# From Statecharts into Model Checking: A Hierarchy-based Translation and Specification Patterns Properties to Generate Test Cases

Valdivino Alexandre de Santiago Júnior
Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos, SP, Brazil
valdivino.santiago@inpe.br

Felipe Elias Costa da Silva
Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos, SP, Brazil
felipe.eliascs@hotmail.com

## ABSTRACT

Complexity and notation of formal methods are still major impediments for a wider use of these mathematical-based approaches in Software Engineering which include its adoption in software testing. While formal, Statecharts are relatively simple to use and many projects in different domains have been relying on them. In this paper, we present a hierarchy-based translation method, HiMoST, to generate software test cases via Model Checking. Starting with a behavioral modeling in Harel's Statecharts, we propose a method to translate from Statecharts into a general structure based on the NuSMV language, and we formalize CTL properties by means of specification patterns and a Combinatorial Interaction Testing algorithm. We also present a cost-effectiveness evaluation (quasiexperiment) to compare four different patterns/pattern scopes. Results indicate that the Precedence Chain ($P$ precedes $S$, $T$) pattern with Global scope presents the best performance.

## CCS Concepts

•**Software and its engineering** → **Software testing and debugging; Formal software verification;** *Empirical software validation;*

## Keywords

Software Testing; Model Checking; Statecharts; Specification Patterns System; Quasiexperiment

## 1. INTRODUCTION

While software testing basically aims at empirically verifying the correctness of a Software Under Test (SUT), formal methods (including formal verification methods [3]) usually deal with the mathematical (formal) correctness of the software. In the past, formal methods and testing had often been viewed as rivals [7], and researchers argued that formal methods would be enough and software testing could then be discarded.

Nowadays, we can say that both processes are important within the Verification & Validation (V&V) context, and companies/institutes can benefit from testing and formal methods because they produce information that can be used in a complementary way, thus enabling the application of both approaches in a single project through joint actions. To corroborate this statement, several studies have been published where Model Checking [3], a formal verification method, helps generating test cases [1, 2, 9, 11, 14, 12, 19, 4, 10, 18, 16, 17]. The main idea is to consider the counterexamples, i.e. traces of the Transition System ($TS$) which show that a certain formalized **trap** property ($\Phi$) is not satisfied by the $TS$, as test cases. Thus, a formalized trap property $\Phi$ forces counterexample generation. However, some studies have used **witness** properties rather than trap ones. Anyway, the greatest challenge is to use the Model Checker (tool) to systematically create such test cases.

However, none of the previous approaches neither explicitly explore Specification Patterns Systems (SPSs) [8, 20] to formalize the properties nor present a more rigorous and systematic empirical evaluation aiming at test case generation. The main reasoning behind SPSs is to provide guidelines to the professionals so that they can formalize their properties (requirements) via a set of patterns and patern scopes. Once identified a pattern and a pattern scope, the SPS provides a series of templates tailored to several logics such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and Timed Computation Tree Logic (TCTL) [3].

Moreover and despite their importance, formal methods have not been widely adopted in practice and major impediments are their complexity for understanding and notations. While formal, Statecharts are relatively simple to use and many projects in different domains have been relying on them to model the behavior of the SUT. Hence, in this paper we present a method, called **Hi**erarchy-based translation from Statecharts into **Mo**del Checking and **S**pecification Patterns Properties for **T**esting (HiMoST), to generate software test cases via Model Checking. Starting with a behavioral modeling in Harel's Statecharts [15], we propose a method to translate from Statecharts into a general structure based on the NuSMV Model Checker language, and we formalize CTL properties by means of the SPS proposed by Dwyer et al. [8] and Combinatorial Interaction Testing (CIT) via the T-Tuple Reallocation (TTR) algorithm [5, 6]. We have par-



*SAST, September 18–19, 2017, Fortaleza, Brazil*



DOI: https://doi.org/10.1145/3128473.3128475

tially implemented HiMoST within the SOLIMVA tool [23]. We also present a cost-effectiveness evaluation (quasiexperiment) to compare four different patterns/pattern scopes. Results indicate that the Precedence Chain ($P$ precedes $S$, $T$) pattern with Global scope presents the best performance.

This paper is organized as follows. Section 2 presents our method, HiMoST. Section 3 details the planning and detailed description of the quasiexperiment. Results and analysis of the empirical evaluation are in Section 4. In Section 5, we present some relevant studies which use Model Checking to generate test cases. Section 6 shows the conclusions and future directions of this research.

## 2. MODEL CHECKING TO GENERATE TEST CASES

The HiMoST method is presented in this section where we detail the main algorithms to translate from a Harel's Statechart model into a general structure that is based on the language of the NuSMV Model Checker. Having this structure, it is then possible to generate the NuSMV code itself. As we will mention later, although we focused on a specific Model Checker, this structure is general enough to be adapted to other tools. By running NuSMV, the $TS$ is then generated. We also show how we suggest the formalization of properties in CTL ($\Phi$) via a combined use of the SPS proposed by Dwyer et al. and the TTR algorithm. Figure 1 is a running example where the model shows two database operations: `add` and `retrieve`. An exception (`exc`) may be raised when this system is in operation and it is properly handled (`handled_exc`) by the SUT. Note that we have two events not shown in the figure: $\lambda_1$ which make the transitions from $T11$ to $T12$, and $\lambda_2$ which allows the change from $T21$ to $T22$. These are events for processing the operations. Moreover, shallow history (H) is present in the model.

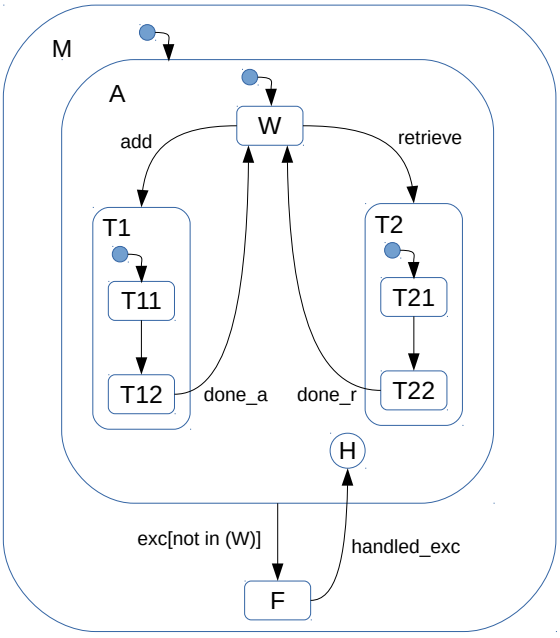


**Figure 1: Statetchart model: database operations**

Our translation approach basically deals with all features of Harel's Statecharts: XOR and AND (parallel) states, events, shallow and deep history, hierarchy. However, in HiMoST, the main characteristic which drives the translation process is hierarchy. We chose hierarchy because of the great potential that Statecharts have to model complex systems or Systems-of-Systems (SoS) via modularization. The main algorithm of the translation is shown in Algorithm 1.

