Verifying Real-Time Embedded Software by Means of Automated State-based Online Testing and the SPIN Model Checker—Application to RTEdge Models

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Applied Science

at

Ottawa-Carleton Institute for Software Engineering
Electrical and Computer, System and computer Engineering

Carleton University
Ottawa, Ontario

Abstract

Verifying a real time embedded application is very challenging especially since one has to consider timing requirements in addition to functional ones. Through online testing, the risks of failure in real time applications are reduced. During online state-based testing the generation and execution of test cases happens concurrently: test case generation uses information from a state-based test model in combination with observed execution behaviour. This thesis describes a practical online testing algorithm that is implemented in the state-based modeling tool RTEdge developed at Edgewater Computer Systems Inc. In addition, thanks to RTEdge's mechanism to map the state model to Promela, thereby allowing the use of the SPIN model checker, when additional coverage is needed after the online testing procedure, SPIN is used to generate additional test cases that will increase coverage. The Production Cell case study shows that our online testing algorithm produces a test suite that achieves high model coverage (specifically state and transition coverage), thus facilitating the automated verification of real-time embedded software.

Acknowledgements

All praise to Allah without his grace and blessing this work will not have been possible.

I would like to thank my highly inspiring and motivating supervisor Professor Yvan Labiche for his valuable advices, guidance, support, and various ideas. His great ideas have guided me to conduct my research properly. This thesis would not have completed without him. I would also like to thank Serban Gheorghe, Alvin Sim, and Patrick Lee from Edgewater Computer Systems Inc for their guidance as well as for providing necessary information regarding the project, and also for their support in completing the project.

Special thanks to my parents for their continuous love, encouragement and support. I would like to thank my husband, Anas, and my precious gifts from Allah, my three sons, Mohammad, Malik, and Majd for brightening up my life and being always there for me.

Table of Contents

List of Tables

List of Illustrations

Chapter 1 - **Introduction**

Specifying, designing, constructing, and verifying software systems is a challenge and doing so for embedded, real-time software is even harder because not only one has to account for functional requirements but one also has to account for non-functional requirements (especially time related ones). RTEdge is a collection of specification modeling, code generation, simulation and analysis tools that facilitate developing time critical real-time embedded applications, such as avionics, nuclear instrumentation and space applications (http://www.edgewater.ca/software-solutions).

Our objective in this thesis was to devise a procedure, as automated as possible, to create test cases from an RTEdge model using an executable version (simulation) of the model provided by the RTEdge's tool. Such test cases could then be executed against different simulation settings to evaluate impact of design decisions as well as on the target (deployment) platform.

We report on the results of this endeavor in this thesis. Specifically, we developed a black-box, online, feedback-driven random test case generation procedure. It is black-box because we only rely on the specification to derive test inputs (and not the implementation of the model), but also because we only use a small portion of the specification: in fact we only use the specification of signals that trigger behaviour in the set of communicating state machines in the RTEdge model because we felt any more detailed analysis of the model would be too difficult (e.g., because of pieces of C/C++ code it contains). It is online since we rely on RTEdge capabilities to simulate the specified application: as a result, test case creation occurs concurrently to the simulation of the test case execution. It is feedback driven because we regularly observe progress in terms of coverage of ele-

ments of the model in order to decide whether to stop or continue the construction of a test case.

1.1 Thesis Contributions

The objectives of our test framework were initially the following: (a) We wanted to derive test cases from an RTEdge model with an (model-driven testing) approach/algorithm as simple as possible; (b) We wanted to rely on the existing features of RTEdge to the maximum extent possible for testing purposes.

Our contributions are summarized below:

1. We developed a black-box, online, feedback-driven random test case generation procedure to automatically generate and execute test cases for RTEdge.
2. We used the SPIN model checker to generate additional test cases for the states/transitions when our random test case generation fails to achieve the user-stated coverage of the model required.
3. We used two selection criteria for the test framework generation which are state coverage and transition coverage.
4. We performed an experimentation to evaluate the test framework by developing two case studies, selecting the well-known Production Cell case study and an Elevator Control System case study.

1.2 Thesis Organization

The rest of the thesis is structured as follows. Chapter 2 provides background on state-based testing, online testing, Linear Temporal Logic (LTL) model checking, and the RTEdge platform we build upon. Chapter 3 discusses related work for state-based testing, online testing and the use of model checking for testing purposes. Chapter 4 introduces

the design and implementation of our test framework while Chapter 5 discusses the case studies. Chapter 6 concludes the thesis.

Chapter 2 - **Background**

In this chapter we discuss material that will be useful in one or more of the subsequent chapters. Specifically, we discuss state-based testing and associated criteria (section 2.1), we introduce the notions of online and offline testing (section 2.2), we present model checking from Linear Temporal Logic (section 2.3), and finally provide an overview of the RTEdge software development environment (section 2.4).

2.1 State-based testing and its selection criteria

State-Based Testing is about generating test cases from one or more state machines that model the system behaviour. The System Under Test (SUT) is tested with respect to how it reacts to different events and sequences of events; thus, state-based testing verifies if the states that are actually reached and the transitions that are actually triggered in the implementation conform to the expected events that can be simulated on the model. The automatic generation and execution of tests cases is possible from state machine(s); thus, verification efforts are reduced which is an advantage over other (more manual) verification techniques. State-based testing also implies the generation of input data and the evaluation of the test results collected from the SUT. A data associated with an event must be generated when such an event occurs in a test case. The event's data can be randomly generated from the set of possible values, or can be generated by using more complicated techniques such as a constraint solver [1]. Alternatively, search-based techniques, for example genetic algorithms, can be used for test data generation [2]. In practice, using every possible input data value and input sequence from a state machine, which is known as exhaustive testing [3], is not feasible. Therefore, it is necessary to use a selection criterion to extract a subset of all possible tests.

Coverage is a measure of how much a test suite satisfies the test requirements imposed by a selection criterion for a piece of software [4]. Different selection criteria can be used to select test cases from state machines such as (1) All-transitions, (2) All n-Transition Sequences, also sometimes referred to as All-transition k-tuples, (3) All-transition pairs, and (4) Round-trip path [3, 4].

All transition coverage is achieved if every transition in a state machine is exercised at least once by a test suite. This criterion guarantees that all states, events and actions are exercised. All n-Transition Sequences coverage requires all possible sequences of transitions of length n is exercised at least once. All transition-pairs coverage is All n-Transition Sequences coverage for n equals two. Also all transitions coverage is equivalent to all n-Transition Sequences coverage with n equals one. All round-trip coverage requires that all paths in a state machine that starts and ends with the same state must be covered. To cover all such paths, a transition tree is typically constructed by traversing the state model.

2.2 Online versus Offline testing

Test cases constructions methods can be categorized into offline and online depending on when the test cases are generated and executed. Offline testing is a testing technique where the complete test suite is generated first, then the test suite is executed [5]. Offline test generation is typically based on a model coverage criterion such [6, 7], on a test purpose [8], or on a fault model [9].

Online testing is a testing technique where the tests are generated and executed at the same time [5]. Online testing generates only a single test primitive from the test model at a time, then the SUT (or a simulation of its behavioural model) immediately executes on

the given test primitive. After that the test output produced by the SUT as well as its time of occurrence is checked against the specification. Next a new test primitive is generated and this repeats until the test script is decided to end the test depending on a termination condition (e.g., some coverage level of the test model).

The advantages of online testing include: since online testing may run for a long time, very long, complicated, and stressful test cases may be constructed and executed; online testing reduces the state-space explosion problem that is faced by many offline test generation tool because a very limited part of the state space needs to be stored at any point of time during test case construction; online testing allows non-determinism in real-time models, which gives a great flexibility during modeling.

One of the advantages of offline testing is that the generated test cases can be executed on different environment such as the development platform, the target platform … (instead of on a dedicated platform allowing model execution in the case of online testing). Also, the generated test cases are quicker to execute because all the specification constraints are resolved at test generation time, in addition, some essential guarantees for the test cases are easier to achieve at the time of test generation, e.g., that the specification is structurally covered. It is also possible to perform optimization technique on the test suite to minimize its size.

2.3 Linear Temporal Logic (LTL) model checking

Simple Promela Interpreter (SPIN) is a powerful verification system that supports the design and verification of distributed software systems [10]. SPIN verification models are mainly focused on proving the correctness of the process communications. Also, it provides program like annotations which are used for defining design choices unambiguous-

ly without the need to provide a concrete (source code) implementation of the corresponding behaviour. Also its concise notation helps to express general correctness requirements. SPIN accepts design specifications written in the PROMELA (a Process Meta Language) verification language [10], and it accepts correctness properties specified in Linear Temporal Logic (LTL) [11].

A model-checker like SPIN uses Kripke structures as a model formalism [12]. A Kripke structure K is a tuple (S, S_0, T, L), where S is the set of states, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, and L: $S \rightarrow 2^{AP}$ is the labeling function that maps each state to a set of atomic propositions that hold in this state. AP is the countable set of atomic propositions. A Kripke structure defines all the possible paths P = traces(K) of a model. A path p := (s_0, … s_n) of Kripke structure K is a finite or infinite sequence such that $\forall\, 0 \leq i < n-1 : (s_i, s_{i+1}) \in T$.

A model-checker like SPIN takes a Kripke structure as model of a system and a temporal logic property as inputs. Then it explores the model's state space to determine whether the property is satisfied or violated, i.e., whether all the paths derived from the model satisfy the property or whether there exist a path derived from the model for which the property is violated. If a property violation is detected, then an executions sequence (counter example) is returned, showing how the violation was reached, or how a certain sequence violates the property.

LTL is a subset of CTL*, and a CTL* formula is a collection of atomic propositions, connected with logical operators (e.g., the usual and, ∧, not, ~, and implication, →), temporal operators (e.g. `F, G, U, X`) and path quantifiers (`A, E`). Operator '`X`' refers to the next state, e.g. `X a` means that proposition `a` has to hold in the next state. '`U`' is the "un-

til" operator, where `a U b` means that proposition `a` has to hold from the current state up to a state where proposition `b` is true (and possibly after), and a state where proposition `b` is true has to exist. '`G`' is the "always" operator, and is used to specify that a proposition has to hold in all states of a path; '`F`' is the "eventually" operator that requires a certain proposition to eventually hold at some point in time in the future. The path quantifiers '`A`' ('all') and '`E`' ('some') require propositions to hold on all or some paths, respectively. In LTL, operator '`F`' is denoted by symbol ◊ and operator "`G`" is denoted by symbol □..

LTL is the subset of CTL* obtained by the following grammar: $\varphi ::= \text{true} \mid \text{false} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \sim\varphi \mid \varphi_1 \rightarrow \varphi_2 \mid X\,\varphi \mid \varphi_1\, U\, \varphi_2 \mid \Diamond\varphi \mid \Box\varphi$, where $a \epsilon AP$.

If a proposition φ is satisfied by a model `K` or path σ, then this is denotated as $K \models \varphi$ or `K`, $\sigma \models \varphi$, respectively. In `K`, $\sigma \models \varphi$, K may be omitted if there is no ambiguity, and we write $\sigma \models \varphi$. If a model violates property φ, then this is denoted as $K \not\models \varphi$.

The semantics of LTL is expressed for infinite traces of a Kripke structure. Assuming σ_i denotes the suffix of the path σ starting from the i^{th} state, and σ^i denotes the i^{th} state of σ, we have the following:

1.	$\sigma \models true$	for all σ, i.e. the true proposition is always satisfied by a path.
2.	$\sigma \not\models false$	for all σ, i.e., the false proposition is never satisfied by a path.
3.	$\sigma \models a$	iff $a \in L(\sigma)$, *i.e., if a path σ satisfies a proposition a, this is equivalent to stating that there exists a labeling function L for path σ that holds a.*
4.	$\sigma \models \sim\varphi$	iff $\sigma \not\models \varphi$, i.e., if a path satisfies the negation of a proposition, this is equivalent to stating that the path does not satisfy that proposition.
5.	$\sigma \models \varphi_1 \wedge \varphi_2$	iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$, a path satisfies the conjunction of two propositions if and only if both propositions are satisfied by the path.

6.	$\sigma \vDash \varphi_1 \rightarrow \varphi_2$	iff $\sigma \nvDash \varphi_1$ or $\sigma \vDash \varphi_2$, since A implies B is logically equivalent to not(A) or B.
7.	$\sigma \vDash \varphi_1 \, U \, \varphi_2$	iff $\exists i \in N_0 : \sigma_i \vDash \varphi_2 \; and \; \forall 0 \leq j \leq i, \sigma_j \vDash \varphi_1$, i.e., there exist a natural number i such that proposition φ_1 holds until (at least) σ^i (i.e., for any path starting before σ^i) and proposition φ_2 holds after that.
8.	$\sigma \vDash \mathrm{X}\varphi$	iff $\sigma_1 \vDash \varphi$
9.	$\sigma \vDash \Box\varphi$	iff $\forall j \, N_0 : \sigma_j \vDash \varphi$
10.	$\sigma \vDash \Diamond \varphi$	iff $\exists j \in N_0 : \sigma_j \vDash \varphi$

For example, let us consider a hypothetical model of a traffic light for which we specify properties by expressing them using LTL:

(a) Once the traffic light is green, the traffic light cannot become red immediately: $\Box \, (green \rightarrow \sim X \, red)$

(b) Once the traffic light is red, the traffic light becomes green eventually: □ (red →◊ green)

(c) Once the traffic light is green, the traffic light becomes red eventually, after being yellow for some time in between: □ (green → (green U yellow) U red)

LTL formula are typically used to express liveness and safety properties. A liveness property illustrates that 'something good eventually happens', For example, assuming φ_2 represents a 'good' property that has to happen eventually after some other property φ_1 has happened, the liveness property can be formed in LTL as $\Box \, (\varphi_1 \rightarrow \Diamond \, \varphi_2)$. A safety property requires that a certain behaviour always or never occurs ('something bad does not happen'). For example, assuming φ_2 represents 'bad' state that should not happen once property φ_1 has been reached, the safety property can be expressed in LTL as $\Box \, (\varphi_1 \rightarrow \sim X\varphi_2)$.

Once an LTL is constructed and executed under a model checker, the LTL property can be violated or satisfied. A model checker such as SPIN returns an example of how this LTL is violated (counterexample) or satisfied (witness).

We end this section with a short discussion on how model-checking can be used for test case construction, assuming we want to use SPIN. When a test model, for instance a state machine, can be transformed into a Kripke structure that can be consumed by SPIN, test case construction can proceed as follows. For each test purpose that one wants to achieve, e.g., covering a transition, one defines an LTL property that states that it is never possible to reach this test purpose. In this context, such a property is often called a trap property [13]. Assuming the test model is correct, i.e., it is indeed possible to reach this test purpose, and SPIN can handle the complexity of this test model, then SPIN will be able to find a counterexample showing how that test purpose can be fulfilled: this counterexample is then a test case achieving that test purpose.

More specifically, assuming (without loss of generality) a transition t is defined by its from-state fs and its to-state ts, to satisfy the transition coverage criterion, a test sequence $\pi \vDash s_0 \ldots s_i, s_{i+1} \ldots s_n$ such that $s_i = \text{fs}$, $s_{i+1} = \text{ts}$ and $s_n = \text{exit}$ must exist. The last (exit) part ensures the path continues completely to the exit state. The witness formula $\{\text{EF}\,(\text{fs} \wedge \text{EX}(\,\text{ts} \wedge \text{EF exit}))\}$ satisfies sequence π; this witness formula means that eventually there exists a path where the from-state fs is reached first, and the to-state ts is reached next. After that eventually the model runs completely to the end; the witness formula illustrates the correct behaviour of the model, so that to create a trap property that illustrates the undesired behviour of the model we need to negate the witness formula to force the model checker to generate the counterexample that shows a trap property is

violated and the trap property express incorrect behaviour of the model. To build a trap property from the witness property, we need basically to negate the witness property; this can be done, by first negating the eventually operator into an always operator (AG) in order to ensure that this path has to always exist. The implication $p \rightarrow q$ means that if the premise p is true then the consequence q must be true; the implication is evaluated to false, if the second premise q is false. We use this fact to create LTL trap property. The fact that fs is eventually followed by ts is transformed into an implication, meaning that if the first proposition (reaching fs) is satisfied, then the second proposition (reaching ts) is not satisfied next; this will create the following LTL property $\{\mathrm{AG}\,(\mathrm{fs} \rightarrow \mathrm{EX} \sim \mathrm{ts})\}$. Since we also want to make sure the model runs to the end after the ts state is reached, we use the exit state afterward; consequently, we have added a third proposition exit to our LTL property, so that if the second proposition ts is satisfied, then the third proposition exit is not always reached; this will violate the model behaviour since the model has to run to the end state eventually. The following LTL trap property is therefore created, $\{\mathrm{AG}\,(\mathrm{fs} \rightarrow \mathrm{EX} \sim (\mathrm{ts} \rightarrow \mathrm{AG} \sim \mathrm{exit}))\}$. This also suggests that such trap properties can be automatically derived from a test model.

2.4 RTEdge Overview

In this section we will use a realistic automobile engine management system, or Engine Control Unit (ECU), to illustrate the capabilities of the RTEdge tool. An ECU is an electronic control that determines the quantity of fuel, ignition timing and other parameters by monitoring the engine through sensors. These sensors exist in different locations throughout the engine and automobile, however they all transmit data back to the central ECU module. At a high level of abstraction, the model consists of several RTEdge cap-

sules (see below for a definition of this notion) that describe real-world automobile components and functions such as fuel and air sensors, throttle position, and acceleration.

The RTEdge toolset is a Model Driven Development (MDD) platform, based on Eclipse, for designing critical real-time embedded applications. RTEdge is built around a modeling subset of AADL [14] and UML 2.0 [15].

The RTEdge toolset, thanks to its architecture supports a number of MDD activities, as discussed in the following subsections.

2.4.1 The specification of a software system as an RTEdge model.

The model is typically a set of state machines communicating through signals and ports (a.k.a., capsules), annotated with assertions (e.g., state invariants), constraints (e.g., guards), activity behaviour specifications under the form of C/C++ code, and expected temporal properties. Fig. 1 shows an Atomic Capsule (AC), called `Fuel_Air_Mix_DB_AC`. The fuel-air mixture database capsule controls how rich or lean the engine runs, by controlling the mixture of fuel and air in the combustion chamber. The `Fuel_Air_Mix_DB_AC` uses a state machine to model this real-time process. The `Fuel_Air_Mix_DB_AC` state machine diagram shows the flow of control in the fuel-air mixture module. When the fuel-air mixture module receives input from the update cycle controller module (UCC) through the signal `FAM_Data_Required` on port `FAM_UCC`, transition `FAM_DB_GotNewUCCInput` is triggered. The activity attached to Transient state (identified with a dashed border) `FAM_DB_ProcessInput` looks up fuel-air mixture data values from the fuel-air mixture data lookup tables. The lookup table is used to compare the current throttle position to the current engine Revolutions Per Minute (RPM) values and to determine exactly how much fuel to inject into the combustion chamber.

