Verifying Complex Software Control Systems from Test Objectives: Application to the ETCS System

Rabea Ameur-Boulifa[1][a], Ana Cavalli[2][b] and Stephane Maag[2][c]
[1]*Institut Mines-Télécom/Télécom ParisTech, Université Paris-Saclay, France*
[2] *Institut Mines-Télécom/Télécom SudParis, Université Paris-Saclay, CNRS UMR 5157 SAMOVAR, France*
Rabea.Ameur-Boulifa@telecom-paristech.fr, {ana.cavalli, stephane.maag}@telecom-sudparis.eu

Keywords: Formal Verification, Safety, Model Checking, Software Control Systems.

Abstract: Ensuring the correctness of complex distributed software systems is a challenging task, the issue of building frameworks for developing such safe and correct systems still remains a difficult issue. Where test coverage is dissatisfying, formal analysis grants much higher potential to discover bugs during the development phase. This paper presents a framework for formal verification of complex systems based on standardized test objectives. The framework integrates a transformation of test objectives into formal properties that are verified on the system by model checking. The overall proposed approach for formal verification is evaluated by the application to the standard European Train Control System (ETCS). Some critical safety properties have been proved on the model, ensuring that the model is correct and reliable.

1 INTRODUCTION

The development, verification and testing of complex software systems (e.g., embedded systems, control systems, etc.) involve high challenges due to the complexity of the implementation and the time required for testing and verifying. In most cases, the deadline is not met, products being launched several months late and without achieving the required performance targets. Within that context, many efforts have been done in the processes of software validation and in particular in the testing of these systems from the verification of formal specifications. However, although languages and formal transformation have been proposed for that purpose, very few works have been devoted to the verification of models from the testing phases.

We herein provide a framework based on formal methods that performs the methods and techniques necessary to automate the development and verification processes. The expected result consists in a more reliable system, in terms of functionality, safety and robustness and a reduction of the time necessary for verification. Indeed, we aim at verifying software systems from the standardized definition of their test objectives.

In order to highlight and assess our methodology, we applied it on a real industrial case study, that is the European Train Control System (ETCS) (ERTMS Commission Group - European Commission, 2017). Over the past century, Europe's railways have been developed within national boundaries, resulting in a variety of different signaling and train control systems, which hampers cross-border traffic. In the scope of increasing their interoperability, the European Union has decided to adopt a standard, the ETCS. We formalized some components of this standard and based on standardized test objectives, we verified some critical properties of the system.

In summary, the main contributions of this paper are:

- a transformation that converts IF-test-objectives to MCL-formulas in such a way that automatic and exhaustive verification of safety properties will be possible on application;
- an integrated framework suitable for software developers. The proposed framework relies on a toolchain for automating test generation, thus it complements the toolchain by allowing formal verification;
- an example illustrating the approach and showing its practical use by applying our framework on a realistic case study.

[a] https://orcid.org/0000-0002-2471-8012
[b] https://orcid.org/0000-0003-2586-9071
[c] https://orcid.org/0000-0002-0305-4712

Ameur-Boulifa, R., Cavalli, A. and Maag, S.
Verifying Complex Software Control Systems from Test Objectives: Application to the ETCS System.
DOI: 10.5220/0007918203970406
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 397-406
ISBN: 978-989-758-379-7
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

The paper is organized as follows. Section 2 presents the related works on the verification and testing of complex systems and the relation between testing and verification techniques. In section 3, basic concepts and definitions for the modelling of the system, and the description of test objectives are described. Section 4 gives the language for the properties specification and the rules for the transformation of test objectives into formal properties to be verified on the formal model. In Section 5, the formal specification of the ETCS system is provided as well as the results of the experimentation performed on it. Finally, Section 6 gives the conclusion and perspectives of this work.

2 RELATED WORKS

Verification and testing of complex systems (e.g., train or avionic software and industrial control systems) have been studied for many years. Several models and techniques have been proposed to either test the systems (Asadollah et al., 2015; Garousi et al., 2018) or verify their formal models (Bérard et al., 2013; Karna et al., 2018). In the following, we cite the works from which we get inspired.

