpGallon()), which takes place three times. The first two times, the gas is actually being pumped because there is enough cash to cover the cost (5*2), but the third time when the event *pumpGallon* occurs, the pump stops as there is not enough cash for one more gallon. The receipt is then printed out, and the machine goes back to state S_1, where either a new customer may start the process of fueling gas (by choosing the payment method), or an employee may turn off the pump. According to ES_1, the last event that takes place is turnoff().

Upon executing the above sequence of events, ES_1, on the original model M in Fig. 11, the sequence of executed transitions $TS(M, ES_1)$ takes place. When the same sequence is executed on model M_T in Fig. 17, the sequence of transitions $TS(M_T, ES_1)$ is executed. At the exception of T_5, all transitions in $TS(M_T, ES_1)$ are different from those in $TS(M, ES_1)$ as they have experienced at least one or more change after applying the different transformations in Table 2. Despite these differences, each transition in $TS(M_T, ES_1)$ is semantically equivalent to its matching transition in TS(M, ES). For example, T_1 and T_1' are semantically equivalent, given that they have the same event (***Activate***), and the same external action (*write(“Pump Activated”)*). Consequently, despite the fact that $AS(M,ES_1) \neq AS(M_T,ES_1)$, the sets of external actions $AS_E(M,ES_1)$ and $AS_E(MT,ES_1)$ are equivalent, and they generate exactly the same external results $Res(M,ES_1) == Res(M_T,ES_1)$.

Looking at the different test cases in both Tables 3 and 4, we recognize that although the sequence of transitions executed may differ between the transformed and the original model for the same test case, the observable behavior is the same. This observable behavior is captured by the sequence of the external actions $AS_E(M, ES)$ and the resulting observable output Res(M, ES) for both the original and the transformed models.

6.5 Threats to Validity

Although the two case studies presented in this section were not meant as experimental studies, but rather as examples demonstrating the applicability of the developed transformation rules and as a complementary approach to the theoretical proofs, we can still list two main threats to validity;

namely: the two case studies are based on small models, and only five test-cases were applied on each model.

The choice of these two models was based on the fact that they are easy to understand and can be easily followed by the reader, along with the demonstration of how the transformation rules were applied. Despite their small size, we have been successful in demonstrating that our approach may improve the different characteristics of these models.

As for the number of test cases, the selected test cases satisfy the full transition coverage criterion, which is a commonly used coverage method for test-suites, where each transition is covered by at least one test case. Consequently, despite the small size of this test-suite, it significantly increases the confidence in the correctness of the presented transformation rules and the proving process.

7 Related Work

Since MDE relies on models to develop, maintain, and evolve software, it is frequently the case that models need to be modified to improve their quality characteristics. Transformation is a major activity in MDE that allows transforming one model to another model or to source-code. Refactoring is considered a special type of transformations where a model is changed to another model of the same type to enhance its quality characteristics without changing its behavior.