**Algorithm 1** The HiMoST method: main algorithm

**input:** $\Sigma = \{\sigma_i \mid i = 1 \cdots m\}$, $\Delta = \{\delta_i \mid i = 1 \cdots n\}$
**output:** $P$ = NuSMV structure

1: $VAR \leftarrow createVAR(\Sigma)$
2: $INIT \leftarrow createINIT(VAR)$
3: $NEXT \leftarrow createNEXT(\Sigma, \Delta, VAR)$
4: **return** $P \leftarrow (VAR \uplus INIT) \uplus NEXT$

In Algorithm 1, $\uplus$ is the disjoint union operator while $\Sigma$ and $\Delta$ characterize all states and transitions of the model, respectively. $\Sigma$ is a set of tuples where each tuple ($\sigma_i$) has several elements such as the name of a state ($n$), the type of state ($t$ = XOR, AND), if this is a state with history related (deep or shallow; $ht$), the hierarchy ($h$) related to this state, the set of outgoing events ($E$), and some others. The set $\Delta$ contains tuples due to the transitions where each tuple ($\delta_i$) is composed of elements such as the source ($s$) and destination ($d$) states, the event ($e$), and the hierarchy related to the transition ($h$). It is assumed that $\Sigma$ and $\Delta$ are somehow obtained based on the Statechart model. Output of Algorithm 1 is a structure based on NuSMV composed of the disjoint union of the sets $VAR$, $INIT$, and $NEXT$ which mean the declaration of variables, the initialization of variables and the transitions of the model, respectively.

The $createVAR$ procedure is presented in Algorithm 2. Note that the output is a set, $VAR$, where each element ($Var_i$) is indeed a set of three other sets. $EvDisp_h$ is a set which controls the firing (value $on$) or not firing (value $off$) of events within each hierarchy level. $States_h$ is indeed another set with all states within a certain hierarchy level (note $\bigcup$ in line 7). On the other hand, $Events_h$ relate to the events within a certain hierarchy level (note $\bigcup$ in line 8). If we have parallelism (AND states), HiMoST creates sets $EvDisp_h$, $States_h$, and $Events_h$ for each state of the AND state within a hierarchy level. In Figure 1, the set of Hierarchy, $Hi$, contains the following elements: $\{M, A, T1, T2\}$. Then, for all $\sigma_i \subset \Sigma$, the algorithm verifies whether the hierarchy in $\sigma_i$ matches some hierarchy $h \in Hi$ (see line 6; the upper script $h$ means the hierarchy element of tuple $\sigma_i$). If this is so, sets $States_h$ and $Events_h$ are created with the respective states and events. For instance, when $h = A \in Hi$, then $States_A = \{W, T1, T2\}$ because these are the states within the hierarchy level $A$. On the other hand, $Events_A = \{add, retrieve, exc\}$. Here, we must note that $exc$ also relates to hierarchy $A$ because it is possible to leave state $A$ if an exception is raised when $T1$ or $T2$ are the active states of $A$.

Algorithm 3 presents the $createINIT$ procedure. The goal is to initialize all sets (variables). At the beginning, $createINIT$ identifies the state ($s_1$) which has an event that can be fired in the first place (line 2). In Figure 1, $s_1 = W$ because this is the initial state of $A$ which has events that can be fired. Hence, the corresponding event

**Algorithm 2** The *createVAR* procedure

**input:** $\Sigma = \{\sigma_i \quad | \quad i = 1 \cdots m\}$
**output:** $VAR = \{Var_i \quad | \quad i = 1 \cdots h\}$

1: $initializeAllSets()$
2: $Hi \leftarrow identifyHierarchy(\Sigma)$
3: **for all** $h \in Hi$ **do**
4: $\quad EvDisp_h \leftarrow \{on\} \cup \{off\}$
5: $\quad$ **for all** $\sigma_i \subset \Sigma$ **do**
6: $\quad\quad$ **if** $\sigma_i^h = h$ **then**
7: $\quad\quad\quad States_h \leftarrow \bigcup \sigma_i^s$
8: $\quad\quad\quad Events_h \leftarrow \bigcup \sigma_i^E$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad Var_i \leftarrow EvDisp_h \cup States_h \cup Events_h$
12: $\quad VAR \leftarrow VAR \cup Var_i$
13: **end for**
14: **return** $VAR$

dispatcher ($EvDisp_h$) is initialized to *on* and all others are initially defined as $off$ (lines 4-8). Thus, initially, $Init_1^{EvDisp_M} = off$, $Init_2^{EvDisp_A} = on$, $Init_3^{EvDisp_{T1}} = off$, and $Init_4^{EvDisp_{T2}} = off$. The initial values of states ($States_h$) and events ($Events_h$) are randomly selected. Note that $|INIT| = |VAR|$ but each element $Init_i^{EvDisp_h}$, $Init_i^{States_h}$, and $Init_i^{Events_h}$ is really a single element and not a set as it is in $Var_i$.

**Algorithm 3** The *createINIT* procedure

**input:** $VAR = \{Var_i \quad | \quad i = 1 \cdots h\}$
**output:** $INIT = \{Init_i \quad | \quad i = 1 \cdots h\}$

1: $initializeAllSets()$
2: $s_1 \leftarrow identifyFirstState(VAR)$
3: **for all** $Var_i \subset VAR$ **do**
4: $\quad$ **if** $Var_i^{States_h^s} = s_1$ **then**
5: $\quad\quad Init_i^{EvDisp_h} \leftarrow on$
6: $\quad$ **else**
7: $\quad\quad Init_i^{EvDisp_h} \leftarrow off$
8: $\quad$ **end if**
9: $\quad Init_i^{States_h} \leftarrow random(Var_i^{States_h})$
10: $\quad Init_i^{Events_h} \leftarrow random(Var_i^{Events_h})$
11: $\quad Init_i \leftarrow Init_i^{EvDisp_h} \cup Init_i^{States_h} \cup Init_i^{Events_h}$
12: $\quad INIT \leftarrow INIT \cup Init_i$
13: **end for**
14: **return** $INIT$

The last step of our method refers to the definition of the transitions (set $NEXT$) in the NuSMV structure. Algorithm 4 presents the *createNEXT* procedure. Inputs are the sets $\Sigma$, $\Delta$, and $VAR$ which have already been defined. Note that we work basically with the same reasoning which has been presented for the creation of $VAR$. In other words, we define sets of tuples for each event dispatcher within a hierarchy ($EVD_h$), state transitions ($ST_h$), and event changes ($EV_h$). Each element in $EVD_h$, $ST_h$, and $EV_h$ is a tuple of the form $\langle evd, s, e, v \rangle$, where $evd$ is a event dispatcher, $s$ is a state, $e$ is a event, and $v$ is the next value of a certain variable. All these elements refer to the same hierarchy level, $h$.