Many works have defined languages or approaches to model complex systems. In (Salem et al., 2015), a UML profile named R-UML has been proposed to model and then verify flexible control systems. The model is enriched in order to consider the management of resource sharing. Besides, a transformation model to state machines is used for the verification process. Also, the authors of (Belghiat and Chaoui, 2015) proposed a Pi-calculus-based approach through transformation of sequence diagrams for the verification process. An interesting mapping is formally defined to analyse and verify well-defined properties. Models transformation is of high importance and has been well described in a very recent survey (Kahani et al., 2018). The transformation is also performed in our approach in order to apply model-checking. However, although these studies propose high semantics and transformation rules, the authors do not propose experiments on real case studies and the properties to be checked are not provided. Besides, in our approach, our transformation through models and languages are easily applied in a sense that the procedures are tooled (based on Fiacre (Garavel et al., 2011) - see Section 5.3).

We also get inspired of the researches on testing and verification of complex software control systems. (Kapinski et al., 2016) presents an interesting survey for modeling, testing, and verifying embedded control systems. The authors note that model-based development approaches are crucial in industrial contexts. They also raise the challenging consolidation between testing and verification processes in software systems. In this area, papers like (Petiot et al., 2014) experimented an incremental methodology of deductive verification assisted by test generation. Though the proposal is very promising, it tackles the software code with no formal state machines. Nevertheless, relevant combination between test and verification was introduced. Time constraints in the validation processes are also very important. In (Mubeen et al., 2017), the authors studied them and proposed an approach to represent and refine them among various abstraction levels. Besides, they verified timing constraints on a complex system. In our paper, such information has also been raised and we noted the importance of such constraints in the model as well as into the verification phase.

It was shown in the survey of Fraser et al. (Fraser et al., 2009) that testing could be performed from verification processes. Many efficient techniques and tools have been developed in that purpose. Still in that way, the authors of (Ferrante et al., 2018) have recently presented the use of model-checking for automated test cases generation applied to a standardized complex aerospace system. However, all these works present the use of verification for testing without raising the issue of the test objectives. Although a recent interesting document about bridging the gap between testing and verification approaches has been published by Microsoft Research (Godefroid, 2016), the methods always propose to apply verification to testing. In our work, we present a way for verifying a formal specification from standardized test objectives in the context of Control Train Systems (TCS). Such verification purposes are not new as depicted in the well-known report published by the CMU on TCS (Platzer and Quesel, 2009). In (Ghazel, 2014), a dynamic model is proposed for model-checking of European TCS specifications. UML, LTL and a model-based methodology is successfully applied and the author clearly focus that work as an entry point for generating test cases. In a more recent work (Jesus Valdivia et al., 2017), the authors proposed a novel testing platform based on virtual laboratory. Braking events have been tested using faults injection. However, as mentioned, although these works are very interesting, none of them started from standardized test objectives (eventually automatically generated) to the verification of the models. This is what we tackle in our work we herein propose.

3 PRELIMINARIES

3.1 Behavioural Models

There exist several modelling languages, such as the symbolic state machines (Mouttappa et al., 2013), to specify complex systems when the number of exchanged messages, data and parameters is important. In our approach, we describe each process (the behaviour of each component) in terms of Labelled Transition Systems (LTS for short) which are strongly based on the definition and semantics of Symbolic Transition Graph with Assignment (Hennessy and Lin, 1995). The LTS extends the general notion of Labelled Transition Systems by adding parameters and value-passing features. Transitions are labelled by parameterised actions, to which are attached a set of parameters and variables.

Definition 1. *(LTS) A Labelled Transition System is a rooted directed graph where each state s is associated with a finite set of free variables and each edge is labelled by a triple* $s \xrightarrow{([b],\alpha,\vec{x}:=\vec{e})} s'$. *Where b is a boolean expression, e is a data expression (which may includes variables), x ranges over data variables,* $\vec{x} := \vec{e}$ *is a multiple assignment and* α *ranges over a set of abstract action algebras.*

The set of abstract actions is a set of action algebras that can encode naturally usual point-to-point message passing calculi using $?a(x_1,\ldots,x_n)$ for inputs, $!a(v_1,\ldots,v_n)$ for outputs. Figure 3 shows the graphical representation of the two LTSs depicting the behaviour of the OBU (On-Board Unit) and RBC components (Radio Block Center).
The parallel composition of processes and their communication is defined through parameterised Network (pNet) (Ameur-Boulifa et al., 2017; Henrio et al., 2015). pNets are tree-like networks of processes. They provide means to represent in a structured and hierarchical way the behaviour of processes, represented as labelled transition systems (LTS with value-passing messages). Composition of pNets is realized by synchronisation vectors that relate the actions of (a subset of) the subnets, with a global action that will be exported at the next level.