When the FAM_DB_ProcessInput state completes, i.e., its activity completes, the state returns to FAM_DB_WaitForInput either with a data update or without. In the former case, transition FAM_DB_SendResultToUCC fires and a signal is sent to the update cycle controller module informing it of the change. In the latter case, transition FAM_DB_NoActionRequire fires and no signal is sent. A transition's head is bold to indicate that an out statement is attached to this transition where the out statement is used to send signals from one Atomic or Proxy Capsule to another, i.e., the transition FAM_DB_SendResultToUCC sends a signal to the update cycle controller module capsule; otherwise, the transition's head is not bold and there is no out statement attached to the transition, i.e. transition FAM_DB_NoActionRequire has no out statement.

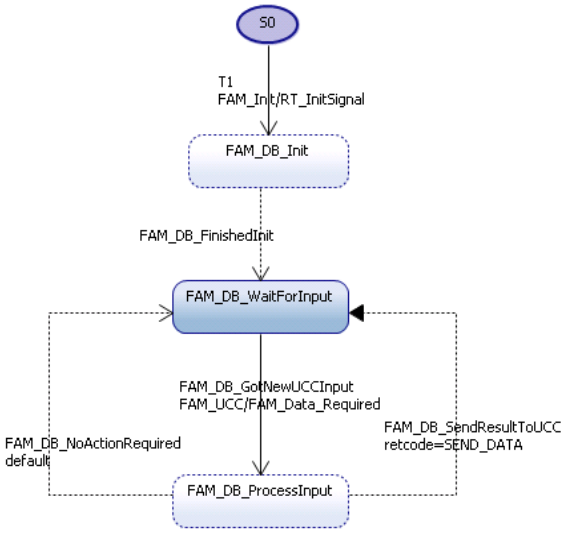


Fig. 1. – the Fuel_Air_Mix_DB_AC state machine diagram

The model is specified to respond to so-called Independent System Inputs as well as Dependent System Inputs. An Independent System Input is generated by the environment of execution of the RT application being designed, independently from any behaviour specified in the RTEdge model. A Dependent System Input is also sent by the environment,

but at the request of the RT application, as specified in the RTEdge model, i.e., the RTEdge model sends a signal to its environment and the environment is expected to respond with a Dependent System Input.

Regardless of its kind, a System Input results in automatic generation of API calls so that external software can send Signals into RTEdge. An Independent System Input is defined by time arrival constraints (typically a period) and may carry data specified in a way similar to C/C++ structures. A Dependent system Input is also specified by a time arrival constraints (the model/application expects it to arrive within a specific amount of time after it sends the request to the execution environment) and data.

For example, Fig. 2 shows an Independent System Input (ISI) definition in RTEdge. ISI `Send_RPM_To_ECU` defines an `EXT_Sensor_Data` signal which is sent from the external environment to the ECU to update its RPM data (using RPM sensor data that the signal carries). `Send_RPM_To_ECU` is periodic with a worst-case arrival rate of 60 millsecond. RTEdge allows only one instance of a given System Input to be processed at any time and if an instance arrives before the previous one has completed, this is considered an overrun. In order to manage overruns the user can specify a queue size for each System Input that will buffer instances up to the specified size: in our example the queue size is set to one. If the queue fills up and another System Input instance arrives, the user has a choice of actions to take: in our case, the overrun situation is ignored, i.e., the information that there has been an overrun is not kept; alternatively, overruns situations can be logged (LOG can be selected instead of IGNORE), or throw an exception (EXCEPTION instead of IGNORE). Regardless, the last overrunning input is dropped.

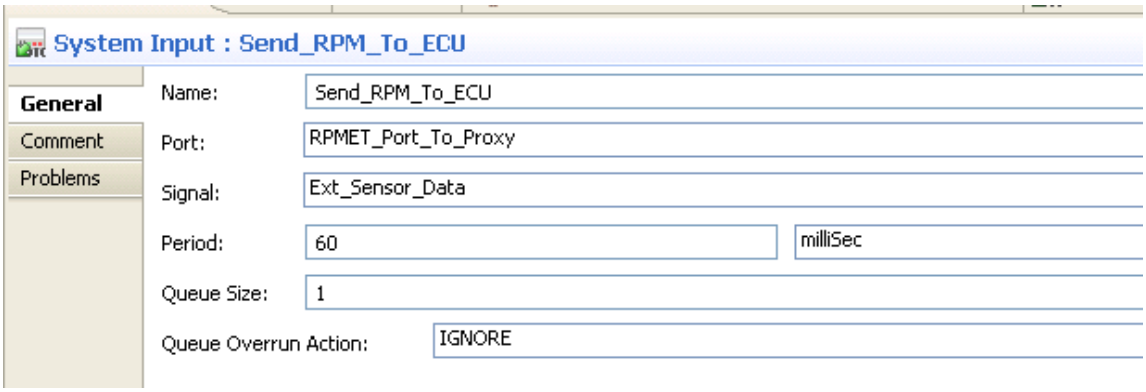
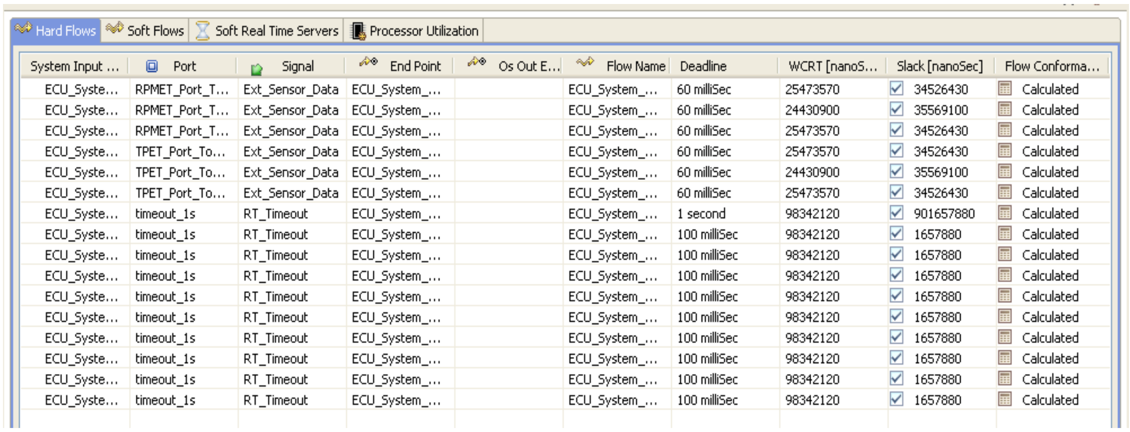


Fig. 2. – Independent system input defined in an RPM_ET External Task Capsule.

2.4.2 Schedulability analysis

Schedulability analysis is the process of analyzing an RTEdge model for temporal correctness and resource utilization. During this process, the Worse Computed Response Time (WCRT) is determined for each transaction, i.e., an execution triggered by an Independent System Input that ends when no more internal transitions are to be triggered, when no more Dependent System Input is expected. For schedulability analysis purposes, RTEdge relies on annotations added to the model by the designer and that specify estimated execution times (in particular for activity behaviour specified in C/C++).

For example, executing the schedulability analysis on our automobile engine management system provides the results of Fig. 3. The first column shows the ISI name, the second column shows the ISI's port, the third column is the ISI's signal, and the end point is shown in the next column. End point is either a Signal leaving the system or a stable state the state machine will have reached after having processed the ISI. The deadline is by default determined by the ISI's period, and the WCRT is shown in the next column. Each row shows the details for each corresponding ISI.



Hard Flows | Soft Flows | Soft Real Time Servers | Processor Utilization

System Input ...	Port	Signal	End Point	Os Out E...	Flow Name	Deadline	WCRT [nanoS...	Slack [nanoSec]	Flow Conforma...
ECU_Syste...	RPMET_Port_T...	Ext_Sensor_Data	ECU_System_...		ECU_System_...	60 milliSec	25473570	34526430	Calculated
ECU_Syste...	RPMET_Port_T...	Ext_Sensor_Data	ECU_System_...		ECU_System_...	60 milliSec	24430900	35569100	Calculated
ECU_Syste...	RPMET_Port_T...	Ext_Sensor_Data	ECU_System_...		ECU_System_...	60 milliSec	25473570	34526430	Calculated
ECU_Syste...	TPET_Port_To...	Ext_Sensor_Data	ECU_System_...		ECU_System_...	60 milliSec	25473570	34526430	Calculated
ECU_Syste...	TPET_Port_To...	Ext_Sensor_Data	ECU_System_...		ECU_System_...	60 milliSec	24430900	35569100	Calculated
ECU_Syste...	TPET_Port_To...	Ext_Sensor_Data	ECU_System_...		ECU_System_...	60 milliSec	25473570	34526430	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	1 second	98342120	901657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated
ECU_Syste...	timeout_1s	RT_Timeout	ECU_System_...		ECU_System_...	100 milliSec	98342120	1657880	Calculated

Fig. 3. –Schedulability analysis result.

2.4.3 Model checking

Various safety and liveness temporal properties can be specified and checked on an RTEdge model. To do that, the RTEdge Formal Link feature can automatically transform an RTEdge model into an equivalent Promela model [10]. The Promela model can then be automatically run under the SPIN [16] model checker [10] to verify safety and liveness properties. RTEdge can then interpret SPIN counter-examples as RTEdge model execution traces.

Specifically, RTEdge Formal Link defines AC (Atomic Capsule) Instance Behavior Annotations. These are conditional assignments to Global Checking Variables only. A global checking variable (GCV) is a variable declared to the RTEdge Application state space, but it is translated into global state variables in the Promela model.

Assertions to be checked are added to prove certain relevant RTEdge model safety properties. Depending on the insertion point in the model, RTEdge distinguishes between the following types of assertions:

1. An Inline Checking Assertion is inserted in a state machine transition of an AC, referencing GCVs and state variables that are specific to this AC (in summary, the

assertion is a condition on variables that are in the scope of the transition). They are evaluated inline, i.e. when the execution reaches their insertion point.

2. An AC Instance Checking Assertion is inserted in an AC instance of an RTEdge Application, referencing GCVs and state variables of this AC. Such an assertion is evaluated on every transition of the AC state machine, after all the behavior annotations and Inline Checking Assertions attached to each transition have been evaluated.

3. An Application Scope Checking Assertion is inserted at the RTEdge Application level, referencing GCVs and is evaluated on every change of the global state, meaning that any change to the model state will trigger the evaluation of Application Scope Checking Assertions.

2.4.4 Other software development related activities

Other activities are supported by RTEdge, such as:

4. RTEdge provides a Periodic Timer through which state machine execution can be triggered. The timer uses a Service to broadcast a timeout Signal to any capsules using the service;
5. The linking of model elements to requirements, for instance collected in IBM DOORS;
6. The verification of the structural correctness of the RTEdge model: e.g., communicating state machines exchange the same set of signals, but in opposite directions.
7. RTEdge defines deployment artifacts that one can use to describe the physical implementation of the target, including the operating system (e.g., solaris) and

compiler options for the target, the deployment strategy, the hardware to deploy to.

8. With a software/hardware mapping, RTEdge's Code Generator can automatically create the application C++ source code. RTEdge can then compile and build (using gcc) for deployment on the target platform.
9. RTEdge provides a debugger to for instance set up breakpoints to observe and modify variables, but also to monitor ports and capsules as well as inject signals.

2.4.5 RTEdge runtime environment: Virtual Time

One specific runtime environment supported by RTEdge is the RTEdge Virtual Time Environment, or simply Virtual Time. Virtual Time simulates the execution of an application automatically generated by RTEdge on a host/development machine rather than on the target platform. Virtual Time is especially helpful during verification since the user has precise control over the arrival of System Inputs (e.g., injection), has precise control over how time passes and Virtual Time offers the same capabilities as the debugger mentioned earlier. In addition, Virtual Time allows the definition of call-back functions whereby one can register a function that will be called under a particular circumstance during the simulation of the model. Particularly interesting to us, call-back functions can be called when a transaction ends, a transaction starts, and an RTEdge periodic timer expires. A transaction is the set of all Flows that result from a single System Input. They will have a common start point (the System Input) but may have multiple endpoints, depending on the paths through the system.

Chapter 3 - **Related Work**

We discuss three fields of research that pertain to our thesis, specifically: state-based testing (section 3.1), random testing (section 3.2), and the combination of software testing and model checking (section 3.3).

As it will become clearer by reading those related work sections and the following chapters, our approach differs from related works on state-based testing and random testing in one or more of the following ways. (1) We do not perform any analysis of the state model to identify new inputs, and we do not perform transformation of this model. Instead, as discussed later in the thesis, we only use information about the signals that can be sent to the system, i.e., the signals the state-based behaviour can respond to; in other words, the state model (and not only the code) is a black-box. (2) Our test model is a set of communicating state machines, which also include pieces of C/C++ code for the specification of actions, activities and guard conditions: guards and actions are not assumed to be linear. (3) We focus on achieving a complete set of coverage objectives instead of one objective at a time, similarly to some recent white-box testing approach [17]. (4) As discussed later in the thesis, when random selection is performed, we sample from a domain using a uniform distribution, without any attempt to improve over this simple random selection.

3.1 State-based testing - related work

State-based testing is a vibrant field of research in software engineering and computer science. The testing strategies put into place when testing from a state machine, or a set of communicating state machines, heavily depends on the kind of behavior specification the state model contains. When the state machine does not have actions (or activities) on transitions or states, or guard conditions, then many structural approaches exist to auto-

matically generate test cases, such as the W-method, the Wp-method or the UIO-method [18]. A set of communicating state machines can under some conditions (so as to avoid a state space explosion) be transformed into a larger extended finite state machine from which the abovementioned techniques can be used [19]. Alternatively, techniques specific to communicating state machines exist (e.g., [20]). Other techniques involve symbolic execution (e.g., [21, 22]) or a meta-heuristic search (e.g., [23]). Alternatively, one can consider testing techniques for labeled transition systems (e.g., [24]). When the state machine (or a set of communicating state machines) has actions (or activities) and/or guards that are all linear, automated test case construction is also feasible: e.g., [5, 25-29]. In all those cases, the testing technique is typically offline since it is possible to statically analyze the model and create feasible test cases prior to executing them. There are exceptions, such as UPPAAL-TRON [29], which is a technique (and a tool) to online test a real-time embedded software from a timed automaton (or network of timed automata) specification. When actions (or activities) and guards are specified with a more complex language, offline testing is typically not possible since it is not possible to statically analyze both the state based behavior and the pieces of code, which can be as complex as any piece of Java/C/C++ code, to create feasible test cases: e.g., RTEdge uses C/C++ code for specifying actions, Conformiq [22] uses a Java-like language for the same purpose; Instead, online testing is necessary to simulate state changes, and more so those pieces of code, to identify the resulting state and therefore identify what can be the next event to send to the implementation/simulation of the state–based behavior.

3.2 Random testing - related work

Another domain of related work pertains to random testing [30], that is the automated generation of test inputs from an input domain or an operational profile, often by using a uniform distribution. Random testing has been shown to be effective, sometimes surprisingly more effective than other (structural) criteria [30]. Such results have recently been confirmed: random testing appears to be more effective and predictable than previously thought [31]. In an attempt to address some of the issues of random testing, specifically the fact that some inputs may be better than others at detecting faults, even when using partitions, some have proposed to maximize test case diversity with adaptive random testing [32]. Although many successes have been reported, its real effectiveness has been put to question recently [33].

Other attempts to enhance random (white-box) testing include directed random testing and feedback random testing, or coverage rewarded random testing. In directed random testing [34], monitoring program execution during random testing and collecting information on executed paths is used to systematically direct the selection of new random inputs to lead execution to trigger new program paths. In feedback random testing [35], unit test for object-oriented classes are created as legal (according to contracts) sequences of method calls and execution results (feedback) in terms of violated contracts and execution outputs (e.g., returned values) is used to drive the extension (with new method calls and new input data) of test cases being constructed. In coverage rewarded random testing [36], reinforcement learning is used to obtain interesting new test cases. The first two techniques have been used together and extended to automatically generate black-box test cases from simulink/stateflow models [37], in an attempts to improve upon existing,

purely random input selection techniques (e.g., Reactis [38]). This required that the state model be flattened and unfolded up to a pre-defined certain depth.

3.3 Testing with Model checker - related work

Model checkers were successfully used during software testing, especially when testing reactive systems. The testing literature has recently been discussing many connections between test generation and model checking. A number of methods have been proposed for using model checkers in test generation: the reader interested in a survey is referred to [12]. The survey classifies the test cases generation approaches into two main categories: In the first category, an approach is based on test coverage where the temporal logic formula represents a particular element of the model to be covered according to a coverage criterion; In the second category, an approach is based on mutation, where a single fault is introduced into a program or specification, and a model checker generates counterexamples which distinguish these mutants from the original model.

Ammann and Black [39, 40] use mutation analysis to generate test cases using a model checker for a Symbolic Model Verifier (SMV) [41] logic model. They define syntactic operators which produce mutants on a given model at the level of the model checker specification. A model checker generates counterexamples which distinguish these mutants from the original model.

In [42], this approach uses Extended Finite State Machines (EFSMs) to be represented by binary decision diagrams (BDDs) and it describes symbolic approaches to generate test suites that satisfy state and transition coverage criteria using the SMV model checker. Another approach is to automatically generate test cases from a state model to satisfy structural coverage criteria [43]. The authors show how to formalize a variety of structur-

al coverage criteria, such as state, transition or MC/DC, and then used a model checker to provide test sequences to achieve a particular coverage.

An alternative approach uses an SCR (Software Cost Reduction) requirements specification model for constructing a test suite sequence [13]. The SCR specifications consist of different types of tables. This approach automatically translates the SCR specification to an SMV or SPIN model, after that it automatically constructs a set of trap properties from the SCR tables.

Alternatively, many approaches use a Statechart test model to generate test cases using a model checker: e.g., satisfying the all states or the all transition criterion by using the NuSMV model checker [43] or SMV [44]. A similar approach has been devised for state-flow models by using the SAL model checker [45]. One important difference with our approach is that activities on states are not supported whereas our test model typically contains such activities. Another difference is that the translation into SAL is manual.

Chapter 4 - **Design and Implementation of an RTEdge Test Framework**

In this chapter, we will describe the principles of the RTEdge Test Framework created. When creating this framework, we wanted to derive test cases from an RTEdge model with an (model-driven testing) approach/algorithm as simple as possible, and we wanted to rely on the existing features of RTEdge to the maximum extent possible for testing purposes.

In this chapter, we first discuss the rationale for the main decisions that drove the design of our test framework. Specifically, we argue in favour of an online test case generation (section 4.1). We then discuss the principles of our test framework in details (section 4.2) and show how it can be implemented with RTEdge, specifically with Virtual Time (section 4.3).