Several works attempted to use model transformation languages, tools, and techniques to apply model refactoring [46], [42], [31], and [14]. Zhang *et al*. [46] used a model transformation engine along with a refactoring browser to apply generic as well as domain-specific model refactorings. Their approach didn't consider state-machine refactoring, and they didn't provide a mechanism to verify user-defined refactorings. Mens *et al*. [42] used graph transformation dependency analysis to detect model inconsistencies. They specified model inconsistencies as graph transformation rules and used critical pair analysis to detect dependencies between the rules in an attempt to resolve conflicting rules and find the best order of applying them. Their goal was thus different than ours. They aimed at identifying structural inconsistencies within the model and then resolved it with pre-defined rules, while our work assumes that the model is correct, but its quality could be enhanced by applying refactoring rules. Additionally, they did not address verifying the rules that they used in their approach. Weis *et al*. [31] developed a tool to allow component-based software developers to express round-trip transformations between the elements of a platform-independent model (PIM) and a platform-specific model (PSM). They expressed the transformation rule using a visual notation that consists of two search patterns and one replacement pattern. The search patterns are used to look for the location to apply the transformation within each model, while the replacement pattern is used to apply the transformation once the matching search pattern is identified. In their approach, they require predefining the order of applying the different transformation rules with an activity diagram where the elements of the activity diagrams are the transformation rules themselves. The authors noted the tremendous difference between refactoring transformations and other types of transformations, given that the latter generates an output model that has no immediate structural similarities with the input model. Their suggestion of the use of a search pattern is, however, similar to the assumption that we make in our work about the use of a search algorithm to identify the location (e.g., sub-model) where a transformation rule can be applied. Using a search algorithm, the order of applying the transformation rules is dynamically determined based on the fitness function[13]. Lastly, Rahimi *et al*. [14] evaluated the suitability of five transformation apporaches (QVT-R, ATL, Kermeta, UML-RSDS, and GrGen.Net) for model refactoring. They used ISO/IEC 9126-1 quality characteristics as the basis of the evaluation. Their study found that GrGen and UML-RSDS are more suited for refactoring transformations given their support for update-in-place transformations where the source model is updated to generate a modified one of the same model type. They also noted that, when considering refactoring transformations, none of the five approaches satisfied half of the quality subcharacteristics considered for the evaluation of the approaches. Our work, on the other hand, does not adopt a particular transformation platform; instead, it is presented at a higher level of abstraction, which allows its adoption by different transformation languages and tools. We used algorithms to specify five transformation rules, and we proved their correctness based on the presented verification approach.

One of the most important tasks when applying a transformation is to verify that it is done "correctly" [35], [36], [38], [39], [40], [49]. Despite the fact that refactoring is considered as a specific type of transformations, we can notice a significant difference in the verification requirements for refactoring transformations compared to other types of transformations [17], [31], [14]. For example, according to Baudry *et al*. [48], [28], the complexity of the required input and output data is one of the main challenges in verifying model transformations using a testing approach. For refactoring transformations, however, the output model has similar elements compared to the input model, and the complexity can be further reduced by looking at a particular element or region of the model instead of considering the entire model. In fact, our approach reduces the complexity of the input and output data required for a refactoring transformation by limiting the transformation rule to a portion of the model called "sub-model" and then reducing the complexity of the proof to only the modified portion of the model. In our approach, we focus on proving the correctness of the rule itself rather than testing the output model.

Only a few studies tried to address the verification of the semantic equivalence of models. Carious *et al*. [32] used model transformation contracts to dynamically verify refinement transformations of a state-chart model during its execution. The source and target models are thus of the same type; however, as opposed to our approach, the models are not expected to be semantically equivalent since they mainly target the evolution of the model, which naturally implies an intended change of the behavior. Hamadane *et al*. [41] proposed a semantic framework for the transformation of ER models to Relational models. Compared to our work, the input and output models of their framework are of different types, while in our work, both the input and output models are EFSM models

following the same meta-model. To prove the correctness of a transformation, Hamadane *et al.* propose using a theorem prover tool which verifies that certain properties are met on the output model after the transformation rule is applied to a given input model. Their proofing approach has two implications that are different than our approach. The first one is that their approach gives only a partial verification of the correctness of the transformation since they are only proving that certain properties are met, which doesn't guarantee the correctness of the mapping between the input and output model. The second implication is that whenever the transformation rule is applied to a different input model, its correctness needs to be checked for the generated output model. Our work, however, is different with respect to these two aspects. In our paper, we verify the correctness of the transformation rule in general regardless of the specific input model. Our verification approach allows demonstrating that for any input model, when a verified transformation rule is applied, it will correctly generate a semantically equivalent output model. Consequently, there is no need to verify the correctness of the output model whenever the rule is applied to a different input model. Secondly, the theoretical proofs that we provide to the presented rules are actual proofs of correctness, not only partial proofs of certain properties. On the other hand, our attempt for automating the verification of a transformation rule gives only partial proof of correctness as only three properties are checked, similar to the approach followed by Hamadane *et al.*