The basic idea to interpret each of the tuple $\langle evd, s, e, v \rangle$

**Algorithm 4** The *createNEXT* procedure

**input:** $\Sigma, \Delta, VAR$
**output:** $NEXT = \{Next_i \quad | \quad i = 1 \cdots h\}$

1: $initializeAllSets()$
2: $Hi \leftarrow identifyHierarchy(\Sigma)$
3: **for all** $h \in Hi$ **do**
4: $\quad [Da, Dd] \leftarrow identifyActivateDeactivateDisp(VAR)$
5: $\quad$ **for all** $da \in Da$ **do**
6: $\quad\quad EVD_h \leftarrow EVD_h \cup \langle Var_i^{EvDisp_h=on}, -, da, on \rangle$
7: $\quad$ **end for**
8: $\quad$ **for all** $dd \in Dd$ **do**
9: $\quad\quad EVD_h \leftarrow EVD_h \cup \langle Var_i^{EvDisp_h=on}, -, dd, off \rangle$
10: $\quad$ **end for**
11: $\quad St \leftarrow identifyStateTransitions(\Delta)$
12: $\quad$ **for all** $st \in St$ **do**
13: $\quad\quad ST_h \leftarrow ST_h \cup \langle Var_i^{EvDisp_h=on}, -, st, \delta_i^d \rangle$
14: $\quad$ **end for**
15: $\quad Ec \leftarrow identifyEventChanges(\Delta)$
16: $\quad$ **for all** $ec \in Ec$ **do**
17: $\quad\quad EV_h \leftarrow EV_h \cup \langle Var_i^{EvDisp_h=on}, -, ec, next(\delta_i^e) \rangle$
18: $\quad$ **end for**
19: $\quad Next_i \leftarrow EVD_h \cup ST_h \cup EV_h$
20: $\quad NEXT \leftarrow NEXT \cup Next_i$
21: **end for**
22: **return** $NEXT$

when generating the next statements in the NuSMV code is as follows:

$$evd \wedge s \wedge e \rightarrow v$$

In other words, we create a next statement where we have a predicate $evd \wedge s \wedge e$ that will result in the next value of a variable ($v$) when it evaluates to true. We should remark that we then create a standardized NuSMV code in this way.

The *identifyActivateDeactiveDisp* procedure (line 4) aims at identifying which events activate or deactivate a certain dispatcher within a hierarchy level. Thus, they show which events, $e$, make it possible to enter or go back to a certain hierarchy level so that events of such hierarchy may be fired in consequent steps. The output are two sets, one with the events to activate ($Da$) and other with events to deactivate ($Dd$) the dispatchers. Let us consider Figure 1 and hierarchy $T1$ ($h = T1$). Hence, $Da_{T1} = \{add, handle\_exc\}$ because from state $W$ the system goes to $T1$ via the event *add*, and goes back to $T1$ when coming back from state $F$ after handling an exception (*handled_exc*) if the the system was in $T11$ or $T12$ when the exception happened. This is due to the shallow history (H) in the model. Precisely, the system returns to $T11$, the initial state of $T1$. Likewise, $Dd_{T1} = \{done\_a, exc\}$ because these are the events that make the system leaves hierarchy $T1$.

After $Da$ and $Dd$ are filled, each tuple in $EVD_h$ can then be created. The tuples are of the form $\langle Var_i^{EvDisp_h=on}, -, da, on \rangle$ or $\langle Var_i^{EvDisp_h=on}, -, dd, off \rangle$. Important to mention that all values in the set $EvDisp_h$ are considered *on* within the predicates. The dash "-" means that the state contribution to the predicate ($s$) is not present. In fact, states are just considered in some next statement predicates in order to deal with very specific situations, e.g. history (deep, shallow). The most common predicate form includes only $evd$ and $e$. Due to space cons-

traints we decided not to show the $createNEXT$ procedure in too detail by choosing to describe it in its most common form.

Consider $h = M$ in Figure 1. One element in $EVD_M$ is $\langle Var_3^{EvDisp_{T1}=on}, -, exc, on\rangle$. Translating this tuple into a next statement predicate for the variable $Var_1^{EvDisp_M}$ this means: if the current active event dispatcher is $T1$ and the event to be fired within $T1$ is $exc$ then the next value of (selected element from) $Var_1^{EvDisp_M}$ is $on$. More specifically, if the system is in $T1$ ($T11$ or $T12$) and an exception happens ($exc$), the model leaves hierarchies $T1$, $A$ and the next active hierarchy will be $M$ ($on$).

The $identifyStateTransitions$ procedure (line 11) takes the set of transitions ($\Delta$) as input and just searches this set in order to find updated values for variables of the kind $Var_i^{States_h}$. The set $St$ consists of mappings $e \mapsto v$ saying which will be the next state ($v$) if $e$ is fired. Hence, the new value, $v$, of the variable is $\delta_i^d \in \delta_i \subset \Delta$, i.e. it is the destination state of the set $\delta_i$. Consider $h = T1$ in Figure 1. One element in $ST_{T1}$ is $\langle Var_3^{EvDisp_{T1}=on}, -, \lambda_1, T12\rangle$. Translating this tuple into a next statement predicate for the variable $Var_3^{States_{T1}}$ it means the transition $T11 \mapsto \lambda_1 \mapsto T12$ in the original Statechart model.

To identify event changes ($identifyEventChanges$ procedure in line 15), the set of transitions ($\Delta$) is also taken as input. Each element in $Ec$ is a mapping $e_i \mapsto e_j$ stating which will be the next event to be fired ($e_j$) considering the current event ($e_i$). In this case, it is possible that $e_j$ is indeed a set because two events may be able to be fired in a certain state $s$. For instance, if $h = A$ then one element in $EV_A$ is $\langle Var_3^{EvDisp_{T1}=on}, -, done\_a, \{add, retrieve\}\rangle$. After processing an $add$ operation ($done\_a$), events $add$ or $retrieve$ may be fired in state $W$. The Model Checker chooses it non-deterministically.

Considering the Statechart model in Figure 1, the set of variables ($VAR$) is made of 4 sets ($Var_i$) which in turn each $Var_i$ is the union of 3 other sets: $EvDisp_h$, $States_h$, and $Events_h$. In total, we have 12 sets which will be 12 variables in the NuSMV code (after the implementation step). Each one of these 12 variables has an associated next statement and, thus, $|NEXT| = 12$.

## 2.1 Implementation

As we have just presented, the output of the HiMoST method is a NuSMV structure, the set $P$, where we have three main sets: $VAR$, $INIT$, and $NEXT$. To translate the sets $VAR$, $INIT$ and $NEXT$ into NuSMV variables, their initial values, and next statements (transitions) respectively, we need to go through these sets to create the corresponding code. For the next statements, we follow the predicate based on tuples that we have described in the last section.

We have partially implemented the solutions proposed in the HiMoST method in the SOLIMVA tool [24]. Moreover, SOLIMVA creates the input sets, $\Sigma$ and $\Delta$, for the HiMoST method by reading a State Chart XML (SCXML) file which our tool itself generates based on the diagrammatic representation of the Statechart model defined in the Graphical User Interface (GUI) of SOLIMVA.

Despite the fact that this translation has been done for NuSMV, we also envisage that transformation into other Model Checkers, such as SPIN, is possible since the set $P$ can be regarded as a general structure. As we have already pointed out, the model in Figure 1 derive 12 sets (in set $VAR$). In NuSMV code, it means that we will have 12 enumerated variables in the `VAR` section and 12 `next` statements in the code. Hence, we may think in 12 Promela processes (`proctype`) where the reasoning of transitions (in set $NEXT$) is implemented via `do` statements, process communications via channels or global (shared) variables, and enumerations can be addressed via message types (`mtype`).