4.1 Selecting an Online Feedback Random Test Case Generation

Since an RTEdge model can become quite complex it appeared very quickly that any static analysis of the model (including static analysis of the C/C++ code in contains) would be too expensive to allow any offline test case generation. In other word, it appeared too expensive to identify a strategy that would analyze the model and allow us to generate adequate test suites made of feasible, executable test cases, according to standard selection criteria [4, 46], similarly to what is done in many other pieces of work (recall the discussion of section 3.1). Specifically we felt it would be too complex or even impossible to devise a test case generation strategy that would typically identify test inputs (i.e., signals to be sent and the data they carry, the times for sending them) in order to ensure that some transitions are triggered, that some states are reached. Instead, we de-

cided to use only information about the Independent and Dependent System Inputs to create test cases: the details of the model, i.e., its communicating state machines, are not used as (primary) test objectives to drive the test case generation. In other words, we consider the tested software as a black-box, not only because we do not look at its implementation (the source code) to create test objectives and therefore test cases, but also because we do not look at the state machines.

Test objectives should then be created from the specification of (In)Dependent System Inputs, that is their timing information (e.g., period) and the data they carry. At this level of abstraction, there is however no clear relation between such specification, specifically timing values and data values carried by signals, and the behaviour actually triggered in communicating state machines: this triggered behaviour is in the state machines, which we do not use to derive test cases. Therefore, instead of using criteria (e.g., based on equivalence classes for the data carried by input signals) to derive test cases, we decided to rely on a random generation (see section 4.2 for details).

Since any random test case generation can run forever unless we identify a stopping criterion, we nevertheless rely on coverage of the model (specifically states and transitions) to decide whether the random generation needs to continue or to stop. Specifically, we monitor model coverage as test cases are generated and executed and we stop random generation when we have reach adequacy or when we do not observe significant coverage improvement: section 4.2 provides more details.

In summary, we developed an online feedback-driven random test case generation.

After the design of the model, a simulation model is built where the simulation is used to verify the model. Our test framework stimulates the RTEdge model (simulation) to verify

the model and to ensure it is fault free. We discuss next the fault model we are targeting. Our test framework can verify if there is an unspecified signal through the simulation, which happens during the simulation when a signal is sent to a particular capsule but the capsule's current state does not accept this specific signal; thus, our test framework reveals if an unspecified signal exists in the RTEdge model. Another fault our test framework can reveal is when our test framework is terminated by one of the stopping criteria. Specifically, a set of covered states and transitions are given at the end of a run of our framework, showing which states and transitions have been visited through the simulation. So, if a set of states or transitions are not visited through the simulation, then a specific path was not taken through the simulation, and this may show there might be a fault in the model that caused the simulated model not talking this particular path. Furthermore, the user-defined assertions are evaluated during the simulation to assert if a particular requirement in the model is satisfied; therefore, if an assertion is evaluated to false during the simulation, then either a particular requirement is not satisfied in the model or the assertion is not correctly defined. Our test framework can reveal a fault in the model when we rely on the SPIN model checker to produce counterexamples that should lead to feasible test cases that increase structural coverage. If SPIN fails to produce a counterexample, this means it is not possible to trigger a specific transition in the model (as specified in the LTL property), which is a fault in the model. The SPIN model checker may also fail if an assertion is violated through the verification process. In addition, it may fail because of an out of memory problem; this means that the model is too large; therefore, SPIN is not able to complete the verification process of the model.

4.2 The Test Framework

As discussed previously, our test framework uses an RTEdge model is an input and, more specifically, it uses the specification of (In)Dependent System Input signals (including their periods, the data they carry) that can be sent to the test model. Because of the time-related aspect of an RTEdge model, our framework is to automatically generate a test script that periodically and randomly sends the input signals to an executable version of the model, and observes the progress in terms of coverage of the model to decide when to stop the creation/execution of test cases.

Fig.4 shows the infrastructure of our framework. It abstracts away from technical details on how this is actually realized with RTEdge's Virtual Time (this is discussed in section 4.3), while allowing us to precisely discuss the main aspects of the framework, specifically the sending of Independent System Inputs (section 4.2.1) and Dependent System Inputs (section 4.2.2), including the random generation of the signal data (section 4.2.3), evaluating assertions to prove certain user-defined properties (section 4.2.4) and the criterion we use to stop the random generation of test cases (section 4.2.5). In doing so, we attempt to show that our test case generation procedure is, in principle, not tied to RTEdge's Virtual Time but could in theory be applied to other environments. Future work will look into how this procedure can be adapted to other modeling environments.

We also discuss the annotations we create in the model to facilitate our online testing strategy (section 4.2.5), as well as how our testing strategy is complemented with model checking in the context of RTEdge (section 4.2.6). Section 4.3 discusses how all these steps are actually realized with RTEdge's Virtual Time.

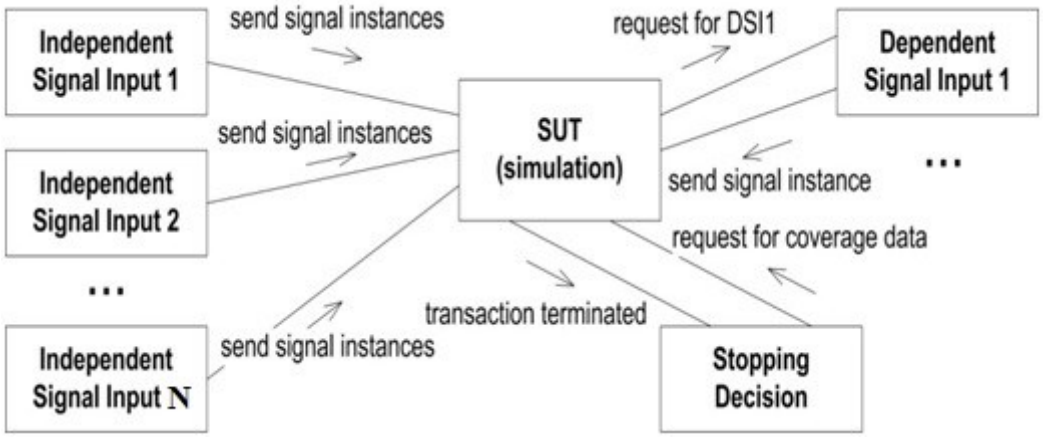


Fig. 4. –Overview of the Test Framework

4.2.1 Sending Independent System Inputs.

The sending of an Independent System Input (ISI) to the SUT is done independently for each ISI, and proceeds as follows. Since an ISI is periodic, we start by sending an instance of the ISI, randomly generating the data it carries (section 4.2.3) at time 0 (zero). Virtual Time manages time and time 0 corresponds to the SUT being created and ready to respond (section 4.3). Then, at each time value equal to a multiple of the ISI's period, we send another ISI instance, again randomly generating its data.

4.2.2 Sending Dependent System Inputs.

A Dependent System Input (DSI) is supposed to be sent by the environment upon request by the SUT. The DSI is supposed to be sent, only once, within a specific amount of time after the environment receives the request. When a request for a DSI is received, we send a DSI instance by randomly selecting a delay between the request arrival for this DSI and the sending of the DSI instance (randomly selected according to the delay specification of the DSI), and randomly selecting data for the DSI (section 4.2.3).

4.2.3 Randomly Generating Signal Data.

We created a signal data random generator for signals that do carry data. This generator supports a subset of the types that RTEdge supports, specifically: bit (1 bit, unsigned), bool (1 bit, unsigned), byte (8 bits, unsigned), short (16 bits, signed), int (32 bits, signed), and char ([a..z][A..Z]). This subset is in fact a set of types supported by SPIN plus character: recall that RTEdge interfaces with SPIN. For these primitive types, the generator randomly (uniform distribution) selects a value in the allowed range.

RTEdge also supports more complex data types which are arrays, enumerations, and structures (similar to C struct). In order to generate an array, the generator determines the data type of the elements in the array, and then generates the required data depending on the user-defined array size. For each structure, the generator determines the data type of each element it contains and generates the required data depending on the element type. In the case of an array or a structure, the generation is recursive. For an enumeration, the generator produces at random (uniform distribution) one of the possible values.

RTEdge allows the refinement of existing types by specifying a reduced allowed range of values, thanks to Data Range Constraints. Our generator uses those constraints to randomly (uniformly from the constrained range) generate values.

4.2.4 Attaching assertions.

As we have discussed in section 2.4.3, assertions to be checked at runtime on the model can be added. The assertions are defined in RTEdge Formal Link to be used by SPIN: they are specified in the RTEdge model in the Promela language. When generating code for the RTEdge model under the VT environment, an assertion's hook is generated for each RTEdge element so that any assertion to be checked on that element can be checked.

The hook is composed of a pointer to a function which is supposed to perform this check. However during code generation, RTEdge does not automatically create (the body of) this function: the assertion's pointer is initialized to null; and as a consequence the assertions are not automatically attached to their insertions points. In our verification framework, we wanted to check those assertions during a simulation. We therefore created code to automatically translate the assertion specification in Promela into C code, specify assertion checking functions with this code and create the hooks. Fortunately, the Promela syntax is close to C and the translation was not too difficult: the most difficult aspects had to do with RTEdge Data Access Point (DAP), the use of variable types. The following steps demonstrate the translation procedure from Promela to C code.

A local variable in an Atomic Capsule (AC) can be accessed using the `locals_ptr ->variable_name` format where `locals_ptr` is a pointer to a structure containing all the local attributes of the capsule, as declared in the model. However, when the RTEdge model is translated into Promela, the local variables are accessed using the `attr.variable_name` format where `attr` is a structure containing all the local variables of the capsule, as declared in the model. (The two structures, `locals_ptr` and `attr`, have the same contents, i.e., the same number of elements with the same names.) Therefore, in order to translate the annotations to C/C++ language, we need first to replace each `attr.` with `local_pointer->`. Secondly, the signal data is accessed in RTEdge using a `event_data` pointer, but in order to access a signal data in the RTEdge's Promela model, we use `sd_`. Thus, our translation process works by replacing the access pointer for the signal data `sd_` with `event_data`. Thirdly, an RTEdge model has two types of Data Access Point (DAP) which are Provider and Required DAP: The

Provider type indicates that DAP's capsule is the provider of the shared data and contains the data associated with it; The Required type indicates that DAP's capsule needs access to the shared data. The access to the shared data is different for the Provider and Required DAP, therefore, we have to check the DAP type before translating an assertion. If the DAP type is Provider, then we replace its access with "`.`"; however, if the DAP type is Required, then we replace its access with "`->`".

By following the above steps to translate the assertions to C code, the assertions could be run under VT environment.

The following example shows an annotation that access a DAP data where its type is Required: `SensorData.S9_Angle == 0`. The corresponding C/C++ translation is: `SensorData->S9_Angle == 0`. The second example shows an annotation that accesses a local variable in Promela where the `Current_Position` variable is an integer: `attr.Current_Position == 0`. The corresponding C/C++ translations is `locals_ptr->Current_Position == 0`. The third example shows an annotation that accesses the signal data: `sd_Counter = 0`. The corresponding C/C++ translation is `event_data.Counter == 0`.

4.2.5 Stopping criterion.

As we have discussed in sections 2.4.2 and 2.4.5, a transaction includes everything the SUT has to do in response to an Independent System Input (ISI), i.e., an ISI has arrived and is consumed. Therefore, we decided that the end of a transaction would be a good time to decide whether to stop or continue generating a test case. Therefore, each time the SUT finishes a transaction, it informs the framework, providing details about the transaction: start and end times of the transaction, states and transitions covered by the transac-

tion. (Note that Virtual Time allows us to stop the simulation of the model to collect that information. We therefore avoid any impact of the collection process on execution times, and therefore deadlines. The simulation thus remains representative of what would actually execute on the deployment platform.)

We decided to define a stopping criterion based on the coverage achieved by test cases, i.e., the transactions they trigger, and we opted for two standard criteria [4]: the state criterion requires that all the states in the model be exercised; the transition criterion requires that all the transitions in the model be exercised.

Ideally, the online testing procedure would stop when an adequate test suite has been generated, i.e., when the predefined level of 100% coverage of the model (i.e., states and transitions) has been achieved. This is however not a guarantee since our test case generation is black-box and random. We therefore need a stopping criterion to be used in case the random generation fails to reach adequacy. We want to avoid an endless test case construction. One possibility could have been to ask the test engineer to decide of a coverage level to reach. But even then, except if one selects a trivial coverage level to reach, in general there would be no guarantee to actually reach it. One difficulty is that due to the random nature of the test case generation, coverage can increase for some transactions, it may appear to have come to a standstill for another transaction, and may increase again later on. Similarly to what is done in some genetic algorithms [47], we decided to observe coverage over a number of random selections, specifically over a user-specified number of transactions, what we refer to as the observation window (i.e., `obsWindow`). If no additional coverage is observed during this window, then we stop. More formally we defined three flags:

- `noNewCoverage` is true if the increase of coverage in terms of states and transitions (i.e., new states and transitions) that is observed during the observation window is less than a used-defined minimum, `minIncr`. This minimum is the increase in coverage a user would expect to observe in the observation window;
- `stateTarget` is true if we have reached adequacy with respect to states;
- `transTarget` is true if we have reached adequacy with respect to transitions.

Our random generation then stops if the following condition is true: `(transTarget or noNewCoverage`; we do not check `stateTarget` flag since transition coverage subsumes state coverage, but we keep it to facilitate the use of the less demanding state-coverage criterion. In other words, if no new coverage is observed during the user specified number of transactions (what we refer to as the observation window), regardless of the level of coverage for state and transitions, we stop. Alternatively, as soon as we have reached adequacy with respect to transition criteria, we also stop. In any other situation, we continue generating test cases.

Our approach also requires some parameters from the test designer:

In the traditional definition of adequacy, covering a model element (e.g., transition) once for each element is sufficient to consider one has reached adequacy. In our case, we allow the user to define how many times each model element needs to be covered in order to consider one has reached adequacy, which we refer to as `minCover`. In other words, an adequate test suite for the transition coverage criterion is achieved when each transition in the model is covered/executed at least `minCover` times. The traditional definition of adequacy is obtained with `minCover=1`.

In summary, our stopping criterion requires the following inputs from the user:

- `obsWindow`, the size of the observation window;
- `minCover`, the minimum number of times each model element needs to be covered in order to reach adequacy;
- `minIncr`, the minimum increase coverage that one expects to observe during the observation window.

4.2.6 Additional test case generation (if necessary) with the SPIN model checker.

Once the stopping criterion has been satisfied, test case generation stops with information on structural coverage of the test model: the number of times each state and transition has been executed is collected. At this point some states or transitions may have been missed by our random, feedback driven online testing procedure. Our framework provides an alternative solution to increase coverage by using the SPIN model checker. As we have showed in section 2.3, we design a trap property for each uncovered state and transition, run the LTL trap property under SPIN. When SPIN is able to provide a counterexample, this becomes a test case that increases coverage.

In order to cover a state, let assume that S is the set of states in the model, and s is one uncovered state. We need first to define a witness formula: $\{\Diamond\ (s\ \wedge\ \Diamond\ exit) | s\ \in S\}$. The corresponding LTL trap property is: $\{\Box\ (s\ \rightarrow \Box\ \sim exit) | s\ \in S\}$ (recall from section 2.3 that this is the negation of the witness formula).

Similarly, assuming fs and ts are the from-state and to-state, respectively of an uncovered transition, a witness can be automatically created: $\{\Diamond\ (\mathrm{fs} \wedge \mathrm{X}(\mathrm{ts}\ \wedge\ \Diamond\ \mathrm{exit}))\}$; and the corresponding LTL trap property is: $\{\Box(\mathrm{fs}\ \rightarrow \mathrm{X} \sim(\mathrm{ts}\ \rightarrow \Box \sim \mathrm{exit}))\}$.

4.3 Framework Realization with Virtual Time

In this section we discuss the mechanisms offered by RTEdge's Virtual Time and how we use them to realize the testing framework described in the previous section. A number of issues need to be discussed: How the testing framework actually sends signals, either Dependent or Independent System ones, to the Virtual Time simulation and how Virtual Time consumes them (section 4.3.1); How coverage data is collected and stored (section 4.3.2); How the testing framework is informed of transaction results (section 4.3.3); How RTEdge interacts with SPIN and how we use that feature (section 4.3.4).

Before entering into the details, it is important to realize that time in Virtual Time is not analogue but progresses in discrete steps where the size of the step varies depending on the difference between the current time and the time of the next event in the Virtual Time Event Queue. For example if the Virtual Time Controller is at a current time of 10,000 Virtual Time Units , where one Virtual Time Unit is equivalent to one nanosecond, and the next event in the event queue is at 17,000 Virtual Time Units, then the size of the next step is 7,000 Virtual Time Units so that the progress time cannot be less than 7,000 Virtual Time Units. Specifically, each time a transition in a capsule is triggered, time advances. And, each time Virtual Time advances time, it is able to perform tasks not related to the simulation proper, such as reporting to some entity outside of the simulation itself on various aspects of the simulation: e.g., what is the current state in each capsule, which transitions were last triggered, i.e., triggered since the last time Virtual Time advanced time. This is done without any impact on the simulation itself since that simulation (time) is implicitly paused.

4.3.1 Sending Signals to the Virtual Time Simulation.

Sending signals, either Independent or Dependent Signal Inputs, to Virtual Time is as simple as putting the signals in a queue from where Virtual Time fetches the next signal to be consumed by the simulation. Each signal deposited in the queue is specified with a signal arrival time (according to the time maintained by Virtual Time, not the time when the signal has been put in the queue) and data values it carries. This is a priority queue with signal time as a priority. This way, when Virtual Time advances time, it looks at the queue for a signal to consume at that time. If there is one, the signal is consumed. As discussed earlier, the sending of Independent Signal Inputs (and Dependent Signal Inputs) happens as long as the test case construction does not stop, i.e., the stopping criterion is not met: more signals are put in the queue as needed. Note also that in addition to fetching from the queue, Virtual Time also populates the queue: e.g., as communicating capsules (state machines) communicate through signals, when one capsule does communicate with another capsule, Virtual Time puts the corresponding signal into the queue.