Lastly, Lano *et al.* [37] proposed a generic framework for modeling languages, transformation specifications, transformation implementations, and verification properties. They proposed the use of the framework as means of expressing the transformations and the verification properties in a formal notation that can be then encoded in a specific language such as B or Z3 to facilitate automating the proofs. Their work provides the means to express the properties of the semantic preservations for the different transformation rules. In our work, however, we provide a clear definition of how two EFSM models, two EFSM sub-models, and two EFSM transitions are considered semantically equivalent regardless of any transformation rule, and we mainly consider the dynamic semantic preservation. We then give three elements of the proof that should be verified for any transformation rule in order to verify that the rule preserves the semantical equivalence of the models.

Finally, the work presented in this paper can be considered as an extension of [13] where Korel *et al.* presented informal definitions of three EFSM refactoring transformation rules. The work was presented within the context of search-based engineering [15], [16], [27], where they presented a search algorithm to enhance EFSM models by reducing their average change impact. The search algorithm was guided by the model's "average change impact" as its fitness function; consequently, the order of applying the transformations was decided based on the improvement of this function. A case study demonstrated that the search algorithm identified a sequence of four transformations using the three transformation rules and reduced the average change impact of the model by 52%. The calculations of the average change impact are detailed in [5], where EFSM dependence analysis was used for this purpose [2], [29], [23], [18]. The work presented in this paper, however, extends the previous work by formally defining and proving the correctness of five transformation rules and introducing a generic approach for defining and verifying new refactoring transformation rules for EFSM models. Additionally, we provide definitions for semantical equivalency for EFSM models, sub-models, and transitions, which form the basis of the verification of the correctness of the refactoring rules. It is worth mentioning here that EFSM models that are used in our work are executable models that are specified with a textual notation and executed on a tool that can generate the execution traces and measure some model characteristics such as the average change impact, the model size, and the transition density.

8 Conclusion and Future Work

In this paper, we proposed an approach for developing, applying, and verifying model transformation rules for EFSM models. We also presented a transformation process that adopts the verified transformation rules. Additionally, we provided five different transformation rules and used the proposed verification approach to verify the correctness of the rules. Each one of the five model transformation rules presented in this paper has pre-conditions. Any sub-model that satisfies the preconditions of a given transformation rule can be identified as a candidate for the transformation. In any model, multiple candidate sub-models might be identified for the different transformation rules, and the selection of a sub-model for the transformation will depend on a predefined metric used to improve certain model characteristics. This selection can be done interactively by an expert system modeler or using a search algorithm [13], [19], [26]. Finally, we presented two case studies to demonstrate that applying a sequence of individually verified transformations will generate a new transformed model that is semantically equivalent to the original model. The two case studies also showed that the presented transformation rules can be applied to enhance predefined model characteristics.

It is worth mentioning that our work provides system modelers with a simplified three-step approach to prove the correctness of new transformation rules. Verifying a new rule is done only once, similar to proving a theorem. Once the rule is verified, it can then be safely applied to models without the need for further verification. The three-step approach is not fully automated; the system modeler needs to manually follow the verification steps similar to the proofs that we presented for the five transformation rules introduced in this paper. The entire transformation process (Fig. 3), on the other hand, is fully automated. This includes identifying the candidate sub-model to apply the transformation, applying the transformation rule on a sub-model, replacing one sub-model with another, and identifying the order of applying the different transformation rules.

Although automating the verification of the transformation rules is not as crucial as automating the transformation process itself, we presented in Section 5 of this paper three properties that should hold true for any new transformation rule, and these properties can be automatically checked for new transformation rules.