## 2.2 Properties Formalization and Test Case Generation

The other contribution of our method is a systematic way to formalize properties via an SPS [8] and CIT based on the TTR algorithm [5, 6]. We believe that this is an important step for practical purposes because professionals demand guidelines to proper apply a theory to their software projects. The systematic manner to formalize CTL properties is presented in Algorithm 5 included with the steps to generate the test suite, $T$.

**Algorithm 5** Properties formalization and test case generation

**input:** $P$ = NuSMV structure, $C$ = NuSMV code
**output:** $T$ = Test Suite

1: Consider all events, $e$, in the original Statechart model by inspecting $Var_i^{Events_h} \subset VAR \subset P$
2: Split equally all such events in three sets: $S_1$, $S_2$, $S_3$
3: Run the TTR algorithm considering sets $S_1$, $S_2$, $S_3$, and $strength = 2$. A Mixed-value Covering Array, $M_{|m|\times 3}$, is the TTR output
4: Define the SPS Pattern and Scope set, $\Psi$
5: **for all** $m \subset M$ **do**
6: **for all** $\psi \in \Psi$ **do**
7: Formalize a CTL property, $\Phi$, according to $\psi$ by replacing the formulas with the elements of $m$
8: **end for**
9: **end for**
10: Run the Model Checker with all $\Phi$ against $C$
11: Consider the counterexamples whose sizes are greater than a threshold, $\tau$
12: Generate an initial test suite, $T_i$, based on such counterexamples
13: Eliminate possible redundant test cases in $T_i$, and generate the final test suite, $T$
14: **return** $T$

First, we should remark that we propose only the events ($e \in Var_i^{Events_h}$) of the original Statechart model to formalize the properties. Other studies generate properties taken into account a combination of different types of variables or even sets that are created due to certain coverage criteria [17]. As events are the basic elements to stimulate the SUT, we decided to consider only events in our formulas with no corresponding use of other variables (sets). We believe that this solution simplifies the generation of formulas because we need to focus only on a combination of events to generate formulas to force the derivation of counterexamples.

Even so, the number of events may produce a great number of combinations and this may be infeasible in practice. Thus, we suggest using a CIT algorithm, TTR, to avoid such an excessive combinations of events. These methods have proven to be suitable for software testing by generating lower cost test suites [6]. Then, we equally divide all

the events in 3 sets, $S_1$, $S_2$ and $S_3$ (line 2), and run the TTR algorithm considering these sets and the degree of interaction (*strength*) equals to 2 (line 3). TTR outcome is a Mixed-value Covering Array, $M_{|m|\times 3}$. Let us consider Figure 1 again. Hence, we have $S_1 = \{exc, handled\_exc, add\}$, $S_2 = \{retrieve, \lambda_1, done\_a\}$, $S_3 = \{\lambda_2, done\_r\}$. By running TTR, we have the results presented in Table 1.

**Table 1: TTR outcome:** $M$

| m | $S_1$ | $S_2$ | $S_3$ |
|---|---|---|---|
| 1 | $exc$ | $retrieve$ | $\lambda_2$ |
| 2 | $exc$ | $\lambda_1$ | $\lambda_2$ |
| 3 | $exc$ | $done\_a$ | $done\_r$ |
| 4 | $handled\_exc$ | $retrieve$ | $\lambda_2$ |
| 5 | $handled\_exc$ | $\lambda_1$ | $done\_r$ |
| 6 | $handled\_exc$ | $done\_a$ | $\lambda_2$ |
| 7 | $add$ | $retrieve$ | $done\_r$ |
| 8 | $add$ | $\lambda_1$ | $\lambda_2$ |
| 9 | $add$ | $done\_a$ | $\lambda_2$ |

SPSs are very important for practical purposes. They give formula templates once a pattern and pattern scope are identified by a professional based on the requirements of the SUT. Currently, there are 9 patterns and 5 pattern scopes for CTL based on [8]. We suggest the following combination of pattern/patern scope as shown below:

1. Absence/Global (ABS). CTL: $\forall\Box(\neg P)$;
2. Response Chain ($S$, $T$ responds to $P$)/Global (REC). CTL: $\forall\Box(P \rightarrow \forall\Diamond(S \wedge \forall \bigcirc (\forall\Diamond(T))))$;
3. Precedence Chain ($P$ precedes $S$, $T$)/Global (PC1). CTL: $\neg\exists[\neg P \cup (S \wedge \neg P \wedge \exists \bigcirc (\exists\Diamond(T)))]$;
4. Precedence Chain ($S$, $T$ precedes $P$)/Global (PC2). CTL: $\neg\exists[\neg S \cup P] \wedge \neg\exists[\neg P \cup (S \wedge \neg P \wedge \exists \bigcirc (\exists[\neg T \cup (P \wedge \neg T)]))]$.

In the above CTL formulas, we have the path quantifiers for all paths ($\forall$) and for some path ($\exists$). Moreover, there are the temporal modalities always ($\Box$), eventually ($\Diamond$), next ($\bigcirc$), and until ($\cup$). Of course, a test designer may choose only one or all of these patterns/pattern scopes or even use another combination. But, we believe that these four suggestions are suitable due to the chain of events that they have related.

From lines 5 to 9 in Algorithm 5, we show how we mix SPS with CIT. Let us assume that we selected only Response Chain ($S$, $T$ responds to $P$)/Global, i.e. $\Psi = \{REC\}$. As we have $|m| = 9$ in Table 1, for each row $m$ we replace the formulas within the SPS template (in this case $P$, $S$ and $T$) with the corresponding elements of $m$. Hence, for $m = 3$ we have:

$$\forall\Box(exc \rightarrow \forall\Diamond(done\_a \wedge \forall \bigcirc (\forall\Diamond(done\_r)))).$$

At first, our test suite has 9 test cases ($|m| = 9$). But not all CTL formulas will produce a counterexample. In addition, some counterexamples may have only a single (the initial state) or few states. Hence, the threshold $\tau$ (line 11) serves to discard very short counterexamples which, in practice, there is no sense to execute. Finally, it is likely that some counterexamples are precisely equal. Then, we need to eliminate possible redundant test cases in order to get the final test suite, $T$.

We should emphasize that there is a certain randomness in defining the sets $S_1$, $S_2$, $S_3$ and, consequently, to generate the CTL formulas. Thus, some CTL properties may seem inadequate because they demand a certain sequence of stimuli to the SUT that are not in accordance with the requirements of the system. But we need to recall that the idea is to force the Model Checker to generate counterexamples. Hence, this randomness is interesting to precisely show that the SUT does not present some "no sense" behaviors. The conclusion is that a test suite generated in this way may be useful not only for conformance testing but also for robustness testing.