Fig. 5 illustrates, under the form of a sequence diagram, that when it is time for Virtual Time to receive a new ISI (since Virtual Time knows about the details of the model, it knows when periodic ISIs are supposed to come) Virtual Time asks our test framework for a new ISI instance. The test framework checks if the ISI carries data. If yes, it randomly generates new data and then the data is added to Virtual Time's queue with a timestamp. The new ISI is then added (to Virtual Time's queue) with a timestamp (and possibly data). The following command is an example of the VT event with data:

```
{ 200, "int d = generateRandomWithProbabil-
ity_SendNewPlate_data();"};
```

where an integer variable `d` is randomly generated by calling the random generator function `generateRandomWithProbability_SendNewPlate_data()`, and the timestamp of the event is 200. The new ISI is added in the following event:

```
{ 200, " AddPLate_ET_CR1_SendNewPlate(d);"};
```

at timestamp 200, where the random integer `d` is passed as a parameter to the ISI function. However, if the ISI does not hold a data, then we only send the ISI, e.g. `{ 200, " AddPLate_ET_CR1_SendNewPlate();"};`

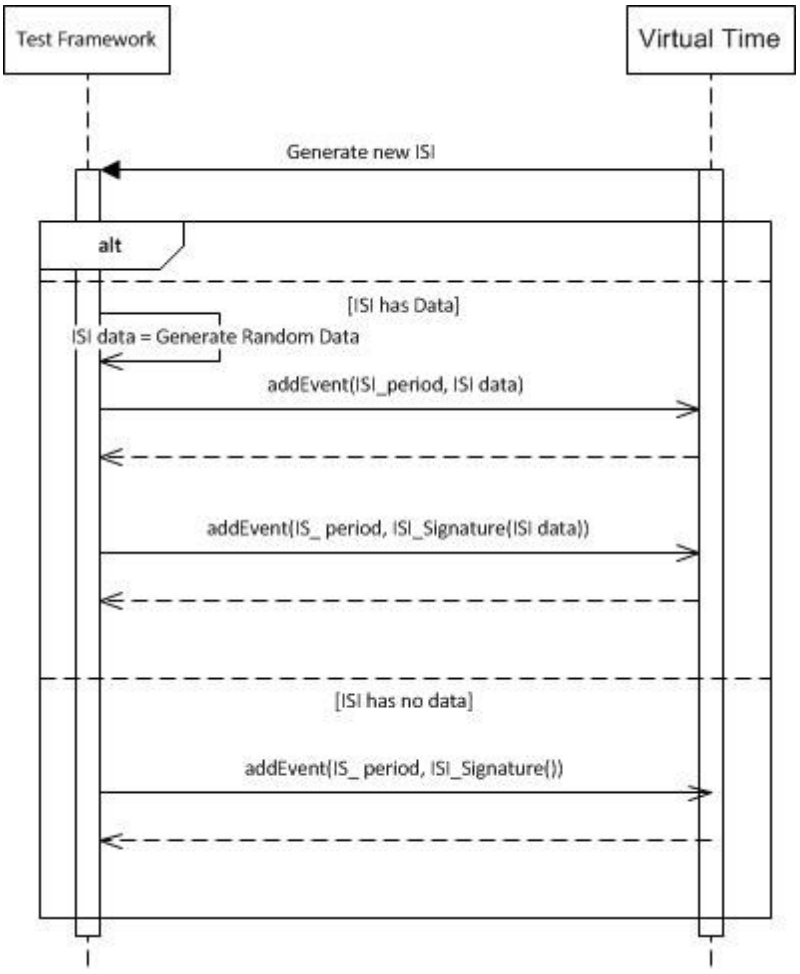


Fig. 5. – Generating Independent System Input

4.3.2 Collecting Coverage Data.

We set up the interaction between our testing framework and Virtual Time, thanks to Virtual Time's API, such that each time Virtual Time advances time (see earlier discussion on that), it reports on the current state of each capsule as well as on the last triggered transition (Remember from the beginning of section 4.3 that each time a transition is triggered, time advances.) For example, Virtual Time maintains and makes available an array, called `hrt_ac_arr[capsule_index]`, that holds characteristics for all the capsules. Virtual Time assigns a unique index to each capsule. To get the current state of a capsule at index `i` in the array, we use the following command: `hrt_ac_arr[i].cur_stateh`. Similarly, assuming `hrts.cur_ach` refers to a capsule, the signal currently being processed by the capsule is obtained by using command `hrts.cur_ach->cur_sigh`.

Our testing framework maintains a counter for each model transition and for each state transition to count the number of times each element is visited during the simulation. It is worth noting that, given information on states and transitions being triggered each time Virtual Time advances time, our framework can easily be adapted to support other coverage criteria such as transition pairs.

Fig. 6 shows a sequence diagram that illustrates how our test framework first collects the capsules' current states, then updates state information by comparing the capsules' previous state with the capsules' current state. After that the current state coverage is updated and transition coverage is updated depending on the current state, previous state, and current signal.

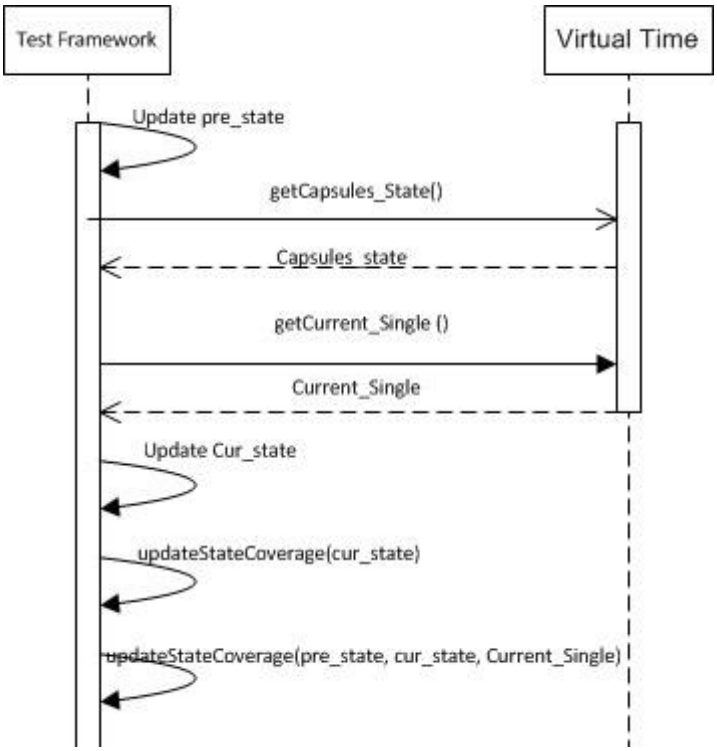


Fig. 6. – State and transition coverage update

4.3.3 Receiving transaction data.

RTEdge provides callback functions that allow us to extend the functionalities of Virtual Time and make sure our testing framework is informed when specific events occur during the simulation Virtual Time executes. Specifically, we use the following callback mechanisms.

We use the `transaction_started_callback` to collect the starting time of a transaction, i.e., a new Independent System Input (ISI) is being consumed by the SUT. Fig. 7 illustrates that when `transaction_started_callback` is called by Virtual Time, our test framework obtains the current time from Virtual Time and records the start time of the ISI. Similarly, the `transaction_finished_callback` function is used to set the end time of a transaction, i.e., an ISI led to a completed transaction: Fig. 8.

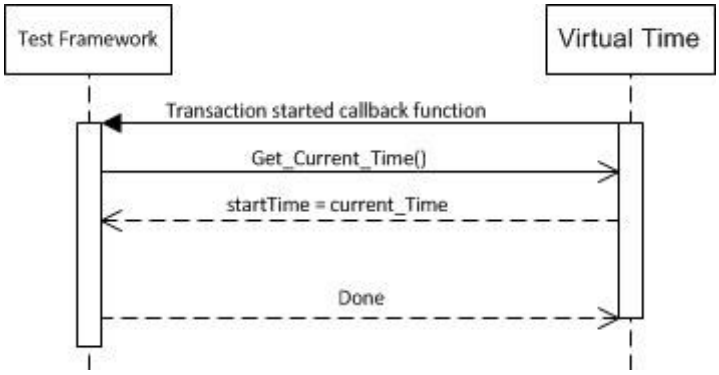


Fig. 7. – Transaction started callback function

These two events then allow the testing framework to calculate the duration of each transaction, and compare this duration to WCRT, the Worst Computed Response Time (recall section 2.4): if the duration value of a transaction is strictly greater than the WCRT of the ISI that triggered that transaction, then there is a fault in the model and the test case has failed. Specifically, this means that some estimates of execution times that were used during schedulability analysis are optimistic. Since the `transaction_finished_callback` function indicates the end of a transaction, this callback function also triggers the decision as to whether to stop or continue the construction of a test case. At the end of the transaction, our test framework updates coverage information (see the previous section).

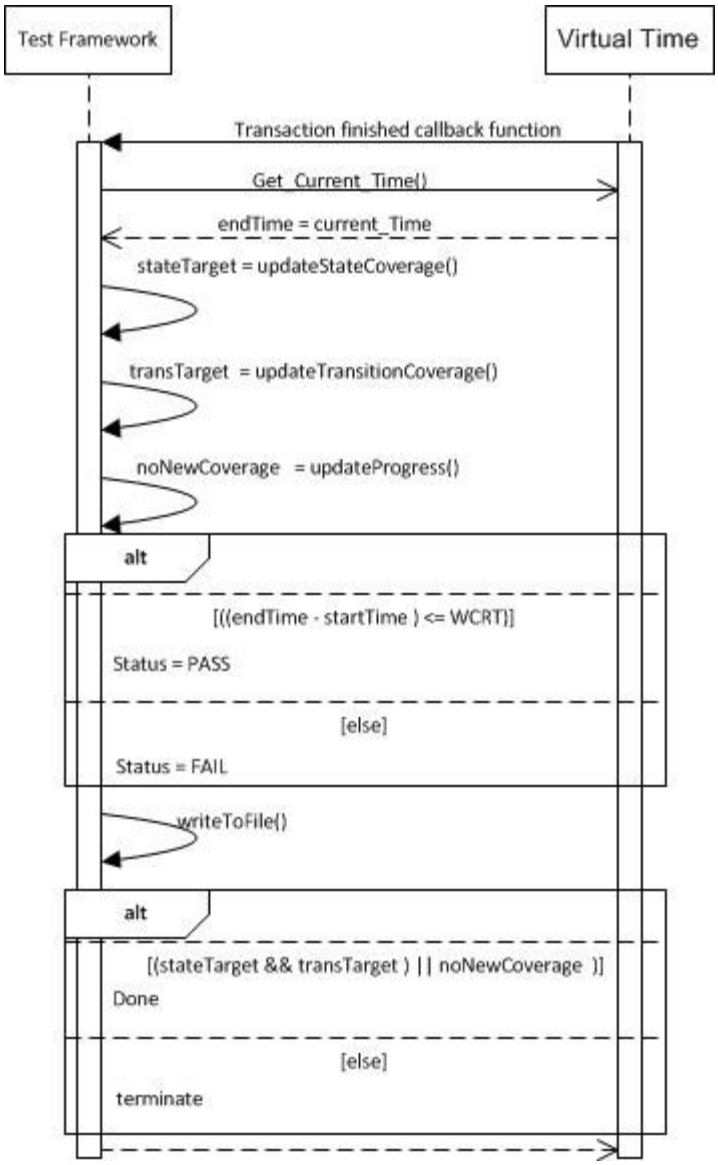


Fig. 8. – Transaction finished callback function

The designer has the possibility to set up timers on capsules in order to detect lack of progress (i.e., lack of change of state, lack of change in behaviour) within a capsule. Timers are similar to periodic events and detect every so often whether progress is being made in a capsule. If this is not the case, the `timer_expired_callback` function informs the testing infrastructure that the timer has expired during the simulation. This mechanism can be used to detect a liveliness problem in the model.

4.3.4 Framework Realization with Formal Link

RTEdge's Formal Link feature first validates the RTEdge model using Formal Validator to ensure the model conforms to RTEdge's meta-model. After that RTEdge Formal Link can automatically translate the given RTEdge model into an equivalent Promela model, assuming the tests performed by Formal Validator pass and that schedulability analysis check passes. When an RTEdge model is translated into Promela, each capsule is given an identification number (ID), each state in the model is given identification number (ID). In addition, an array of structures is defined where each structure element, that is called "state", holds the current state of each capsule in the model, the size of the array being equal to the number of capsules in the model. The capsule ID is the array index. The value of each structure element "state" in the array is the capsule's current state that is defined by a state ID. For instance, given that the array name is acinfo, assuming a `Feed_Belt` capsule has ID 0, that state "`waitForPLate`" in this capsule has ID 2, the current state for this capsule is defined as `acinfo[0].state == 2`. This will be useful to validate the counterexample since when we execute the model under SPIN the counterexample will show the current state at every execution point in the above format.

The SPIN model checker can then run on the Promela model and an LTL property as inputs and generate a counterexample (if the LTL property is not satisfied). Assuming a random test case generation has stopped because of lack of progress and leaves some states or transitions uncovered, our test framework automatically defines the LTL trap properties for each such uncovered state or transition and then runs SPIN separately for each trap property. If a trap property is violated the SPIN model checker generates a counterexample. The counterexample is an execution trace that shows how it has been

violated. Thanks to the integration between SPIN and RTEdge, the counter example generated by SPIN is translated into a path in the RTEdge model for the engineer to view. The RTEdge counterexample can be (manually) validated by tracing the `acinfo[capsule'ID].state` variables along the path of the trace executions until the last state of the trace where the violation of the trap properties occurs; thus, this counterexample will be considered as evidence that the corresponding state or transition can in fact be covered and the generated counterexample is considered as a test case to cover the state or transition.

Chapter 5 - **Case Studies**

We have selected two case studies, which are the Production Cell and Elevator Control System. We used RTEdge to design the specification for each case study, and then we used our test framework on the two cases studies. In this chapter, we will describe the Production Cell case study (section 5.1), and the Elevator Control System (section 5.2).

5.1 Production Cell

We describe the specification of the case study (section 5.1.1) and then discuss its design using RTEdge (section 5.1.2). Finally, we discuss the result of using our testing framework (sections 5.1.3 and 5.1.4).

5.1.1 Production Cell Case Study Overview

The Production Cell case study (Fig. 9) is a realistic industry application [48] and was originally used to demonstrate the benefits of formal methods for the specification and verification of critical real-time applications. It consists of two belts (the feed belt and the deposit belt), an elevating rotary table, a robot, a press, and a crane. The production cell cycle is described as follows. The feed belt transports a plate to the table, which is positioned between the feed belt and the robot. The table needs to rotate and move up to the right position (angle and height) so the robot can pick the plate up. Once the plate has been picked up by the robot (using its first arm), the table rotates back and moves down to receive another plate. The robot consists of two arms, one arm is always used for loading the press with blank plates that are taken from the feed belt, and the other arm is used for unloading the press and loading the deposit belt. The arms are located at different heights, and they always rotate together. The first robot arm loads a plate to the press

when the press is in its middle position. Then the press moves up to press a plate. Once the plate is forged, the press moves down to its lower position, so the second robot arm can unload the plate. Once unloaded, the press moves up, back to its middle position. The second robot arm transports the plate to the deposit belt. After that the crane takes the plate from the deposit belt.

The production cell implementation contains 14 sensors and 13 actuators. Actuators are used to switch the motors on and off or change their directions, and sensors return value to the control program about the system state. The specification of the production cell has three kinds of non-functional properties. The safety requirements are the most important ones: collisions between devices must not occur; plates must be dropped in the safe area, two consecutive plates must be transported at an adequate distance to avoid placing two plates in the press, a movability restriction is implemented to prevent any machine from moving further than what it is allowed. The second important requirement is the liveness of the system: each plate transported by the feed belt should eventually be forged and arrive to the end of the deposit belt. The third requirement is that the design of the Production Cell should be flexible and could be easily be modified to similar Production Cells.

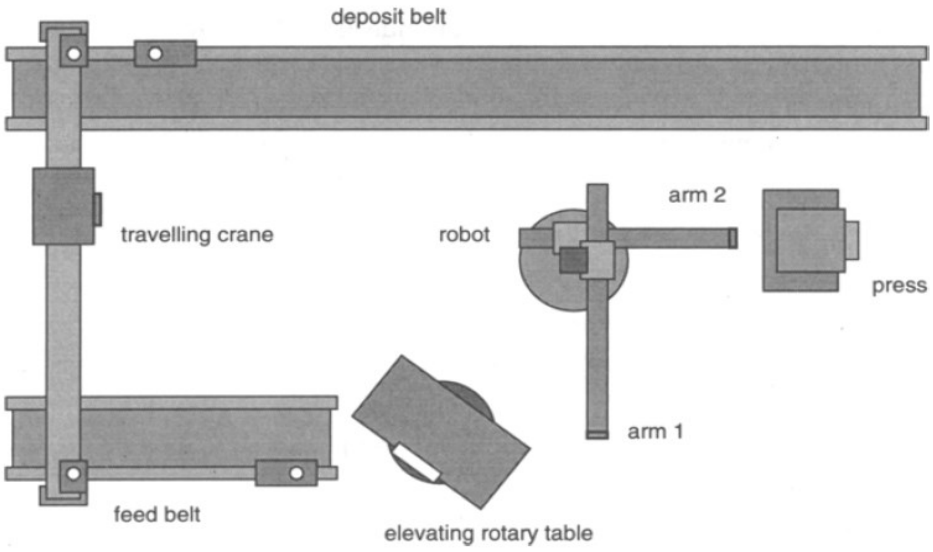


Fig. 9. –A structural overview of the Production Cell [48]

5.1.2 RTEdge model of the Production Cell

We designed the production cell using RTEdge. Specifically, we modeled the five major capsules: feed belt, rotary table, robot, press, and deposit belt; and omitted the crane as, similarly to others [49], we found it was unnecessary for our purpose. These are individual capsules since they all have to achieve their task independently and they must ensure the correct processing by communication. We assumed movements of a plate in the Production Cell take time and, similarly to others [49], we assumed specific movements (e.g., travelling on the feed belt to the table) take fixed amounts of time, and different movements require different durations. For example, the forging of a plate in the press needs more time than moving the robot.

We simplified the interactions between the controlling software we model and its environment made of sensors and actuators. Specifically, we replaced the sensors in the model with actions in the model thereby simulating how time elapses during those movements. As a result, when a capsule in some particular state needs a sensor value to pro-

ceed with the simulation, then it will wait a specific amount of time in its current state before moving to the next state (where the sensor data is used), thereby simulating that there might be a delayed response by the sensor. For instance, the robot must read a sensor in order to bring its fist arm next to the press.

Fig. 10. shows an overview of the RTEdge Production Cell model, with its capsules and their communication links. The five primary capsules, representing the belts, table, robot and press are located at the center of the model. The other (surrounding) capsules simulate the environment, such as the arrival of a plate or movements, as discussed above.

The feed belt communicates with an external capsule to simulate the receipt of a new plate, and it communicates with the rotary table to simulate a plate moving to the table. The press only communicates with the robot. The robot has to communicate with the elevating rotary table and the deposit belt since it also exchanges plates with these capsules.

RTEdge models the behavior of a capsule as a state model. We designed all the communicating state machines of the model (i.e., the capsules) such that certain safety requirements are enforced. Specifically, the capsules communications ensure that specific sequences of signals will be ignored. For instance, the table will not accept a signal specifying the arrival of a new plate if it is already holding a plate (i.e., the table waits until it is ready to accept a new plate): as a result, when this is the case, the feed belt stops and waits until the table is again ready to accept a new plate. Table 1 shows each capsule and its corresponding number of states and transitions.