In future research, we plan to investigate automated methods for the verification of the model transformation rules while developing more advanced transformation rules. We also plan to develop optimized search algorithms, e.g., search based on genetic algorithms, to identify candidate sub-models for model transformations targeting enhancing the model quality. Finally, we plan to perform an extended empirical study for larger models in order to investigate the usefulness of model transformation rules. Extending the approach to other behavioral models will also be considered in our future research work.

Finally, the presented transformation process and verification approach could be generalized beyond EFSM models. For example, they could be applied to other state-based models such as UML state diagrams and SDL models. Its generalization to other behavioral models such as UML sequence and activity diagrams can be investigated in future research. Indeed, the generalization of our approach consists of two main practices, namely, limiting the boundaries of a transformation to a sub-model and verifying the correctness of the transformation rule by considering the semantical equivalence of the sub-models. Since proofs are applied on the sub-model level, not on the whole model-level, the complexity of these proofs is greatly reduced.

ACKNOWLEDGMENTS

This work was supported by the Kuwait Foundation for Advancement of Science (KFAS) through Project P116-18QS-01

REFERENCES

[1] L. Tahat, B. Korel, G. Koutsogiannakis, and N. Almasri, "State-based models in regression test suite prioritization," *Softw. Qual. J.*, vol. 25, no. 3, pp. 703–742, 2017.

[2] B. Korel, L. H. Tahat, and B. Vaysburg, "Model based regression test reduction using dependence analysis," in *Proc. Int. Conf. Softw. Maintenance*, 2002, pp. 214–223.

[3] B. Korel and L. Tahat, "Understanding modification in state-based model," in *Proc. 12th IEEE Int. Conf. Prog. Comprehension*, 2004, pp. 246–250.

[4] L. Tahat, B. Korel, M. Harman, and H. Ural, "Regression test suite prioritization using system models," *Softw. Testing Verification Rel.*, vol. 22, no. 7, pp. 481–506, 2012.

[5] N. Almasri, L. Tahat, and B. Korel, "Toward automatically quantifying the impact of a change in systems," *Softw. Qual. J.*, vol. 25, no. 3, pp. 601–640, 2017.

[6] I. Boussaïd, P. Siarry, and M. Ahmed-Nacer, "A survey on search-based model-driven engineering," *Automat. Softw. Eng.*, vol. 24, no. 2, pp. 233–294, 2017.

[7] S. R. Dalal *et al.*, "Model-based testing in practice," in *Proc. Int. Conf. Softw. Eng.*, 1999, pp. 285–294.

[8] J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications," in *Proc. 1st Int. Symp. Formal Methods Europe Ind.-Strength Formal Methods*, 1993, pp. 268–284.

[9] C. Kwang-Ting and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proc. 30th ACM/IEEE Des. Automat. Conf .*, 1993, pp. 86–91.

[10] L. H. Tahat, B. Vaysburg, B. Korel, and A. J. Bader, "Requirement-based automated black-box test generation," in *Proc. 25th Annu. Int. Comput. Softw. Appl. Conf.*, 2001, pp. 489–495.

[11] B. Vaysburg, L. Tahat, B. Korel, and A. Bader, "Automating test case generation from sdl specifications," in *Proc. 18th Int. Conf. Testing Comput. Softw.*, 2001, pp. 130–139.

[12] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, "Slicing of state -based models," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2003, pp. 34–43.

[13] B. Korel, N. Almasri, and L. Tahat, "Towards minimizing the impact of changes using search-based approach," in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds., Cham, Switzerland: Springer, 2018, pp. 262–277.

[14] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, and P. Van Gorp, "Evaluation of model transformation approaches for model refactoring," *Sci. Comput. Prog.*, vol. 85, pp. 5–40, 2014.

[15] M. Harman and B. F. Jones, "Search-based software engineering," *Inform. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.

[16] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: An empirical study," *J. Softw. Maintenance Evol. Res. Pract.*, vol. 20, no. 5, pp. 345–364, 2008.

[17] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, 2006.