Considering the 4 patterns/pattern scopes we propose, $\tau = 2$ (i.e. size of counterexamples more than 2 to consider as a test case), elimination of redundant test cases, and Figure 1, the test suite $T$ according to the HiMoST method is as shown bellow:

$$\begin{aligned} T = \{&[add, \lambda_1, done\_a, exc],\\ &[add, \lambda_1, exc, handled\_exc, \lambda_1, exc, handled\_exc],\\ &[add, \lambda_1, done\_a, add],\\ &[add, \lambda_1, exc, handled\_exc, \lambda_1],\\ &[add, \lambda_1, done\_a, retrieve, \lambda_2],\\ &[add, \lambda_1, done\_a, retrieve, \lambda_2, done\_r],\\ &[add, \lambda_1, done\_a, retrieve]\}. \end{aligned}$$

Note that 7 test cases were generated (each test case is enclosed by $[\cdots]$). In this work, we define a test case ($tc$) as: $tc = [ts_i \quad | \quad i \in \mathbb{N}\backslash\{0\}]$, where $tc$ = test case, and $ts_i$ = test step $i$. A test step, $ts_i$, is an atomic activity to prepare or stimulate the SUT. The stimulus contains the test input data and, possibly, the expected results (in some cases, there are no explicit expected results). In other words, a test case is a sequence (repetitions are allowed and order matters) of test steps.

It is important to stress that the first 4 test cases were due to Response Chain ($S$, $T$ responds to $P$)/Global (REC), and the last 3 due to Precedence Chain ($P$ precedes $S$, $T$)/Global (PC1). The other two patterns generated no test case.

# 3. MULTI-OBJECTIVE EXPERIMENTAL EVALUATION

In this section, we present the description of a multi-objetive empirical evaluation. In fact, this empirical evaluation is classified as a quasiexperiment.

## 3.1 Definition

This multi-objective empirical evaluation aims at assessing together two characteristics, cost and effectiveness, of test suites generated via each one of the patterns/pattern scopes presented in Section 2: ABS, REC, PC1, and PC2. Hence, rather than using all of these patterns/pattern scopes to derive a unique test suite, we want to evaluate which of these patterns has better performance, with respect to cost and effectiveness, if we consider them alone to generate a test suite.

Cost is defined as the total number of test steps ($\#ts$) of all test cases of a test suite. This is a more realistic definition of cost because one test case, $tc_1$, might have associated a few test steps, let us say 5, and, for instance, a second test case, $tc_2$, might be composed of 100 test steps. We define effectiveness in accordance with three perspectives.

First, we examine the proportion of transitions covered in the original Statecharts developed for a sample (case study). Second, we examine the coverage of instructions, and finally the coverage of branches by running the test suite against the code of the sample.

## 3.2 Context and Research Question

The experiment was conducted by the researchers who defined it. We used the implementation of the TTR algorithm [5, 6] to create the matrix $M$, and NuSMV 2.6.0 to accomplish Model Checking for test case generation. All samples (case studies) were developed in Java. Hence, we verified coverage of instructions and branches via EclEmma.

Our set of samples is composed of four case studies as detailed below:

1. Simple SWPDC. This is a simplified version of the Software for the Payload Data Handling Computer (SWPDC) [23, 24]. It is a space application software product developed at the *Instituto Nacional de Pesquisas Espaciais* (INPE - National Institute for Space Research). The main goal of this project was to outsource the development of software embedded in satellite payload;
2. Train Controller. This software implements the main operations to safely transport train passengers. It simulates the operations of the doors of a train;
3. Digital Library. With this product, it is possible to build a record of books and create a Library, by adding new books, saving them, etc.;
4. TTR. The implementation of the TTR algorithm itself was used as a case study.

The Research Question (RQ) we want to answer is shown below:

**RQ_1** - What is the best approach regarding cost-effectiveness for test case generation considering these four patterns/pattern scopes within the HiMoST method: ABS, REC, PC1, and PC2?

## 3.3 Variables

With respect to the independent variables, the ones which can be manipulated or controlled during the process of trial and define the causes of the hypotheses, we considered the selected patterns/pattern scopes. The dependent variables, where we can perceive the result of manipulation of the independent variables, are the number of test steps due to each test suite, the proportion of transitions covered in the original Statechart model, the coverage of instructions and coverage of branches by running the test suite against the code of the sample.

## 3.4 Description of the Experiment

We started by creating one Statechart model for a sample (case study). This model was created in the GUI of the SOLIMVA tool. By running SOLIMVA, an SCXML file was generated which represents this model, and our tool reads such an SCXML file and creates the sets $\Sigma$ and $\Delta$ presented in Section 2. After that, the implementation of the HiMoST method within the SOLIMVA tool translated the Statechart model into the NuSMV code. As we have already mentioned, this translation process is not fully implemented yet so that we needed to make some manual adjustments in the NuSMV code before going on.

After that, we followed Algorithm 5 (Section 2.2) for properties formalization and test case generation. Then, we created a test suite due to each pattern/pattern scope and measured the dependent variables. We selected an initial sample and we simply recorded the total number of test steps due to each test suite: $T_{ABS}$, $T_{REC}$, $T_{PC1}$, and $T_{PC2}$. Regarding effectiveness, we visually inspected which transitions in the Statechart model were covered based on each test suite. Hence, we calculated the proportion of covered transitions taken into account the total number of transitions of the Statechart model. Covered instructions and branches were achieved via EclEmma after running each test suite.

We repeated these previous steps for all the other samples. Then, we needed a consolidated value to represent the cost and effectiveness of a test suite. For cost, we considered the average number of test steps per test case ($\overline{ts}$) taking into account all 4 samples, and we normalized this mean value based on the selected threshold ($\tau$). Thus, if $\tau = 2$ then the normalized result, $\overline{ts_n}$, is: $\overline{ts_n} = ((\tau + 1)/\overline{ts}) \times 100$ (we multiplied by 100 to have a percentage between 0 and 100). Hence, as close $\overline{ts}$ to $\tau + 1$, better the cost (minimum cost).

Effectiveness regarding the proportion of transitions covered in the original Statechart model, the coverage of instructions and coverage of branches is simply the average value taken into account the 4 samples. Hence, each pattern/pattern scope derives a point, $p$, in a 4-dimensional space of the form:

$$(\overline{ts_n}, \overline{tr}, \overline{in}, \overline{br}),$$

where $\overline{ts_n}$ is the normalized average number of test steps per test case, $\overline{tr}$ is the average value of the proportion of transitions covered in the original Statechart model, $\overline{in}$ is the average value of covered instructions, and $\overline{br}$ is the average value of covered branches.

To know which approach is more interesting in terms of cost-effectiveness, we need to observe which pattern/pattern scope generates a smaller test suite (smaller average number of test steps per test case) and has greatest effectiveness (greatest coverage). We achieved this conclusion based on the Euclidean distance, $d(o, p)$, between an optimum point, $o$, and each $p$ from each pattern/pattern scope. This optimum point is:

$$o = (\alpha \times 100, \beta \times 100, \beta \times 100, \beta \times 100).$$

Note that the first dimension relates to cost ($\overline{ts_n}$ in $p$) and the other three dimensions are related to effectiveness ($\overline{tr}, \overline{in}, \overline{br}$ in $p$). Besides, $\alpha$ and $\beta$ are weights. In other words, if we want that cost and effectiveness contribute equally then $\alpha = \beta = 1$. But, if we decide that effectiveness must contribute more in the cost-effectiveness outcome then we must consider $\beta > \alpha$. In this case, we must also multiply each dimension in $p$ which relates to effectiveness ($\overline{tr}, \overline{in}, \overline{br}$) by $\beta$. Giving more weight to effectiveness seems a good alternative because, eventually, very short test cases may not be good to detect software defects. Thus, a smaller Euclidean distance between $o$ and $p$ means a better solution with respect to cost-effectiveness.