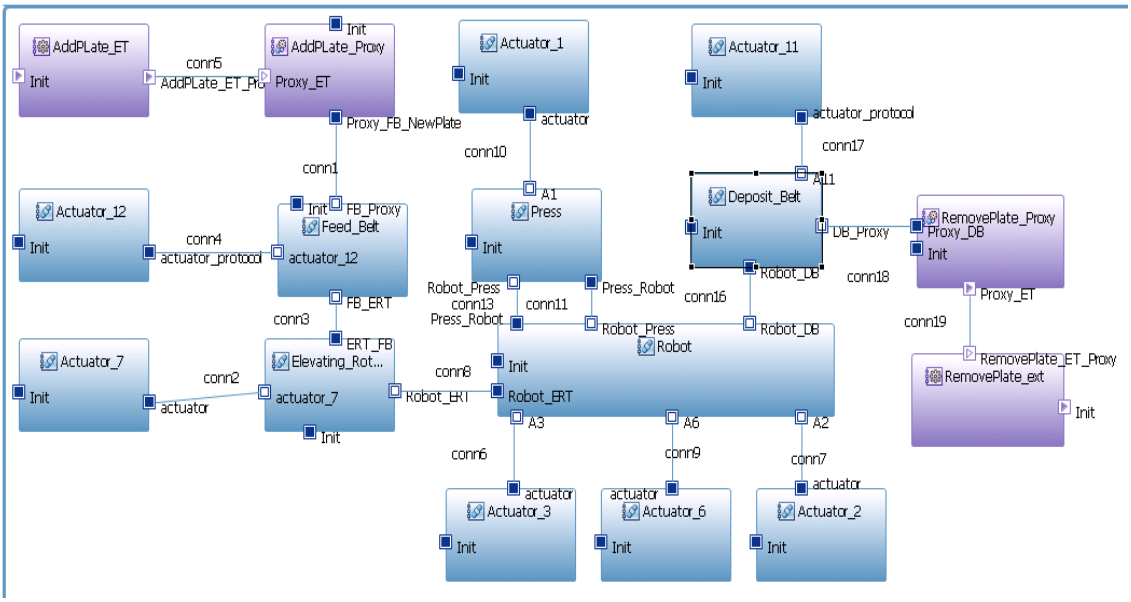


Fig. 10. –Overview of the RTEdge model for the Production Cell

Table 1.—Number of states (total 138) and transitions (total 168) in the capsules of Fig. 10

Capsule Name	States	Transitions	Capsule Name	States	Transitions
Feed Belt	7	7	A6	9	16
Elevating Rotary Table	11	11	A7	5	8
Robot	47	47	A11	4	4
Press	14	14	A12	4	4
Deposit Belt	7	7	Add_Plate External	1	1
A1	6	12	Add_Plate Proxy	2	2
A2	9	16	Remove_Plate Proxy	2	2
A3	9	16	Remove_Plate External	1	1

When the `Feed_Belt` receives a new plate from the proxy on the `FB_Proxy` port, this makes the belt transition from `waitForInput` to `ReceivePlate`: Fig. 12 (a). As a result the belt has a new plate: the transient state `RecievePlate` has an activity, which code is shown in Fig. 11 (the code updates the feed belt's local variable). Once the local variables are updated, an activate signal is sent to the feed belt's actuator to start moving. As a result the belt transitions from `RecievePlate` to `ActivatyActuator`. When `ActivatyActuator` receives an `ack1` signal on port `actuator_1`, the belt starts moving: this is modeled by the `BeltMoving` state. To simulate how time passes as

plates move around, we created transients states with attached activities that make time pass. We specified a transient state for each plate movement. `BeltMoving` simulates the movement of a plate from the beginning of the feed belt to its end. (Recall transient states are distinguished from stable states by their dash border and must be further specified with an activity which code executes while in the state.) The feed belt sends another signal to stop the actuator (i.e., the motor), when the plate reaches the end of the belt, resulting in the belt transitioning from `DeactivatyActuator` The actuator will send an `ack-stop` signal. As a result the feed belt will be in the `ReadyToSendPlate` state. At this point, the feed belt has a plate at its end, so it sends a signal to the rotary table, informing it that a plate is in the right position. After that, the `Ready` signal is sent on port `ERT_FB` to make the table transition from state `waitForInput` to state `ProcessNewPlate`: Fig. 12 (b), bottom right. As a result, the table sends the `SendPlate` signal to the belt on port `FB_ERT`, making the belt's capsule transition to state `SendPlate` where the belt updates its local variables. After that the plate is sent to the table, resulting in the belt transitioning to the `WaitForInput` state. As a result the belt is ready to accept a new plate and the table has a new plate.

```
processPlate.cpp | RecievedPlate.cpp | BeltMoving.cpp | Process_To_ERT.cpp | Process
 7 #include <RTEdgeModel/types/Feed_Belt_AC.h>
 8
 9 // -- USER #include LINES BEGIN --
10 // -- USER #include LINES END --
11
12 #ifdef __CINT__
13 static
14 #endif
15 void Feed_Belt_AC::ProcessNewPlate( Feed_Belt_AC * locals_ptr )
16 {
17     // -- USER ACTIVITY CODE BEGIN --
18
19     locals_ptr->counter++;
20     locals_ptr->PlateOnTable = true;
21     // -- USER ACTIVITY CODE END --
22 }
23
```

Fig. 11. Feed Belt activity code to process a new plate.

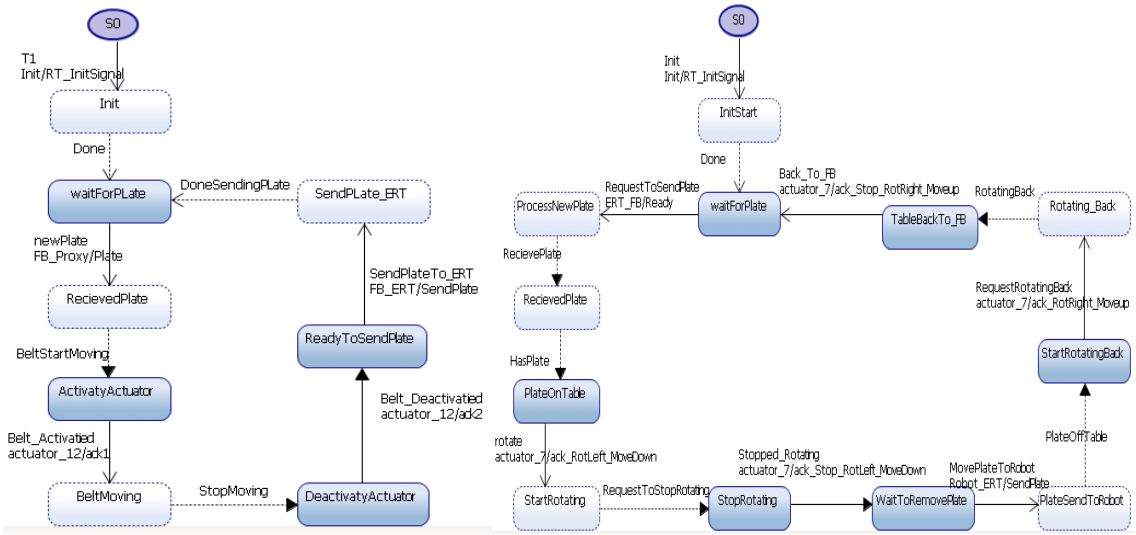


Fig. 12–Details of the feed belt (a) and rotary table (b) capsules

In Fig. 12 (b), when the table is ready to receive a plate (state `ProcessNewPlate`), it sends signal `SendPlate` on port `FB_ERT`. As a result the table transitions from `ProcessNewPlate` to `RecievedPlate` where the local variables are updated. The table sends a signal to its actuator to start rotating, resulting in the table transitioning from `RecievedPlate` to `PlateOnTable`. Once `PlateOnTable` receives an `ack_Rot_Left` on port `actuator_7`, the table transitions to the `StartRotating` state. `StartRotating` simulates the table rotation from the feed belt position to robot position. After a fixed delay (again, we simulate movement with activities on transient states), we consider the table is in the right position to service the robot. Then the table sends another signal to stop its actuator (transition from state `StartRotating` to state `StopRotating`), the actuator will send an `ack-stop` signal. At this point, the table has a plate at the right position to service the Robot. The table sends the robot a signal to inform it that it has a plate in the right position.

5.1.3 Result: random test case construction

We used our framework to derive test cases from the Production Cell model. At the same time, since our framework can detect deviations from Worst Computed Execution Times or some liveliness problems, the framework participates in the verification of the model.

Recall (end of section 4.2.5) that our framework requires three different inputs. In a first experiment, we set those inputs as follows: The number of times a state/transition needs to be covered is set to one (`minCover=1`), the observation window (i.e., number of transactions) for studying coverage progress is set to three (`obsWindow=3`), and the minimum number of new states/transitions that needs to be covered in the observation window is set to one (`minIncr=1`).

With this first setting, the automated test case generation created one test case with one transaction, i.e., one instance of ISI `AddPLate_ET_CR1_SendNewPlate()`, specifying that one plate is provided to the production cell. This ISI, which does not carry any data, is sent at time zero. The stated coverage goal was achieved, there was no need for additional transactions, no need to observe coverage progress over past the observation window (recall the stopping criterion).

In a second experiment, we kept the values of the last two parameters and required that each state and each transition be covered at least three times: `minCover=3`, `obsWindow=3`, `minIncr=1`. The intent was to study the performance of our approach on a more demanding objective.

The result of the test case generation is reported in Table 2. We generated one test case involving three Independent System Inputs (i.e., three plates). The following three ISI are generated:

- `{0,"AddPLate_ET_CR1_SendNewPlate();"},`
- `{4000,"AddPLate_ET_CR1_SendNewPlate();"},`
- `{8000,"AddPLate_ET_CR1_SendNewPlate();"}.`

The resulting transactions took 4300 or 3950 nano-seconds to execute. The transaction status is passed if the transaction's execution time is smaller than the WCRT computed by the schedulability analysis and no timeout was reported. However, the transaction status is failed if there is a fault of the model (recall the end of section 4.1), a timeout is reported through the simulation, or the transaction's execution time is greater than the WCRT computed by the schedulability analysis.

All transactions passed (last column), indicating that the duration of each transaction was found to be smaller than the WCRT computed by the schedulability analysis and no timeout was reported (no call to the `timer_expired_callback` function). All transactions also have passed because no fault was found in the model, specifically there was no unspecified signal through the simulation of the model, resulting in the pass status for all transactions. Also, the attached assertions had passed through the model simulation, indicating that there was no undesired behaviour in the model through the simulation, resulting in all transactions to pass. Since our test framework indicates that there are no faults in the model, we can conclude that all safety properties of the production cell are satisfied through the verification of the model.

The table also shows that the first transaction covers five (new) states: states `RTEdgeModel::A7_AC::S1`, `RTEdgeModel::actuator_2::S1`, `RTEdgeModel::A3_AC::S1`, `RTEdgeModel::A6_AC::S1`, `RTEdgeModel::A1_AC::WaitforInput` are the first stable states in the Production Cell's actua-

tors state machines. These states receive many activate/deactivate, or start/stop signals from the Production Cell's capsules. Therefore, they are covered from the first transaction. The table reports that no (new) transition is covered by the first transaction. This refers to the coverage objective, which is to cover each transition at least three times. The table indicates that this objective has not been met for a single transition. However, obviously, transitions have been covered in the mode. Actually, all the transitions have been covered once. Since the state coverage goal is not achieved, the stopping criterion is not true and test case construction proceeds with a second ISI (i.e., plate): eight new states are covered which are:

1. `RTEdgeModel::Feed_Belt_AC::waitForPLate`
2. `RTEdgeModel::ERT_AC::waitForPlate`
3. `RTEdgeModel::A12_AC::S2`
4. `RTEdgeModel::AddNewPlate_Proxy::WaitForPlate`
5. `RTEdgeModel::Robot_AC::WaitForInput`
6. `RTEdgeModel::Press_AC::WaitForInput`
7. `RTEdgeModel::Deposite_Belt_AC::S2`
8. `RTEdgeModel::A11_AC::S2`

The table indicates that no (new) transition is covered, meaning that no new coverage objective has been met for a single transition. The main reason for this is that each ISI does not exercise any transition more than once in our Production Cell model. Test case construction therefore continued with a third ISI: 125 new states are covered and 168 transitions are covered, since our design of the Production Cell dose not exercise any transition more than one for each ISI, all the transitions are covered one more time when the third ISI is executed, which leads to each transition being executed three times at this stage. At

this stage, each of the state/transition was covered at least three times, resulting in the test case construction to stop.

Table 2.—Resulting test case

ISI #	Transaction duration (nano-sec.)	Number of				Status
		New states	States covered	New transitions	Transitions covered	
1	4300	5	5	0	0	PASS
2	3950	8	13	0	0	PASS
3	3950	125	138	168	168	PASS

5.1.4 Result: SPIN model checker test case construction

Our online test framework was successfully able to achieve the state and transition coverage criteria for the production cell case study as it has been shown in the previous section. However, since we want to use the SPIN model checker to generate counterexample for the given trap property to cover some uncovered state/transition through our framework, we have selected one state and one transition in the production cell model, assumed they had not been covered through our online framework, and then used our framework to generate counterexamples, i.e., test cases covering those elements. Recall section 4.3.4. We have chosen the `Feed_Belt` capsule to select one of its states to be covered using SPIN model checker. We have assumed that the state `waitForPLate` has not been covered. The `Feed_Belt` capsule unique id is 0 and the id of state `waitForPLate` in capsule `Feed_Belt` is 2; thus, if the current state of the `Feed_Belt` capsule is `waitForPLate`, then the `acinfo` array, as we have discussed in section 4.3.4, is such that `acinfo[0].state = 2`. The predicate for the state variable is automatically defined by our framework (section 4.3.4) in Promela as:

```
#define predicate0 /*noCheck*/
```

```
(acinfo[FEEDBELT_CR1_FEED_BELT_AC].state ==
    Feed_Belt_AC_waitForPLate)
```

After that the Trap property is automatically defined as:

```
![](predicate0->[]!tgpredsendnewplate1);
```

where `tgpredsendnewplate1` is the variable in the Promela code that identifies the model has run to its competition.

Fig. 13 shows the counterexample returned by SPIN for this state's trap property. The red line in Fig. 13, line 573, shows the last `acinfo.state` appears before the SPIN claims that the LTL property is violated. The current state that is shown in the counterexample is `acinfo[0].state == 2`, which is the same above formula for `waitForPLate` state for the `Feed_Belt` capsule. This confirms that the `waitForPLate` state is visited through the model execution. Thus, the execution sequences shows the state is reach through the execution of the model.

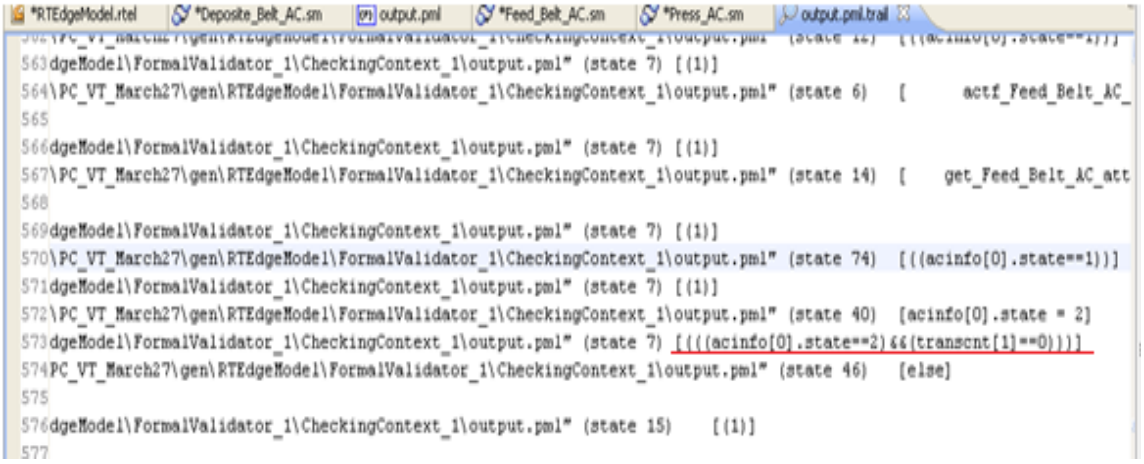


Fig. 13. – Counterexample of the state's trap property

To use our framework to cover a transition using the SPIN model checker, we have assumed that transition `RequestToSendPlate` in the `Elevating_Rotary_Table` capsule has not been covered. Since the transition is defined using its from-state and to-state

in Promela, we have defined two predicates to hold. The id of capsule `Elevating_Rotary_Table` is 1, the from-state of the transition is `waitForPlate`, which has id equal to 2; so that if the current state of the `Elevating_Rotary_Table` capsule is `waitForPlate`, then the `acinfo` array, as we have discussed in section 4.3.4, is such that `acinfo[1].state = 2`, and the to-state of the transition is `ProcessNewPlate`, with id 3, i.e., `acinfo[1].state = 3`, if the current state is `ProcessNewPlate` for the `Elevating_Rotary_Table` capsule. As a result the following predicates are defined:

```
#define predicateFrom0 /*noCheck*/
(acinfo[ELEVATING_ROTARY_TABLE_CR1_ERT_AC].state ==
    ERT_AC_waitForPlate)
#define predicateTo0 /*noCheck*/
(acinfo[ELEVATING_ROTARY_TABLE_CR1_ERT_AC].state ==
    ERT_AC_ProcessNewPlate)
```

After we have defined the predicates, the LTL trap property is defined as:

```
![]( predicateFrom0 ->X!( predicateTo0 -
    >[]!tgpredsendnewplate1))
```

The transition's trap property has generated a counterexample as shown in Fig. 14. The first red line, at line 524,shows the current state at this point of the execution is `waitForPlate` of the `Elevating_Rotary_Table` capsule, since `acinfo[1].state = 2`, as it has been discussed above. The next current state appears at line 527 in Fig. 14, this line shows the `acinfo[1].state == 3` which means the to-state `ProcessNewPlate` of `Elevating_Rotary_Table` capsule is visited next. At this point of the exe-

cution terminates because the LTL property is violated and the transition is visited because the from-state is visited first and then the to-state is visited.

Fig. 14. – Counterexample of the Transition's trap property

5.2 Elevator Control System

We describe the specification of the case study (section 5.2.1) and then discuss its design using RTEdge (section 5.2.2). Finally, we discuss the result of using our testing framework (sections 5.2.3 and 5.2.3).