[18] K. Androutsopoulos, D. Clark, M. Harman, R. M. Hierons, Z. Li, and L. Tratt, "Amorphous slicing of extended finite state machines," *IEEE Trans. Softw. Eng.*, vol. 39, no. 7, pp. 892–909, Jul. 2013.

[19] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, "Model transformation modularization as a many-objective optimization problem," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1009–1032, Nov. 2017.

[20] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[21] M. Misbhauddin and M. Alshayeb, "UML model refactoring: A systematic literature review," *Empirical Softw. Eng.*, vol. 20, no. 1, pp. 206–251, 2015.

[22] L. Lúcio *et al.*, "Model transformation intents and their properties," *Softw. Syst. Model.*, vol. 15, no. 3, pp. 647–684, 2016.

[23] K. Androutsopoulos, N. Gold, M. Harman, Z. Li, and L. Tratt, "A theoretical and empirical study of EFSM dependence," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2009, pp. 287–296.

[24] O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, *Model Driven Engineering Languages and Systems*. Berlin, Germany: Springer, 2013.

[25] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.

[26] M. W. Mkaouer, M. Kessentini, S. Bechikh, and M. Ó Cinnéide, "A robust multi-objective approach for software refactoring under uncertainty," in *Search-Based Software Engineering*, C. Le Goues and S. Yoo, Eds., Cham, Switzerland: Springer, 2014, pp. 168–183.

[27] O. Räihä, "A survey on search-based software design," *Comput. Sci. Rev.*, vol. 4, no. 4, pp. 203–249, 2010.

[28] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. L. Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Commun. ACM*, vol. 53, pp. 139–143, 2010.

[29] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, "Control dependence for extended finite state machines," in *Fundamental Approaches to Software Engineering*, M. Chechik and M. Wirsing, Eds., Berlin, Germany: Springer, 2009, pp. 216–230.

[30] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Professional, 2003.

[31] T. Weis, A. Ulbrich, and K. Geihs, "Model metamorphosis," *IEEE Softw*, vol. 20, no. 5, pp. 46–51, Sep./Oct. 2003.

[32] E. Cariou, N. Belloir, F. Barbier, and N. Djemam, "OCL contracts for the verification of model transformations," 2010. [Online]. Available: http://ecariou.perso.univ-pau.fr/papers/ocl09-presentation.pdf

[33] C. Krause, Henshin, "The eclipse foundation. eclipse.org," 2021. Retrieved: Sep. 18, 2021. [Online]. Available: https://www.eclipse.org/henshin/

[34] Object Management Group, "MDA guide (version 1.0.1)," Jun. 2003. [Online]. Available: https://www.omg.org/news/meetings/workshops/UML_2003_Manual/002_MDA_Guide_v1.0.1.pdf

[35] M. Amrani *et al.*, "Formal verification techniques for model transformations: A tridimensional classification," *J. Object Technol.*, vol. 14, no. 3, pp. 1–43, 2015.

[36] K. Stenzel, N. Moebius, and W. Reif, "Formal verification of QVT transformations for code generation," in *Model Driven Engineering Languages and Systems*, J. Whittle, T. Clark, and T. Kühne, Berlin, Germany: Springer, 2011, pp. 533–547.

[37] L. Kevin, T. Clark, and S. Kolahdouz-Rahimi, "A framework for model transformation verification," *Formal Aspects Comput.*, vol. 27, no. 1, pp. 193–235, 2015.