We should state that for the ABS partern/pattern scope, we replaced $P$ by a disjunction ($\vee$) of three formulas (one

contribution from each set $S_1$, $S_2$, $S_3$). In this situation, the CTL formula template is $\forall\Box(\neg(P_1 \vee P_2 \vee P_3))$. One last remark is the way we executed the test suites. For SWPDC and Train Controller, the level is unit testing. Therefore, we created JUnit test suites and ran them against these samples. For Digital Library and TTR, the level is system testing. Thus, test steps are indeed interactions with GUI elements (Digital Library) or input data via command line interface (TTR).

## 3.5 Validity

The conclusion validity relates to how sure we are that the treatment we used in an experiment is really related to the actual observed outcome. One of the threats to the conclusion validity is the reliability of the measures. Our results are associated with four tools: SOLIMVA in which HiMoST has been partially implemented, TTR, NuSMV which automatically created the test cases, and EclEmma responsible to automatically provide instruction and branch coverages. The only measure obtained in a totally manual way was the one of the transitions covered in the original Statechart model. Hovewer, the Statechart models are not very big (the most complex has 10 states and 12 transitions) and hence we were able to obtain this measure reliably by visual inspection. We believe that replication of this study by other researchers will produce similar results and our study has a high conclusion validity.

For the internal validity, we need to be sure whether other factors have not caused the outcome, factors that have not been controlled or measured. There are many threats to internal validity such as testing effects (measuring the subjects repeatedly), history (external events of the experiment may influence the responses of the subjects, e.g. interruption of the treatment), maturation (participants might mature during the study), selection bias (differences between groups), etc. The participants of our experiment were behavioral models and source code, and thus we neither had any human/nature/social factor nor unanticipated events to interrupt the collection of the measures once started to pose an internal validity. Hence, we claim that our quasiexperiment has a high internal validity.

In the construct validity, the goal is to ensure that the treatment reflects the construction of the cause, and the result the construction of the effect. This is also high because we selected four classical CTL patterns/pattern scopes to assess the cause, and the results, supported by the decision-making procedure via Euclidean distance to an optimum point, clearly provided the basis for the decision to be made between the four strategies.

However, we have a threat to external validity, specifically a threat to population which refers to how significant is the set of samples of the population used in the experiment. We chose four samples with one Statechart model per sample. Hence, we need more significant samples in order to generalize the results. But, we believe that the results of this quasiexperiment are interesting where we can perceive some repeated outcomes when assessing a particular test suite (more details in Section 4).

# 4. RESULTS AND ANALYSIS

In this section, we present the results of the multi-objective evaluation. Table 2 shows the results related to cost where column #tc has the total number of test cases due to a pattern/pattern scope, #ts presents the total number of test steps due to a pattern/pattern scope, rep is the number of repeated test cases, and disc shows the number of discarded test cases. A test case was discarded if its length (number of test steps) was less than or equal to the threshold $\tau$. We chose $\tau = 2$ and hence a test case must have length at least 3 to be considered in the final test suite, $T$. Also note the average number of test steps per test case ($\overline{ts}$) and its normalized value ($\overline{ts_n}$).

The TTR algorithm generated 16, 9, 16, and 12 combinations for SWPDC, Train Controller, Digital Library and TTR, respectively. We then had 16, 9, 16, and 12 formalized CTL properties for each pattern/pattern scope. We noticed that ABS presented the better cost (greatest $\overline{ts_n}$) followed by REC and in the last position is PC1. To our surprise, PC2 generated no test case for all 4 samples. In fact, it did generate counterexamples (test cases) for all CTL properties for all 4 case studies but they were all discarded because these counterexamples had only 1 state (the initial state) and thus less than our $\tau$. Since PC2 is a Precedence Chain pattern as it is PC1, just changing the order of precedence of formulas, we did expect that some test cases were created as it happened when choosing PC1.

ABS also created a significant number of test cases which were discarded. For instance, for the Train Controller, 1 single test case was in the final $T_{ABS}$ while we had to ged rid of 7 test cases. On the other hand, REC showed a proportionally huge number of repeated test cases. For the Digital Library, $T_{REC}$ has only 4 test cases while 12 test cases were repeated.

Huge number of discarded or repeated test cases is an indication that, in accordance with the strategy proposed in the HiMoST method, these patterns can contribute, albeit marginally, to increasing the cost of the testing process as a whole. Even if it is possible to automate this verification of test cases that are repeated or need to be discarded, this excessive proportion of useless test cases tends to increase the demand for processing to perform such checks. Although the cost of the testing process is deeply influenced by the test execution activity, meaning that the smaller the number of test steps of a suite the better (in this respect, ABS is considered the best alternative), the cost to generate the final test suites can not be completely neglected.

PC1 had more satisfied ($true$) CTL formulas. Let us consider the sample SWPDC in Table 2. The final $T_{PC1}$ is composed of 5 test cases, and we have just one repeated test case. As 16 CTL formulas were verified, the conclusion is that 10 of such formulas hold. Thus, it is not necessary to perform any processing/verification of repeated or discarded test cases for such formulas that were satisfied.

In Table 3, we see the results with respect to effectiveness. In this table, column tr is the proportion of transitions covered in the original Statechart model, in means the percentage of covered instructions of the source code, and br is the percentage of covered branches of the source code. The average of the covered transitions ($\overline{tr}$), instructions ($\overline{in}$), and branches ($\overline{br}$) are in the last row of Table 3.

Considering the three measures of effectiveness ($\overline{tr}$, $\overline{in}$, $\overline{br}$), we note that PC1 is the best solution followed by REC, and ABS is the worst. This is precisely the inverse order of the cost analysis. In some samples (SWPDC, Digital Library), PC1 even achieved 100% of covered transitions of the Statechart model.