5.2.1 Elevator Control System Study Overview

The Elevator Control system controls at least one elevator. The system schedules elevators, responding to requests from users at various floors and within the elevators. It also controls the motion of the elevators between floors. The control system should be configurable for buildings with different number of floors, different numbers of elevators; we have assumed that the building has four floors, and two elevators. Each elevator has:

- A set of elevator buttons used by the users to select the destination floor;
- A set of elevator lamps corresponding to each button, indicating the floors that have been selected as destinations in the current trip;

- An elevator position lamp indicating the current floor of the elevator during the trip;
- An elevator motor controlled by commands to move and stop. A status sensor indicates when asked by the control system whether the elevator is moving or stopped;
- Emergency brakes which are triggered in different unsafe conditions;
- An elevator door controlled by commands to open and close the door;

Each floor has:

- Up and down floor call buttons, used by a user to request an elevator for going in a certain direction (up or down), except the bottom floor and the top floor.
- A corresponding pair of floor lamps indicating the direction(s) already requested. The bottom and top floors only have one lamp.

Each elevator has a pair of direction lamps to indicate whether an arriving elevator is heading in the up or down direction. There is also an arrival sensor at each floor in each elevator shaft to detect the arrival of an elevator at the respective floor.

For system safety certification, the emergency brake will be triggered and the elevator will be forced to stop under any unsafe conditions, such as:

- if an elevator is commanded to stop but will not stop at the desired floor;
- if an elevator is commanded to move but will not move;

Once a hall call button is pressed by a user, one of the elevators will be dispatched to serve the request. After entering the elevator, the passenger presses a selected elevator button to indicate his/her destination floor. The elevator then moves up/down to the destination floor. To increase the utilization of the entire system, the elevator may stop at

other floors on the way to service other requests for the same movement direction. The elevator stops at the nearest requested floors for all the requests assigned to it by the controller. When all calls are served, the elevator stops and waits for the next call to arrive, and the above cycle repeats again. The control policy should comply with a common rule that requests are to be served on a first come-first-serve basis, except for other requests in the same direction that could be served simultaneously. So, when the elevator is heading in a particular direction, the coordinator will assign to it other requests heading in the same direction along the way. An elevator will move up and down as long as there are outstanding requests to be served.

5.2.2 RTEdge model of the Elevator Control System

Fig. 15 shows an overview of the RTEdge Elevator Control System model, with its capsules and their communication links. Table 3 shows the total number of states and transitions for each capsule. The following are description for the main capsules:

- `Controlling_cab`: This capsule concurrently controls the two cabs movements between the four floors of the building. Once the `Controlling_cab` receives a request from the `Coordinator_Request`, it calculates the cab direction. After that it sends a request to `Direction_Display` to show the cab direction, then it starts processing the request to the specific cab; it sends a request to the Elevator engine to start moving to the given destination; concurrently, this capsule will be also waiting for a new request to be processed by the other cab. While the elevator is travelling, the elevator controller sends a signal to the `Floor_Number_Display` capsule to indicate the current floor of the elevator during the trip. Once the elevator reaches (i.e., is near) its destinations, it sends

first a request to stop the engine at a specific floor. Secondly it sends a request to open the elevator doors. And it sends requests to floor or hall buttons to turn their lights off. It also sends a request to `Direction_Display` to turn the cab direction light corresponding to the cab direction. After a specific time, it sends a request to close the elevator doors.

- `Coordinator_Request`: This capsule stores the requests' data in a queue, and it coordinates the dispatching of request to the `Controlling_cab` capsule. When it receives a new request from the hall or the floor call buttons, it adds the request to a queue, and then it determines the next request to be processed, and the best cab to process this request. After that this capsule sends the request to the `Controlling_cab` capsule to process it, and then the request is removed from the queue. After that it starts processing the next request in the queue.
- `Floor_Buttons`: As discussed previously, our case study has two elevators and four floors. Consequently, we have two series of four floor request buttons labeled one through four. When a floor button is pressed (new ISI), this capsule receives a press signal indicating a new request is sent; after that this capsule turns the pressed button light on, and then this capsule sends the new request to `Controlling_cab` in order to start processing the new request. Once the request is finished processing, this capsule receives a turnLightoff signal from the `Controlling_cab`.
- `Hall_Buttons`: Given the configuration of our case study, we have two series of floor buttons. Each floor has one up and one down button, except the top floor where there is only a down button and the bottom floor where there is only an up

button. When a hall button is pressed (new ISI), this capsule receives a press signal indicating a new hall request is received along with floor number and a direction; as a result this capsule sends a new request to `Coordinator_Request` to start processing the request. Also, this capsule turns the button's light on once it receives a new request. Once the request is finished processing, this capsule receives a turnLightoff signal from the `Controlling_cab`.

- EmergencyBrake: This capsule works in the emergency situation to stop the elevator. If this capsule receives a stop signal from `Controlling_cab` to stop a cab, then this capsule enforces stopping this particular cab. Also, if this capsule receives a move signal from `Controlling_cab` in order to enforce moving a cab, then this capsule enforces this particular cab to start moving.
- `Elevator_Engine`: The elevator engine is responsible for moving an elevator cab up and down between floors. `Controlling_cab` interacts with the `Elevator_Engine` by sending it a move signal that specifies in what direction the engine should be going in. When the elevator engine starts moving, this capsule waits a specific amount of time at the current state to simulate the movement of the elevator, and then it updates the arrival sensor value to indicate the arrival of an elevator at the respective floor. A stop signal is sent to stop the engine when the required floor is reached. A status sensor that indicates when asked by the control system whether the elevator is moving or stopped is modeled by a Boolean variable: whenever a move signal is received this Boolean variable is switched to true; otherwise, it set to false.

- `Direction_Display`: The interior of each elevator cab has a display that indicates the current direction of the elevator cab; it is either up or down. The `Controlling_cab` interacts with this display by sending it a signal that tells it which direction to display.
- `Floor_Number_Display`: The interior of each elevator cab has a display that indicates to its passengers which floor the elevator cab is currently on. The `Controlling_cab` interacts with this display by sending a signal that tells it which floor number to display.
- `Door_Operator`: This capsule coordinates the opening and closing of the elevator doors. `Controlling_cab` interacts with this capsule by sending signals to open or close the doors and by receiving an Ack signal when the doors have been completely opened or closed.
- `Opening_Device`: this capsule simulates the mechanical device that opens and closes the cab's doors. Once it receives the signal from the `Door_Operator` to open the door, it waits a specific amount of time at the current state, and then it sends an Ack signal to the `Door_Operator` capsule to confirm that the cab's doors are open. Similarly, when this capsule receives a close signal from the door operator capsule, it waits at the current state a specific amount of time, and then it sends an Ack signal to confirm that the doors are closed.
- `HallRequest_ext`: this capsule periodically generates a new hall button request (ISI) along with its data, which are floor number and direction.
- `FloorRequest_ext`: this capsule periodically generates a new floor button request (ISI) along with its data, which are the cab ID, and floor number.

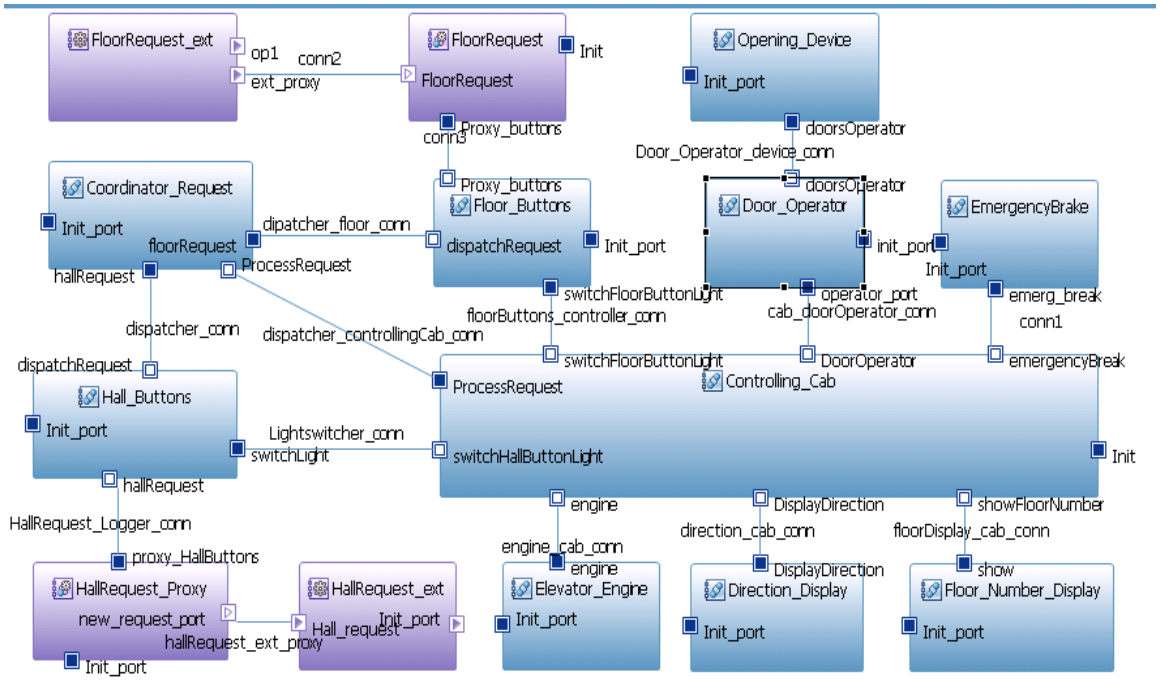


Fig. 15–Overview of the RTEdge model for the Elevator Control System

Table 3.—Number of states (total 73) and transitions (total 86) in the capsules of Fig. 15.

Capsule Name	States	Transitions	Capsule Name	States	Transitions
HallRequest_proxy	3	3	Hall_Buttons_AC	6	8
Floor_Number_Display_AC	2	2	Elevator_Engine_AC	7	9
Door_operator_AC	7	7	Coordinator_Request_AC	7	8
Controlling_Cab_AC	23	28	Direction_Display_AC	2	2
Floor_Buttons_AC	6	8	EmergencyBreak_AC	3	4
FloorRequest_proxy	3	3	OpeningDevice_AC	4	4

When the `Hall_Buttons` capsule receives a new request from the proxy on the `hallRequest` port, this makes the `Hall_Buttons` transition from `waitForRequest` to `ProcessRequest`: Fig. 16. Once the local variables are updated in the transient state `ProcessRequest`, the request is sent to the `Coordinator_Request` capsule. As a result the capsule transitions from `ProcessRequest` to `WaitForAck`. After that, an `Ack` signal on port `dispatchRequest` is received by `Hall_Buttons` to confirm that the request is added to the queue in the `Coordinator_Request`. When a `TurnLightOn` signal on port `switchLight` is received, the `Hall_Buttons` capsule transitions from

WaitForDisplayLight to ProcessLightOn. The transient state ProcessLightOn simulates turning the light on for the pressed button; as a result, the capsule transitions from ProcessLightOn to WaitForRequest. The capsule is ready now to accept a new request.

Fig. 17 shows the Coordinator_Request state machine. When this coordinator capsule receives a new request from the Hall_Buttons capsule, i.e., through signal sendRequest on port hallRequest, this coordinator capsule transitions from WaitingForRequest to addRequestToQueue. After the request is added to the queue, the coordinator capsule transitions from addRequestToQueue to DetermineNextRequest to start processing the new request. The transient state DetermineNextRequest has an activity code that calculates the next request that complies with a common rule that requests are to be served on a first come-first-serve basis, except for other requests in the same direction that could be served simultaneously. As a result the capsule transitions from state DetermineNextRequest to DetermineRequestCab. Once the next request is determined, and the cab, which will process the next request, is determined, the next request signal is sent to the cabs controller to start processing the next request. As a result the coordinator capsule transitions from DetermineRequestCab to WaitForAck_Cab1 or WaitForAck_Cab2 depending on which cab is selected to serve the next request. When the coordinator capsule receives an Ack signal to confirm that the cab has started processing the request, the coordinator capsule transitions to RemoveRequest. The transient state isQueueEmpty checks if the queue is empty or not. If the queue is not empty, then the coordinator capsule transitions form RemoveRequest to DetermineNextRquest to start processing the next request. If

the queue is empty the coordinator capsule transitions from `RemoveRequest` to `WaitingForRequest`; At this point, the coordinator has no request in the queue, and it is ready to accept a new request.

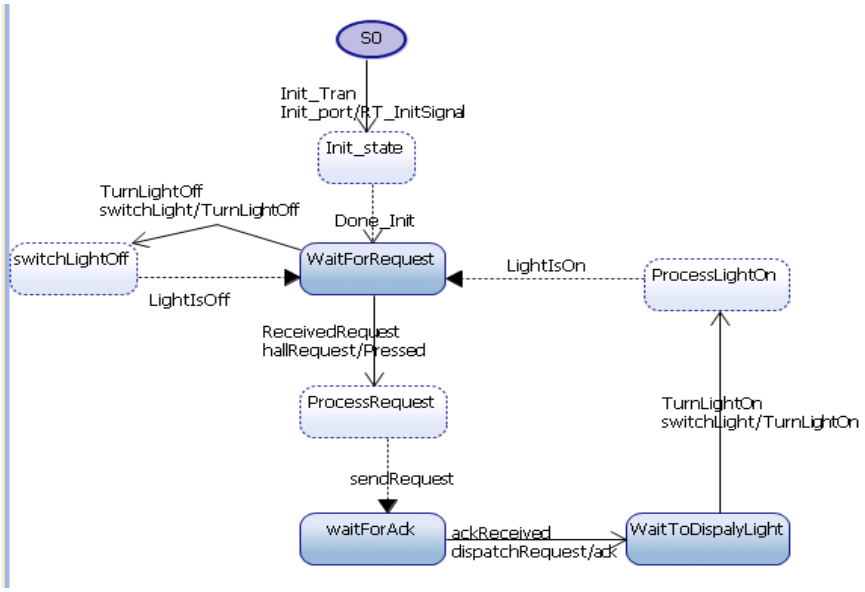


Fig. 16–Details of the Hall_Buttons capsule

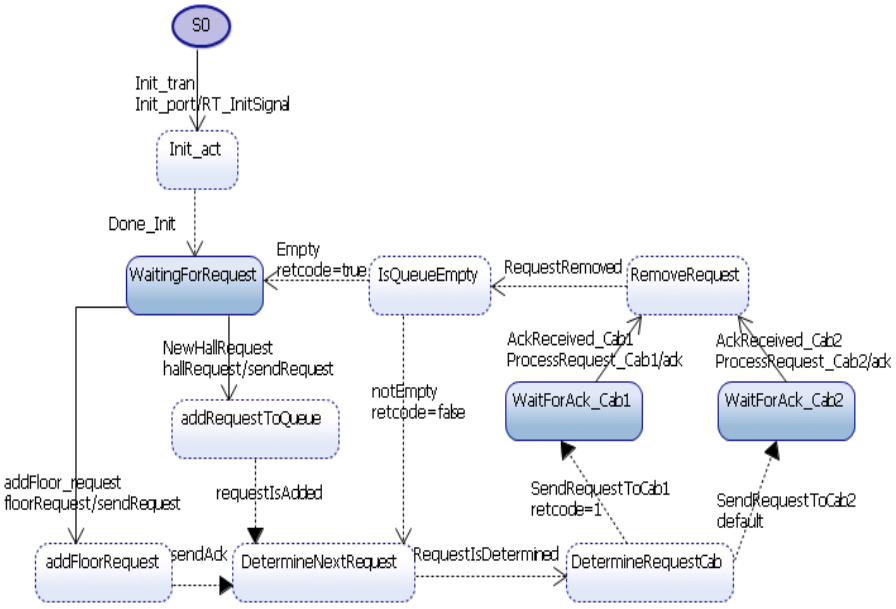


Fig. 17- Details of the Coordinator_Request capsule

5.2.3 Result: random test case construction

We used our framework to derive test cases from the Elevator Control System model. At the same time, since our framework can detect deviations from Worst Computed Execution Times, model faults as we have discussed at the end of section 4.1 or some liveliness problems, the framework participates in the verification of the model.

Recall (end of section 4.2.5) that our framework requires three different inputs. We set those inputs as follows: The number of times a state/transition needs to be covered is set to one (minCover=1), the observation window (i.e., number of transactions) for studying coverage progress is set to two (obsWindow=2), and the minimum number of new states/transitions that need to be covered in the observation window is set to one (minIncr=1).

Our model has two ISIs, which are the floor button request and the hall button request. In real time, these system inputs are aperiodic; however, RTEdge only sends each ISI periodically according to a user-defined period, hence, the ISIs are sent periodically depending on a period we define. The period of an ISI must be equal to or greater than WCRT of the transaction this ISI triggers. Therefore, in order to define a period for ISIs, we have first estimated the required time for our model to complete a transaction for ISI, then we performed a schedulability analysis of the model, which returned the WCRT. We set the period of each ISI to the computed WCRT which is six seconds for both ISIs. The result of the test case generation is reported in Table 4. We generated one test case involving five ISIs:

1- The first ISI comes from the hall buttons with the following randomly generated data: Direction = 1, and Floor number = 2;

2- The second ISI comes from the floor buttons with the following randomly generated data: CabID = 0, and Floor number = 4;
3- The third ISI comes from the hall buttons with the following randomly generated data: Direction = 1, and Floor number = 1;
4- The fourth ISI comes from the hall buttons with the following randomly generated data: CabID = 1, and Floor number = 3;
5- The fifth ISI comes from the hall buttons with the following randomly generated data: Direction = 0, and Floor number = 3;

The resulting transactions for the hall buttons requests took 1800, and 1950 nano-seconds to execute; also, the resulting transactions for floor buttons requests took 2000 or 2150 nano-seconds to execute. All transactions passed (last column in Table 4), indicating that the duration of each transaction was found to be smaller than the WCRT computed by the schedulability analysis and no timeout was reported (no call to the `timer_expired_callback` function). Also, there was no unspecified signal through the model simulation. The first ISI is a request from the Hall buttons at floor number two, where the direction is up; this transaction has covered 56 new state and 54 new transitions. These states and transitions are covered from the first transaction since they are in the path in the elevator model that is required to serve the hall request. Also, since our coverage objective is to cover each state and transition at least one, theses states and transitions are covered at least once through the model simulation for the first transaction. The table indicates that this objective has not been met because the state and transition coverage goal is not achieved, the stopping criterion is not true and test case construction proceeds with a second ISI where six new states and nine new transitions are

covered; thus, the state coverage has increase to 62 covered states, and the transition has increased to 63 covered transitions. The table indicates that the all state and all transition coverage has not achieved yet, meaning that no coverage objective has been met for all state and all transition coverage; therefore, test case construction continued with a third ISI: seven new states are covered and 15 new transitions are covered, that makes 69 states are covered in total, and 78 transitions are covered in total. Since the table indicates that our coverage objective has not been achieved yet, meaning all state and all transition coverage has not been achieved, test case construction therefore continued with a fourth ISI. The table indicates that no new state and transition is covered, so that the all state and transition coverage has not achieved yet, and since we need to observe coverage progress over past the observation window (recall the stopping criterion) which is set to two, the test case construction carried with a fifth ISI: no new state and transition has been covered. Since the observation window is set to two, and the last two transactions do not cover any new state and has been covered, i.e., the coverage objective has not been met, test case construction stopped.