[38] L. Ab. Rahim and J. Whittle, "A survey of approaches for verifying model transformations," *Softw. Syst. Model.*, vol. 14, pp. 1003–1028, 2013.
[39] M. Asztalos, L. Lengyel, and T. Levendovszky "Towards automated, formal verification of model transformations," in *Proc. 3rd Int. Conf. Softw. Testing Verification Validation*, 2010, pp. 15–24.
[40] D. Calegari and N. Szasz, "Verification of model transformations: A survey of the state-of-the-art," *Electron. Notes Theor. Comput. Sci.*, vol. 292, pp. 5–25, 2013.
[41] M. E. Hamdane, K. Berramla, A. Chaoui, and A. El H. Benyamina, "A semantic framework to improve model-to-model transformations," in *Proc. Int. Conf. Europe Middle East North Afr. Inf. Syst. Technol. Support Learn.*, 2018, pp. 290–300.
[42] T. Mens, R. Van Der Straeten, and M. D'Hondt, "Detecting and resolving model inconsistencies using transformation dependency analysis," in *Proc. 9th Int. Conf. Model Driven Eng. Lang. Syst.*, 2006, pp. 200–214.
[43] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Prog.*, vol. 72, pp. 31–39, 2008.
[44] J. Cuadrado, J. Molina, and M. Tortosa, "RubyTL: A practical, extensible transformation language," in *Proc. Eur. Conf. Model Driven Architecture - Found. Appl.*, 2006, pp. 158–172.
[45] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Softw. Eng.*, vol. 21, pp. 811–853, 2016.
[46] Y. Lin, J. Zhang, and J. Gray, "A testing framework for model transformations," in *Model-Driven Software Development-Research and Practice in Software Engineering*, Berlin, Germany: Springer, 2005, pp. 219–236.
[47] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson, "Semantic anchoring with model transformations," in *Proc. Eur. Conf. Model Driven Architecture - Found. Appl.*, 2005,.
[48] B. Baudry *et al.*, "Model transformation testing challenges," in *Proc. Workshop Integration Model Driven Develop. Model Driven Testing*, 2006. [Online]. Available: https://hal.inria.fr/inria-00542781/document
[49] F. Fleurey, J. Steel, and B. Baudry, "Validation in model-driven engineering: Testing model transformations," in *Proc. 1st Int. Workshop Model Des. Validation*, 2004, pp. 29–40.
[50] J. Wang, S. Kim, and D. Carrington, "Automatic generation of test models for model transformations," in *Proc. 19th Australian Conf. Softw. Eng.*, 2008, pp. 432–440.
[51] D. Kolovos, R. Paige, and F. Polack, "The epsilon transformation language," in *Proc. Theory Pract. Model Transformations*, 2008, pp. 46–60.

Nada Almasri received the MSc and PhD degrees in computer science from INSA de Lyon, France, in 2000 and 2005, respectively. She is currently an assistant professor of management information systems with the Gulf University for Science and Technology, Kuwait. She was a lecturer with the David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada. Her research interests include software engineering and software change impact analysis.

Bogdan Korel received the MS degree in electrical engineering from the Technical University of Kiev, Ukraine, and the PhD degree in systems engineering from Oakland University, Rochester, MI, USA, in 1986. He is currently an associate professor and associate chair of the Computer Science Department, Illinois Institute of Technology. He has contributed to several journals and conference proceedings in the area of software engineering. He has authored or coauthored more than 90 conferences and journal papers. His research interests include software engineering, automated software system analysis, and software reliability and testing.

Luay Tahat received the master's degree in computer science from Northeastern Illinois University, Chicago, USA, and the PhD degree in computer science from the Illinois Institute of Technology/Illinois Tech (IIT), Chicago, USA, in 2007. Since 2008, he has been an associate professor of management information system with the Gulf University for Science and Technology, Kuwait. Before that, he has more than 15 years of professional/industrial networking experience with AT&T, Lucent Technologies/Bell Labs, Alcatel-Lucent (Nokia currently), USA, and IBM over mobile and fixed networks. He was a part-time professor with the IIT. With Alcatel-Lucent, he held several positions in software development, system engineering, and system architecture and has contributed to several areas in the fields of software engineering. He has authored or coauthored papers in more than 40 journals and conference proceedings. His research interests include software testing, software maintenance, and wireless network solutions.