Table 2: Cost results

| Cost | ABS | | | | REC | | | | PC1 | | | | PC2 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #tc | #ts | rep | disc | #tc | #ts | rep | disc | #tc | #ts | rep | disc | #tc | #ts | rep | disc |
| SWPDC | 4 | 13 | 0 | 12 | 4 | 34 | 10 | 0 | 5 | 51 | 1 | 0 | 0 | 0 | 0 | 16 |
| Train | 1 | 3 | 1 | 7 | 3 | 14 | 6 | 0 | 4 | 22 | 0 | 0 | 0 | 0 | 0 | 9 |
| Library | 3 | 10 | 3 | 10 | 4 | 30 | 12 | 0 | 4 | 40 | 2 | 0 | 0 | 0 | 0 | 16 |
| TTR | 3 | 11 | 3 | 6 | 4 | 20 | 8 | 0 | 3 | 24 | 0 | 0 | 0 | 0 | 0 | 12 |
| $\overline{ts}$ | | 3.36 | | | | 6.53 | | | | 8.56 | | | | - | | |
| $\overline{ts_n}$(%) | | 89.19 | | | | 45.92 | | | | 35.04 | | | | - | | |

Table 3: Effectiveness results

| Effectiveness | ABS | | | REC | | | PC1 | | |
|---|---|---|---|---|---|---|---|---|---|
| | tr (%) | in (%) | br (%) | tr (%) | in (%) | br (%) | tr (%) | in (%) | br (%) |
| SWPDC | 58.33 | 61 | 27 | 83.33 | 72 | 31 | 100 | 74 | 33 |
| Train | 33.33 | 50 | 22 | 44.44 | 58 | 33 | 88.89 | 87 | 56 |
| Library | 54.55 | 54 | 24 | 72.73 | 65 | 35 | 100 | 66 | 37 |
| TTR | 50 | 26 | 13 | 40 | 26 | 13 | 90 | 73 | 70 |
| Average | 49.05 | 47.75 | 21.5 | 60.13 | 55.25 | 28 | 94.72 | 75 | 49 |

However, we must emphasize that even PC1 did not not present a very high average coverage of instructions (75%) and especially branch coverage (49%) of the source code. This is explained by the Statechart model. In general, such models have some normal behavior addressed and a few exception handling situations. Hence, to get higher coverage of the source code it is required more detailed situations in the Statechart models and particularly more events that are associated with non-normal behavior of the SUT.

In spite of these isolated evaluations, we are truly interested in obtaining a combined cost-effectiveness result as we have previously explained. Hence, we calculated the Euclidean distance between the optimum point, $o$, and each point $p$ due to the patterns. Table 4 shows the results for two situations. In the uniform case (row Equal), cost and effectiveness have the same weights. Hence, $\alpha = \beta = 1$ (see Section 3.4). In the second situation (row Double Effectiveness), we doubled the influence of effectiveness compared to cost, i.e. $\beta = 2$ and $\alpha = 1$.

Table 4: Cost-Effectiveness results: distances

| Cost-Effectiveness | ABS | REC | PC1 |
|---|---|---|---|
| Equal | 107.73 | 108.17 | 86.45 |
| Double Effectiveness | 214.64 | 195.01 | 131.28 |

Based on Table 4, we see that PC1 is better in all situations. When cost and effectiveness contribute equally, the Euclidean distance from the optimum point, $o$, to ABS is 24.62% greater than the distance from $o$ to PC1. And from $o$ to REC, the distance is 25.12% greater than from $o$ to PC1. However, we can see that basically there is no cost-effectiveness difference if we compare ABS with REC with a slight advantage for ABS.

If we double the contribution of effectiveness, again PC1 is the best option where the Euclidean distances from $o$ to ABS and to REC are 63.5% and 48.55% greater than the distance from $o$ to PC1, respectively. However, here REC performs better than ABS where the distance from $o$ to the latter is 10.07% greater than from $o$ to the former.

The overall conclusion of this quasiexperiment is that the Precedence Chain ($P$ precedes $S$, $T$) pattern with Global scope (PC1) is the best test suite regarding cost-effectiveness. The Response Chain ($S$, $T$ responds to $P$) pattern with Global scope (REC) is better than the Absence pattern with Global scope (ABS) only if we consider that effectiveness has a priority over cost. If they contribute equally, there is no difference between REC and ABS.

This conclusion is interesting because some previous works [12] basically use, although the authors did not explicitly mention, ABS to force the Model Checker to generate test cases (counterexamples). Note that ABS is a type of safety property which informally means that "something bad will never happen" [3]. Hence, the results presented in this work suggest that it is interesting to investigate other patterns/pattern scopes to obtain test suites with greater cost-effectiveness.

## 5. RELATED WORK

In [12], the authors divided the strategies to generate test cases via Model Checkers in two classes: coverage-based and mutation-based classes. In the coverage-based class, Model Checkers can be used to automatically create test suites which satisfy a certain coverage criteria (e.g. state, transitions, instructions, branches). They also subdivided this category into some subclasses. In specification language-specific coverage criteria, the concept of trap properties was proposed related to Software Cost Reduction (SCR) specifications [13]. Trap properties were created from SCR tables, and such properties derived a test suite to satisfy branch coverage for SCR specifications.

Other coverage-based subclass is coverage of general transition systems where a framework for test case generation based on specifications addressing structural coverage was defined regardless of any specific formalism [21, 22]. Thus, several coverage criteria similar to the structural ones for Control Flow Graphs (branch, Modified Condition/Decision Coverage (MC/DC), etc.) were presented.

Another subclass is control and data flow coverage crite-

ria [18, 16, 17, 10]. The same perspective of structural testing was used where control and data flow coverage criteria were considered. Extended Finite State Machines (EFSMs) [18], Data Flow Graphs [16], Statecharts [17], and Functional Block Diagram (FBD) [10] (see remarks below) were some models and CTL was the selected logic for most of the studies. However, the idea was not to use trap but rather to rely on **witness** properties. Roughly speaking, a witness property is a negation of a trap property. And most of the strategies defined CTL formulas to address the control and data flow coverage criteria.

One particular study related to control and data flow coverage which addressed real-time analysis is shown in [10]. The authors described a tool-supported approach that allows test case generation of programs written in FBD, a programming language suited to industrial application systems. They translated FBD programs into timed automata [3] and the UPPAAL Model Checker was used to generate test cases. Control flow criteria (condition coverage, MC/DC, etc.) were selected for test case generation. Also, the strategy is based on witness properties. One drawback of their proposal is that they must annotate the created model such that a condition describing a single test case can be formulated. This may demand a huge effort from users.

We can mention two main differences between our method and these previous approaches. First, the reasoning of our method is not driven by the coverage of any aspect of the original model of the SUT. In other words, we do not aim to cover state, transition, branch, all-definitions, all-uses, etc. of the models. We take all the events (input data) of the model and randomly combine them via a CIT algorithm and generate trap properties via specification patterns. Hence, coverage of the model (and the respective source code) is an effect and not a cause for us. Second, it is the simplicity to generate the formulas. For instance, some approaches [18, 16, 17] require the identification of def and use sets (data flow testing) to create the formulas while we just rely on the events that make the transitions happen. Other approaches [12] consider the expected result of the original model as a variable of the NuSMV code increasing the state space. We just consider the inputs (events) resulting in a smaller state space.

The second main class in [12] is based on mutation testing [2, 1, 11]. Basically, the authors suggested creating mutated models [2], $TS_m$, or mutated properties [2, 1], $\Phi_m$, and generating test cases. Despite its importance, mutation testing has two main issues. The huge number of mutants (models, properties) and determination of equivalent mutants may impose limitations for practical applications of these approaches.