We studied the remaining uncovered states and transitions. The main reason for this is that there were no emergency situations to stop the elevator through the simulation, so that the states and transitions that simulate the elevator's emergency brake have never been exercised.

Table 4 indicates that four states and eight transitions are not covered in our Elevator Control System. In the next section, we will use the SPIN model checker to cover the uncovered states and transitions.

Table 4.— Result for Elevator Control System test cases

ISI #	Transaction duration (nano-sec.)	Number of				Status
		New states	States covered	New transitions	Transitions covered	
1	1950	56	56	54	54	PASS
2	2150	6	62	9	63	PASS
3	1800	7	69	15	78	PASS
4	2000	0	69	0	78	PASS
5	1800	0	69	0	78	PASS

5.2.4 Result: SPIN model checker test case construction

Our online test framework was successfully able to cover 69 (94.5%) states and 78 (90.7%) transitions for the Elevator Control System case study as it has been shown in the previous section, but missed four states and eight transitions. As a result, we used the SPIN model checker to generate counterexamples for trap properties to cover these uncovered state/transition.

As we have discussed in section 5.2.2, when `Elevator_Engine` receives a moving signal from `Controlling_cab` to order the `Elevator_Engine` to start moving, the status sensor which is modeled/simulated as a Boolean variable is updated in an activity (in the `Elevator_Engine` capsule): the value of the variable becomes true when a signal is received to request movement; it becomes false when a signal to stop is received. Once this is done, `Controlling_cab` checks the status sensor value to confirm whether the `Elevator_Engine` has updated the status sensor value and the elevator started moving or not. As we have discussed in section 5.2.1, one of the elevator requirements is to use the emergence brake if an elevator is commanded to move but does not move. Therefore, when `Controlling_cab` checks the status sensor value and it appears that the value is false (i.e., the `Elevator_Engine` has not started moving), then the elevator emergency brake is used. However, since we always update the status sensor value to a true value

when the `Elevator_Engine` receives a moving signal, an emergency situation has never occurred during the online testing. This is a design decision we made since we did not have access to a real hardware (elevator). We created the model in a model-in-the-loop procedure. When introducing hardware (hardware-in-the-loop), the Boolean variable would be replaced by some form of communication with the actual hardware. In case the hardware malfunctions and the engine does not start when it is asked to start, then the Boolean variable would become false, and this would trigger the emergency behaviour. Similarly, the status sensor value is updated to false when a stop signal is received by the `Elevator_Engine`. If the `Elevator_Engine` is not stopped, then the emergency brake is used. However, since the `Elevator_Engine` always updates the status sensor value when a stop signal is received, and this never happens, the emergency brake also has never occurred during the online testing.

During the model verification using the SPIN model checker, the global variables of the model automatically become part of the verifier's internal state vector during verification. Since the contents of any C code fragment is not interpreted by SPIN when it generates the verifier, SPIN cannot know about the presence of the declaration for any local variable, and therefore the local variables remain invisible to the verifier: local variables appears outside the state vector. A local variable can be manipulated as a global variable, but the values assigned to it are not considered to be part of the global system state that the verifier tracks. As we have discussed above, whenever a stop/move signal is received by the `Elevator_Engine`, SPIN evaluates (i.e., simulates the execution of) the activity code where the status sensor variable is updated, the status sensor variable is considered a local variable that remains invisible to the verifier and its declaration appears outside the

state vector. Therefore, the status sensor value is not considered to be part of the global state that the verifier track, i.e., SPIN does not try to evaluate the impact of changes to this variable value. As a result the emergency paths in the model are never exercised during the verification process.

The following states have not been covered through our online test framework:

1. RTEdgeModel::Controlling_Cab_AC::waitForEmergAck
2. RTEdgeModel::Controlling_Cab_AC::S1
3. RTEdgeModel::EmergencyBreak_AC::S1
4. RTEdgeModel::EmergencyBreak_AC::S2

The following transitions have not been covered through our online test framework, the transition is defined as from state, to state, transition name:

1. RTEdgeMod-
   el::Controlling_Cab_AC::checkStatus,RTEdgeModel::Controlling_C
   ab_AC::waitForEmergAck,emerg_stop
2. RTEdgeMod-
   el::Controlling_Cab_AC::waitForEmergAck,RTEdgeModel::Controlli
   ng_Cab_AC::WaitForRequest,ack_emerg
3. RTEdgeMod-
   el::Controlling_Cab_AC::S1,RTEdgeModel::Controlling_Cab_AC::Wa
   itForRequest,T2
4. RTEdgeMod-
   el::Controlling_Cab_AC::checkEngineStatus,RTEdgeModel::Control
   ling_Cab_AC::S1,T10

5. RTEdgeMod-
   el::EmergencyBreak_AC::Init_state,RTEdgeModel::EmergencyBreak_
   AC::S1,RequestEmergBreak
6. RTEdgeMod-
   el::EmergencyBreak_AC::S1,RTEdgeModel::EmergencyBreak_AC::Init
   _state,ack
7. RTEdgeMod-
   el::EmergencyBreak_AC::Init_state,RTEdgeModel::EmergencyBreak_
   AC::S2,T1
8. RTEdgeMod-
   el::EmergencyBreak_AC::S2,RTEdgeModel::EmergencyBreak_AC::Init
   _state,T2

Appendix A shows the full result of using the SPIN model checker to generate counter-examples for the uncovered state. (We will discuss uncovered transitions next.) In this section, we will show the result for one state and we have chosen the `Controlling_Cab_AC` capsule to select one of its uncovered states, specifically `waitForEmergAck`, to be covered using the SPIN model checker. The `Controlling_Cab_AC` capsule's unique id is 7 and the id of state `waitForEmergAck` in capsule `Controlling_Cab_AC` is 19. Thus, if the current state of the `Controlling_Cab_AC` capsule is `waitForEmergAck`, then the `acinfo` array (recall from section 4.3.4 that this array, which is defined as an array of structures where each structure element, that is called "state", holds the current state of each capsule in the model) is such that `acinfo[7].state = 19`. The predicate for the state variable is automatically defined by our framework (section 4.3.4) in Promela as (assuming capsule identifiers and state identifi-

ers are stored in enumerations and we have `CONTROLLING_CAB_CONTROLLING_CAB_AC=7` and `Controlling_Cab_AC_waitForEmergAck=19`):

```
#define predicate0 /*noCheck*/
 (acinfo[CONTROLLING_CAB_CONTROLLING_CAB_AC].state==
 Controlling_Cab_AC_waitForEmergAck)
```

After that the Trap property is automatically defined as:

```
![](predicate0->[]!tgpredfloorrequest2)
```

where `tgpredfloorrequest2` is the variable in the Promela code that identifies the model has run to completion. This trap property specifies that if `predicate0`, which holds the uncovered state definition "`Controlling_Cab_AC_waitForEmergAck`", is always reached, this implies that the model will not alwaysrun to its completion. .

Fig. 18 shows the counterexample returned by SPIN for this state's trap property. The red line in Fig. 18, line 715, shows the last `acinfo.state` appears before the SPIN claims that the LTL property is violated. The current state that is shown in the counterexample is `acinfo[7].state == 19`, which specifies that state `waitForEmergAck` of the `Controlling_Cab_AC` capsule is reachable. This confirms that the `waitForEmergAck` state is visited through the model execution. Thus, the counterexample shows the state is reached through the execution of the model.

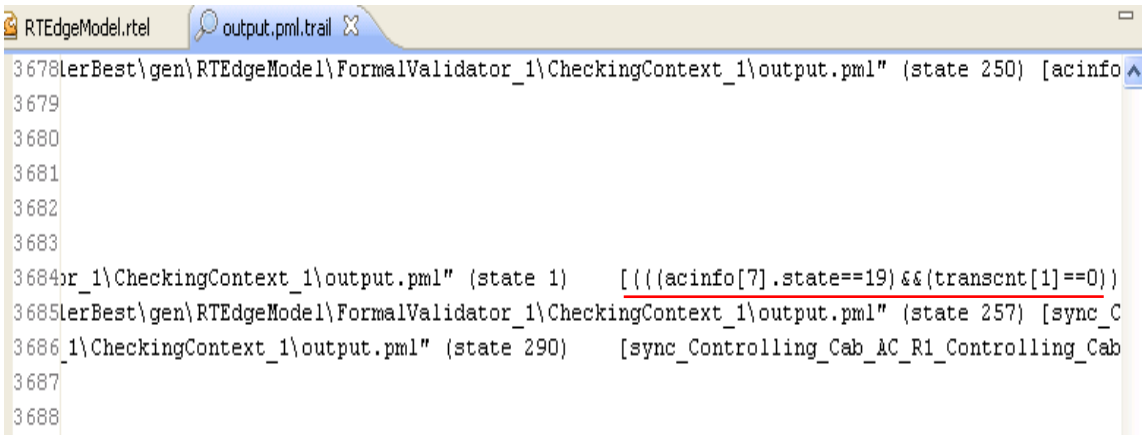


Fig. 18– Counterexample of the state's trap property

There are eight transitions that have not been covered using our online test framework. Appendix A. 2 shows the full result of using the SPIN model checker to generate counterexamples for these uncovered transitions. Next we will show how we used the SPIN model checker to cover one of these uncovered transitions. We have chosen the uncovered transition T2 in the `EmergencyBreak_AC` capsule: this transition is a representative example of other transitions. Since the transition is defined using its from-state and to-state in Promela, we have defined two predicates to hold (recall section 4.2.6). The id of capsule `EmergencyBreak_AC` is 11, the from-state of the transition is `S2`, which has id equal to 3 (i.e., is the current state of the `EmergencyBreak_AC` capsule is `S2`, then the `acinfo` array, as we have discussed in section 4.3.4, is such that `acinfo[11].state = 3`), and the to-state of the transition is `Init_state`, with id 1. As a result the following predicates are defined (again, enumerations are used):

```
#define predicateFrom7 /*noCheck*/
(acinfo[EMERGENCYBREAK_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_S2)
```

```
#define predicateTo7 /*noCheck*/
(acinfo[EMERGENCYBREAK_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_Init_state)
```

After we have defined the predicates, the LTL trap property is defined as:

`![](predicateFrom7->X!(predicateTo7->[]!tgpredfloorrequest2))`

; which specifies that the from-state is reached first, then always the to-state is not reached next, and the model is not run to completion.

The transition's trap property has generated a counterexample as shown in Fig. 19. The first red line, at line 1350, shows the current state at this point of the execution is `S2` of the `EmergencyBreak_AC` capsule, since `acinfo[11].state = 3`, as it has been discussed above. The next current state appears at line 1357 in Fig. 19. This line shows `acinfo[11].state == 1`, which means the to-state `Init_state` of `EmergencyBreak_AC` capsule is visited next. At this point SPIN terminates because the LTL property is violated and the transition is visited because the from-state is visited first and then the to-state is visited and the model run to completion.

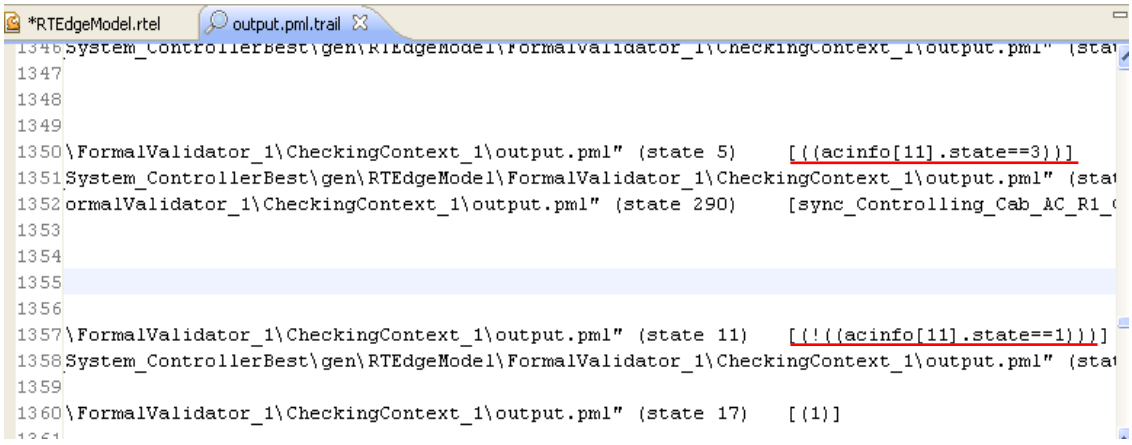


Fig. 19 – Counterexample of the Transition's trap property

Chapter 6 - **Conclusion**

In the domain of embedded, time critical, real-time systems, assurance that the system meets its timing as well as functional requirements is a key to the successful implementation of the real-time embedded solution. The very real challenges of proving correctness of a real time embedded application leads to high cost of software development and maintenance.

To facilitate the verification of such systems, in the context of a Model-Driven Development based on RTEdge (the tool created by our sponsor), we developed a black-box, online, feedback-driven random test case generation procedure. . It is black-box because we only rely on the specification to derive test inputs (and not the implementation of the model), but also because we only use a small portion of the specification: in fact we only use the specification of signals that trigger behaviour in the set of communicating state machines in the RTEdge model because we felt any more detailed analysis of the model would be too difficult (e.g., because of pieces of C/C++ code it contains). It is online since we rely on RTEdge capabilities to simulate the specified application: as a result, test case creation occurs concurrently to the simulation of the test case execution. It is feedback driven because we regularly observe progress in terms of coverage of elements of the model in order to decide whether to stop or continue the construction of a test case.

We experimented with this procedure on a well-known (Production Cell) case study and showed that we can effectively reach adequacy for well-known selection criteria: coverage of each state and transition at least once, coverage of each state and transition at least three times. We conjecture that such results might in part be due to some structural characteristics of our model. In addition, this procedure is experimented on an Elevator

Control System case study which showed that we can obtain adequate results for the selection criteria: coverage of each state and transition at least once, and showed the complementarity with the SPIN model checker. We intend to study this in future work with additional experiments.

In case this conjecture is confirmed, i.e., other case studies show that no adequate test suite is obtained by using our black-box, online, feedback-driven random test case generation procedure, we do not feel overly concerned since thanks to the mapping from an RTEdge model to Promela we can automatically use SPIN, as discussed in this thesis, to give us test cases that will exercise the states/transitions that are missed by our test case generation procedure. Using Promela and SPIN only would not be economical to achieve the same level of coverage. However, combining testing and a formal method would be economical, as advocated by others (e.g., [12]).

In future work we would like to experiment with other models. We would like to study the fault detection effectiveness of our tests. We could also investigate how our test case generation procedure, though exemplified with RTEdge, can be (hopefully easily) adapted to other model-driven development environments.

Bibliography

1. Zhang, J., Chen, X., Wang, X.: Path-oriented test data generation using symbolic execution and constraint solving techniques. Software Engineering and Formal Methods, pp. 242-250 (2004)
2. Lefticaru, R., Ipate, F.: Automatic State-Based Test Generation Using Genetic Algorithms. Symbolic and Numeric Algorithms for Scientific Computing, pp. 188-195 (2007)
3. Binder, R.: Testing Object-Oriented Systems. Addison-Wesley (2000)
4. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
5. Utting, M., Legeard, B.: Practical Model-based testing. Morgan Kaufmann
6. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A Temporal Logic Based Theory of Test Coverage and Generation. In: Katoen, J.-P., Stevens, P. (eds.) TACAS, vol. 2280, pp. 327-341. Springer Berlin Heidelberg (2002)
7. Hessel, A., Larsen, K.G., Nielsen, B., Pettersson, P., Skou, A.: Time-optimal Real-Time Test Case Generation using UPPAAL. In: Formal Approaches to Testing of Software, pp. 114-130. Springer, (Year)
8. Jéron, T., Morel, P.: Test Generation Derived from Model-Checking. In: Computer Aided Verification, pp. 108-122. Springer, (Year)
9. Higashino, T., Nakata, A., Taniguchi, K., Cavalli, A.R.: Generating Test Cases for a Timed I/O Automaton Model. In: IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS), pp. 197-214. Springer, (Year)
10. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley (2003)
11. Amir, P.: The temporal logic of programs. Foundations of Computer Science,18th Annual Symposium on, pp. 46- 57 (1977)
12. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Software Testing, Verification & Reliability 19, 215-261 (2009)
13. Gargantini, A., Heitmeyer, C.: Using Model Checking to Generate Tests from Requirements Specifications. European Software Engineering Conferenc, vol. 1687, pp. 146–162. Springer (1999)
14. Feiler, P., Gluch, D., Hudak, J.: The Architecture Analysis & Design Language (AADL): An Introduction. Software Engineering, Carnegie Mellon University CMU/SEI-2006-TN-011, (2006)
15. http://www.omg.org/spec/UML/2.0/, 2013
16. Holzmann, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engineering 23, 279-295 (1997)
17. Fraser, G., Arcuri, A.: Whole Test Suite Generation. IEEE TSE 39, 276-291 (2013)
18. Mathur, A.P.: Foundations of Software Testing (2008)
19. Luo, G., Bochmann, G.V., Petrenko, A.: Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. Software Engineering, IEEE Transactions 20, 149-162 (1994)
20. Li, J.J., Wong, W.E.: Automatic test generation from communicating extended finite state machine (CEFSM)-based models. In: Proc. IEEE International Symposium on Real-Time Distributed Computing, pp. 181-185. (Year)

21. Jin, X., Ciardo, G., Kim, T.-H., Zhao, Y.: Symbolic verification and test generation for a network of communicating FSMs. In: Proc. Int. Conf. on Automated technology for verification and analysis, pp. 432-442. Springer, (Year)
22. Conformiq Software Ltd., http://www.verifysoft.com/ttcn-3_qtronic_sip.pdf, 2013
23. Guo, Q., Hierons, R., Harman, M., Derderian, K.: Computing Unique Input/Output Sequences Using Genetic Algorithms. LNCS 2931 Proc. Formal Approaches to Software Testing, pp. 164-177 (2004)
24. Tretmans, J.: Model based testing with labelled transition systems. LNCS 1-38 (2008)
25. Kalaji, A.S., Hierons, R.M., Swift, S.: An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. Information and Software Technology 53, 1297-1318 (2011)
26. Duale, A.Y., Uyar, M.Ü.: A method enabling feasible conformance test sequence generation for EFSM models. Computers, IEEE Transactions 53, 614-627 (2004)
27. Schwarzl, C., Peischl, B.: Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems. Quality Software, 2010 International Conference, pp. 122-131, Zhangjiajie (July 2010)
28. Vain, J., Kull, A., Kääramees, M., Markvardt, M., Raiend, K.: In: Zander, J., Schiferdecker, I., Mosterman, P. (eds.) Model-Based Testing for Embedded Systems, pp. 425-452. CRC (2011)
29. Larsen, K.G., Mikucionis, M., Nielsen, B., Skou, A.: Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In: Proc. ACM international conference on Embedded software, pp. 299-306. (Year)
30. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Transactions on Software Engineering (TSE) SE-10, 438–444 (1984)
31. Arcuri, A., Iqbal, M.Z., Briand, L.: Formal analysis of the effectiveness and predictability of random testing. ACM International Symposium on Software Testing and Analysis, pp. 219-230 (2010)
32. Chen, T.Y., Leung, H., Mak., I.K.: Adaptive random testing. Asian Computing Science Conference, vol. 3321, pp. 320-329. Springer (2005)
33. Arcuri, A., Briand, L.: Adaptive random testing: an illusion of effectiveness? Proc. ACM Int. Symp. on Software Testing and Analysis, pp. 265-275 (2011)
34. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. Proc. ACM Conference on Programming language design and implementation, pp. 213–223 (2005)
35. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball., T.: Feedback-directed random test generation. Proc. ACM/IEEE ICSE, pp. 75–84 (2007)
36. Groce, A.: Coverage rewarded: Test input generation via adaptation-based programming. Proc. ASE, pp. 380-383 (2011)
37. Satpathy, M., Yeolekar, A., Ramesh, S.: Randomized directed testing (redirect) for simulink/stateflow models. Proc. ACM international conference on Embedded software, pp. 217–226, New York, NY (2008)
38. Cleaveland, R., Smolka, S.A., Sims, S.T.: An Instrumentation-Based Approach to Controller Model Validation. Automotive Software Workshop, San Diego