The Polyglot tool allows the translation from a Statechart model into a common intermediate representation which is then translated into Java code that represents the structure of the model [4]. Polyglot is integrated with the Java Pathfinder verification toolset which can generate test cases for Java [25]. We believe that our translation proposal is more generic because we provide a structure which is composed of sets (variables, their initializations, and transitions of the model) and not particularly coupled with a given programming language as it seems to be the case of Polyglot. The authors mentioned they used specification patterns but it is not clear which pattern/pattern scope they relied on and which formal language (LTL, CTL,...) they selected.

Unit checking is a method that allows the symbolic verification of a unit of code and the derivation of test cases [14]. It combines principles of Model Checking and Theorem Proving and searches the paths of the flow chart of a program for possible executions that satisfy the specification in LTL. Although the authors claimed that most of the features were implemented, they did not show whether their method is adequate for several application domains. The case studies they presented look like scientific/mathematical computing programs.

To sum up, we can cite the following differences and in some cases advantages of our method, HiMoST, compared with these studies: (i) our transformation method, although creates a structure suitable to generate code for the NuSMV language, can be regarded as general enough. At the end, we generate sets that can be mapped to other Model Checkers and which are not specifically coupled with a certain programming language [4]; (ii) our method is not motivated to cover certain aspects of the original model as others [13, 21, 22, 18, 16, 17, 10]. To some extent, we use ideas from random testing to create sets of events that are input to a CIT algorithm. So we make use of specification patterns to formalize properties to force the creation of counterexamples. In addition, we believe that we propose a clear, relatively simple and systematic way to formalize CTL properties and generate test cases. Systematic procedures are very important to transfer the knowledge produced by academia to real world practice; and (iii) although some studies presented interesting empirical evaluations [16, 11, 10, 4], they did not do it via a rigorous evaluation as we shown in this paper (quasiexperiment). We also present a clear way to jointly consider cost and effectiveness in a multi-objective perspective.

# 6. CONCLUSIONS

This paper presented a novel method, HiMoST, where a formal verification method (Model Checking) helped another V&V process, i.e. software testing. HiMoST is a hierarchy-based translation strategy where starting with a behavioral modeling in Harel's Statecharts, it transforms such a model into a general structure based on the NuSMV language. Properties are formalized by joining CIT and specification patterns with some principles from random testing. The translation process which results in a general structure and the way we formalize the properties and create the final test suite, $T$, are important contributions of our research.

We also presented a rigorous cost-effectiveness evaluation (quasiexperiment) to compare four different patterns/pattern scopes. Results indicate that the Precedence Chain ($P$ precedes $S$, $T$) pattern with Global scope demonstrates the best performance not only if we consider cost and effectiveness contributing equally but also if we double the contribution of effectiveness. The Response Chain ($S$, $T$ responds to $P$) pattern with Global scope is better than the Absence pattern with Global scope only if we consider that effectiveness has a priority over cost. If they contribute equally, there is no difference between them.

We had to make some manual adjustments in the NuSMV code produced by the SOLIMVA tool. Hence, we need to finish the implementation of HiMoST within our tool. We can not generalize the results because the set of samples needs to be improved. Thus, we will develop a controlled experiment

with more samples in order we can tackle this threat to external validity. We also aim to develop another controlled experiment, with mutation testing to address effectiveness, comparing our method with other Model Checking-based test case generation approaches or even with studies that generate test cases based on Statechart test criteria.

## 7. ACKNOWLEDGMENTS

This research has been supported by *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq).

## 8. REFERENCES

[1] P. Ammann and P. Black. A specification-based coverage metric to evaluate test sets. In *Proc. 4th IEEE Int. Symp. High-Assurance Systems Eng.*, pages 239–248. IEEE Computer Society, 1999.

[2] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. IEEE Int. Conf. Formal Engineering Methods (ICFEM)*, pages 46–54. IEEE Computer Society, 1998.

[3] C. Baier and J.-P. Katoen. *Principles of model checking*. The MIT Press, Cambridge, MA, USA, 2008. 975 p.

[4] D. Balasubramanian, C. S. Pasareanu, G. Karsai, and M. R. Lowry. Polyglot: Systematic analysis for multiple statechart formalisms. In N. Piterman and S. A. Smolka, editors, *Tools and Algs. Construction and Analysis Syst.*, volume 7795 of *LNCS*, pages 523–529. Springer Berlin/Heidelberg, 2013.

[5] J. M. Balera and V. A. Santiago Júnior. T-tuple reallocation: An algorithm to create mixed-level covering arrays to support software test case generation. In O. Gervasi et al., editor, *Computational Science and Its Applications – ICCSA 2015*, volume 9158 of *LNCS*, pages 503–517. Springer International Publishing, 2015.

[6] J. M. Balera and V. A. Santiago Júnior. A controlled experiment for combinatorial testing. In *Proc. 1st Brazilian Symp. Systematic and Automated Software Testing (SAST)*, pages 1–10. ACM, 2016.

[7] J. Bowen, K. Bogdanov, J. A. Clark, M. Harman, R. Hierons, and P. Krause. FORTEST: formal methods and testing. In *Proc. 26th Annual Int. Computer Software and Applications Conference (COMPSAC)*, pages 91–101. IEEE, 2002.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 411–420. ACM, 1999.

[9] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Tools and algorithms for the construction and analysis of systems*, volume 1217 of *LNCS*, pages 384–398. Springer Berlin/Heidelberg, 1997.

[10] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2014.

[11] G. Fraser and F. Wotawa. Mutant minimization for model-checker based test-case generation. In *Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques - MUTATION*, pages 161–168. IEEE, 2007.

[12] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[13] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engineering Notes*, 24(6):146–162, 1999.

[14] E. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Aspects of Computing*, 17(2):201–221, 2005.

[15] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[16] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 232–242. IEEE, 2003.

[17] H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha. Automatic test generation from statecharts using model checking. Technical report, University of Pennsylvania, 2001.

[18] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In J.-P. Katoen and P. Stevens, editors, *Tools and Algs. Construction and Analysis Syst.*, volume 2280 of *LNCS*, pages 327–341. Springer Berlin/Heidelberg, 2002.

[19] M. Kadono, T. Tsuchiya, and T. Kikuno. Using the nusmv model checker for test generation from statecharts. In *Proc. 15th IEEE Pacific Rim Int. Symp. Dependable Computing*, pages 37–42. IEEE Computer Society, 2009.

[20] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 372–381. ACM, 2005.

[21] S. Rayadurgam and M. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. 8th Annual IEEE International Conf. and Workshop Eng. of Computer Based Systems*, pages 83–91. IEEE Computer Society, 2001.

[22] S. Rayadurgam and M. Heimdahl. Test-sequence generation from formal requirement models. In *Proc. 6th IEEE Int. Symp. High Assurance Systems Eng.*, pages 23–31. IEEE Computer Society, 2001.

[23] V. A. Santiago Júnior. *SOLIMVA: A methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications*. PhD thesis, Instituto Nacional de Pesquisas Espaciais (INPE), 2011. 264 p.

[24] V. A. Santiago Júnior and N. L. Vijaykumar. Generating model-based test cases from natural language requirements for space application software. *Software Quality Journal*, 20(1):77–143, 2012.

[25] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proc. 2004 ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, pages 97–107. ACM, 2004.