39. Ammann, P.E., Black, P.E.: A specification-based coverage metric to evaluate test sets. In: High-Assurance Systems Engineering Proceedings. 4th IEEE International Symposium on, pp. 239- 248. (Year)
40. Ammann, P.E., Black, P.E., Majurski, W.: Using model checking to generate tests from specifications. In: the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98}, pp. 46-54. IEEE Computer Society, (Year)
41. McMillan, K.L.: Symbolic Model Checking. Springer (1993)
42. Geist, D., Farkas, M., Landver, A., Lichtenstein, Y., Ur, S., Wolfsthal, Y.: Coverage-directed test generation using symbolic techniques. In: Srivas, M., Camilleri, A. (eds.) First International Conference, FMCAD, vol. 1166, pp. 143-158. Springer (1996)
43. Rayadurgam, S., Heimdahl, M.P.E.: Coverage Based Test-Case Generation using Model Checkers. Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, ECBS 2001, pp. 83- 91, Washington, DC, USA (2001)
44. Hong, H.S., Lee, I., Sokolsky, O.: Automatic test generation from statecharts using model checking. In: FATES'01,Workshop on Formal Approaches to Testing of Software, pp. 15-30. (Year)
45. Hamon, G., Hamon, G.E., Moura, L.D., Rushby, J.: Generating efficient test sets with a model checker. 2nd International Conference on Software Engineering and Formal Methods, pp. 261-270. IEEE Press (2004)
46. Lee, D.Y., Yannakakis, M.: Principles and methods of testing finite state machines-a survey. Proceedings of the IEEE 84, 1090-1123 (1996)
47. Haupt, R.L., Haupt, S.E.: Practical Genetic Algorithms. Wiley (1998)
48. Formal Development of Reactive Systems: Case Study Production Cell. In: Lewerentz, C., Lindner, T. (eds.) Lecture Notes in Computer Science. Springer (1995)
49. Burns, A.: How to Verify a Safe Real-Time System The Application of Model Checking and a Timed Automata to the Production Cell Case Study. Real-Time Systems Journal (1998)

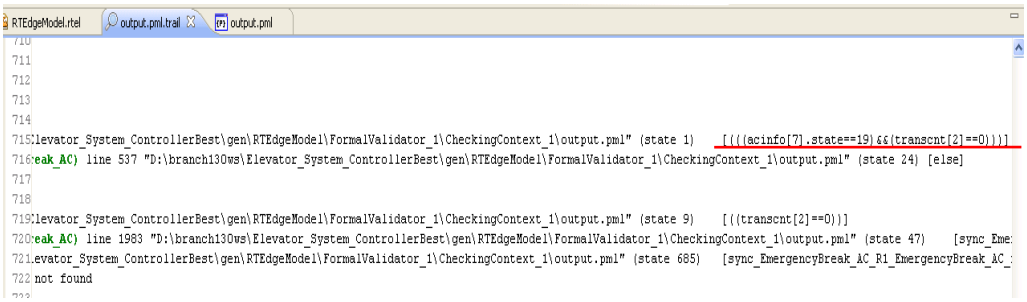
Appendix A Elevator Control System Result

A. 1 Results using SPIN model checker for the uncovered states

1. #define predicate0 /*noCheck*/
(acinfo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state ==
Controlling_Cab_AC_waitForEmergAck)

LTL:

#define Controlling_Cab_AC_waitForEmergAck 19
#define CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC 7



2. #define predicate1 /*noCheck*/
(acinfo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state ==
Controlling_Cab_AC_S1)

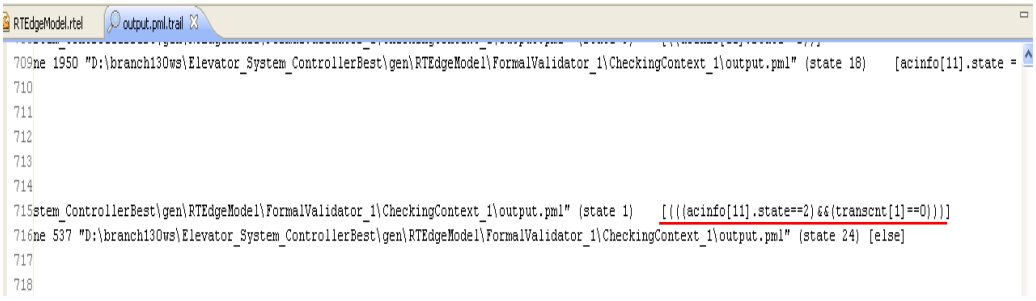
LTL:

#define Controlling_Cab_AC_S1 22
#define CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC 7

3. #define predicate2 /*noCheck*/
(acinfo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emer-
gencyBreak_AC_S1)

LTL:

#define EmergencyBreak_AC_S1 2
#define EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC 11

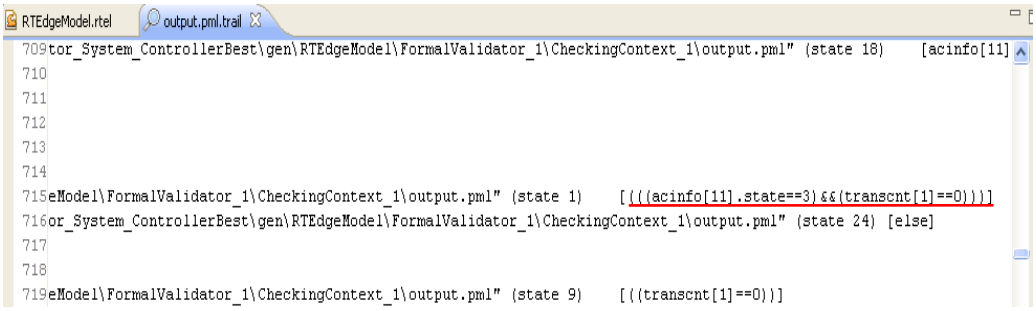


4. #define predicate3 /*noCheck*/ (acinfo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == EmergencyBreak_AC_S2)

   LTL:

   #define EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC 11

   #define EmergencyBreak_AC_S2 3



A. 2 Results using SPIN model checker for the uncovered transitions

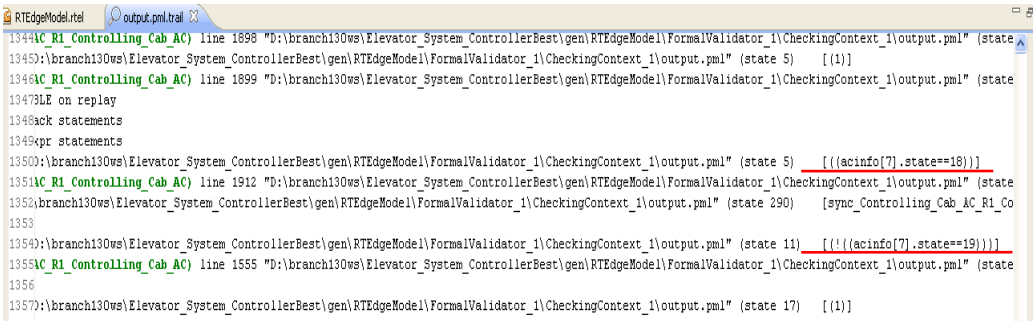
1. #define predicateFrom0 /*noCheck*/ (acinfo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state == Controlling_Cab_AC_checkStatus)

   #define predicateTo0 /*noCheck*/ (acinfo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state == Controlling_Cab_AC_waitForEmergAck)

   LTL:![](predicateFrom0->X!(predicateTo0->[]!tgpredfloorrequest2))

   #define CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC 7

   #define Controlling_Cab_AC_checkStatus 18
   #define Controlling_Cab_AC_waitForEmergAck 19

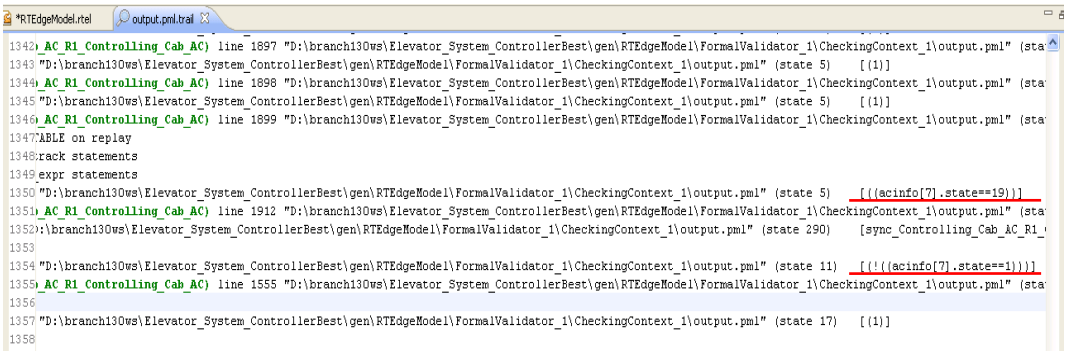


2.
```
#define predicateFrom1 /*noCheck*/ (acin-
fo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state ==
Controlling_Cab_AC_waitForEmergAck)
#define predicateTo1 /*noCheck*/
(acinfo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state ==
Controlling_Cab_AC_WaitForRequest)

LTL:![](predicateFrom1->X!(predicateTo1->[]!tgpredfloorrequest2))

#define CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC 7

#define Controlling_Cab_AC_waitForEmergAck 19
#define Controlling_Cab_AC_WaitForRequest 1
```

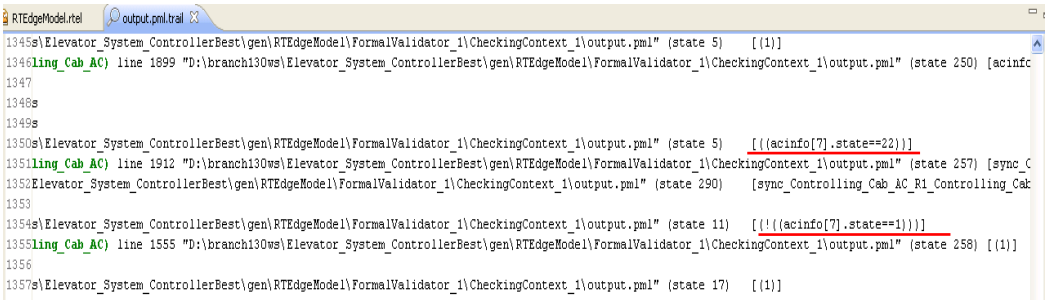


3.
```
#define predicateFrom2 /*noCheck*/ (acin-
fo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state ==
Controlling_Cab_AC_S1)
#define predicateTo2 /*noCheck*/
(acinfo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state ==
Controlling_Cab_AC_WaitForRequest)

LTL:![](predicateFrom2->X!(predicateTo2->[]!tgpredfloorrequest2))

#define CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC 7

#define Controlling_Cab_AC_S1 22
#define Controlling_Cab_AC_WaitForRequest 1
```

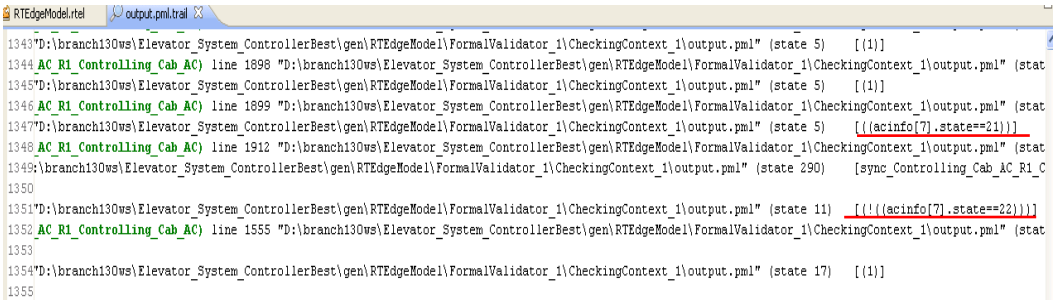


4.
```
#define predicateFrom3 /*noCheck*/ (acin-
fo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state ==
Controlling_Cab_AC_checkEngineStatus)
#define predicateTo3 /*noCheck*/
(acinfo[CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC].state ==
Controlling_Cab_AC_S1)

LTL: ![](predicateFrom3->X!(predicateTo3->[]!tgpredhallrequest1))

#define CONTROLLING_CAB_AC_R1_CONTROLLING_CAB_AC 7

#define Controlling_Cab_AC_checkEngineStatus 21
#define Controlling_Cab_AC_S1 22
```



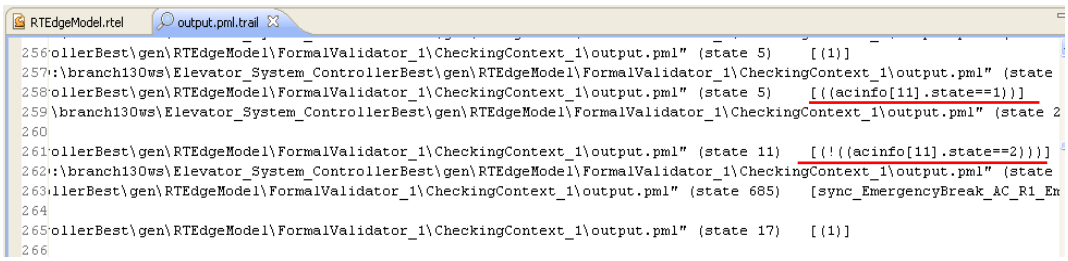
5.
```
#define predicateFrom4 /*noCheck*/ (acin-
fo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_Init_state)

#define predicateTo4 /*noCheck*/
(acinfo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_S1)

LTL: ![](predicateFrom4->X!(predicateTo4->[]!tgpredhallrequest1))

#define EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC 11

#define EmergencyBreak_AC_Init_state 1
#define EmergencyBreak_AC_S1 2
```



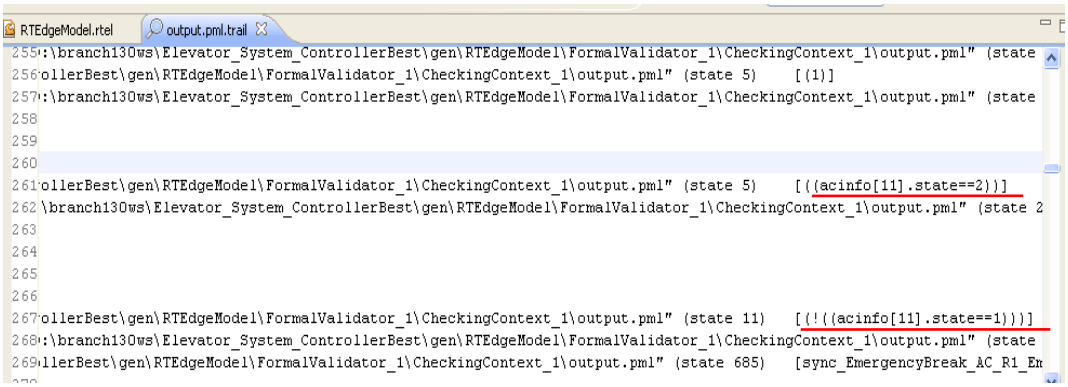
6.
```
#define predicateFrom5 /*noCheck*/ (acin-
fo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_S1)

#define predicateTo5 /*noCheck*/
(acinfo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_Init_state)

LTL: ![](predicateFrom5->X!(predicateTo5->[]!tgpredhallrequest1))

#define EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC 11

#define EmergencyBreak_AC_S1 2
#define EmergencyBreak_AC_Init_state 1
```



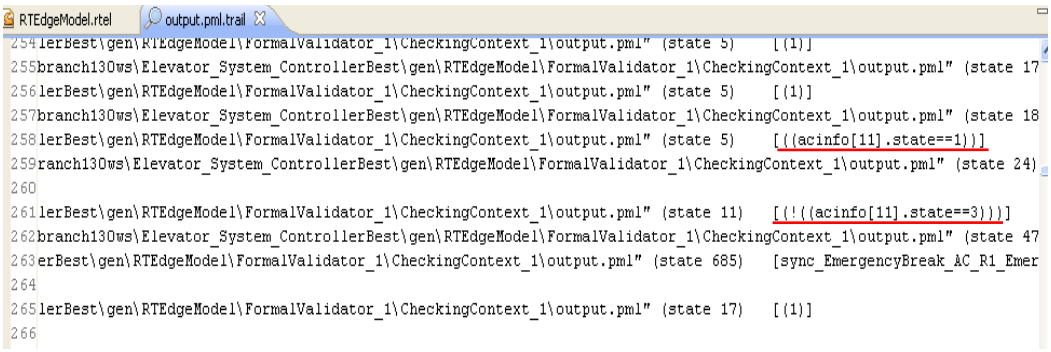
7.
```
#define predicateFrom6 /*noCheck*/ (acin-
fo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_Init_state)

#define predicateTo6 /*noCheck*/
(acinfo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_S2)

LTL:![](predicateFrom6->X!(predicateTo6->[]!tgpredfloorrequest2))

#define EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC 11

#define EmergencyBreak_AC_Init_state 1
#define EmergencyBreak_AC_S2 3
```



8.
```
#define predicateFrom7 /*noCheck*/ (acin-
fo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_S2)

#define predicateTo7 /*noCheck*/
(acinfo[EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC].state == Emergen-
cyBreak_AC_Init_state)

LTL:![](predicateFrom7->X!(predicateTo7->[]!tgpredfloorrequest2))

#define EMERGENCYBREAK_AC_R1_EMERGENCYBREAK_AC 11

#define EmergencyBreak_AC_S2 3
#define EmergencyBreak_AC_Init_state 1
```