Figure 2 gives a graphical representation of pNets. The pNet shown in this example is represented by a set of four boxes: OBU, RBC, GETSPEED and RELEASESPEED boxes inside the ETCS_SYSTEM box (hierarchy). Each box representing parameterised process (which can be formed of other pNets or LTS and that have parameters and local variables), is surrounded by labelled ports encoding a particular sort of the corresponding pNet. The ports are interconnected through edges for communication and synchronisation. Edges are translated to synchronisation vectors.

3.2 The IF Language

IF is a language based on the semantic temporized state machines, allowing the description of existing concepts into specification formalisms (Bozga et al., 2002). A real-time system described using IF language is composed of processes running in parallel and interacting asynchronously through shared variables and message exchanges via communication channels. The description of a system in IF consists in the definition of data types, constants, shared variables, communication signals and processes. The signals set is divided into inputs and outputs provided by the environment of a current state machine or sent to its environment, respectively. One of the main advantages of the IF language is the ease of use to formally specify test objectives as described in the next section.

3.3 Test Objectives

In our work, we focus on the functional properties to be tested on a system or implementation (namely implementation or system under test (SUT)). In that context, a test objective describes a particular functionality of a SUT by specifying the property to be checked in the system implementation. It is an observable action of the system that once described in IF language (Bozga et al., 2004) is used for guiding the space exploration of the system's states.

A test objective is described as a conjunction of conditions, including the following optional conditions: instance of a process with an identifier, a state of the system (a source state or a destination state), an action of the system (a message sent, a message received, an internal action), a variable of the process or a clock of the process, specifying a value and its state (active or inactive). Table 1 shows the general structure of a test objective in the IF description.

From a test objective, a set of test cases is generated. Basically, a test case is a sequence of input and output actions. It represents a trace of an LTS that satisfies the test objective.

4 FRAMEWORK - FROM TEST OBJECTIVES TO VERIFIED PROPERTIES

Our goal is to use formal methods both to increase the quality of such systems through enhancing the

Table 1: General structure of Test Objective.

TO	$=$	$TO_1 \wedge TO_2$
TO_1	$=$	$P_1 \wedge P_2 \wedge \cdots \wedge P_5$
P_1	$=$	$process : instance = \{proc\}id$
P_2	$=$	$state : source = s_1$
P_3	$=$	$state : destination = s'_1$
P_4	$=$	$input\ action : \alpha_1(parameters)$
P_5	$=$	$variable : (v_1 = value)*$
TO_2	$=$	$Q_1 \wedge Q_2 \wedge \cdots \wedge Q_5$
Q_1	$=$	$process : instance = \{proc\}id$
Q_2	$=$	$state : source = s_2$
Q_3	$=$	$state : destination = s'_2$
Q_4	$=$	$output\ action : \alpha_2(parameters)$
Q_5	$=$	$variable : (v_2 = value)*$

verification activity, and to prevent unnecessary tests. However, since we aim at a general approach for specifying properties, we advocate to use the test objectives to generate formal specifications or properties that may be used for proving or disproving the correctness of the systems. As described before (Section 3.3) test objectives express the desired or unexpected behaviour of a system in terms of input and output actions. Their distinctive features are typically the dealing with data parameters that are generally abstracted away in formal models because verification problems are undecidable for infinite systems. Furthermore, the crucial characteristic of our models is the parameterised action.

Precisely, MCL (Model Checking Language) is a language for expressing properties that addresses this crucial matter: representing and handling data, and reasoning about their value.

4.1 Property Language

Basic MCL logic extends action in modal μ-calculus with data variables (Mateescu and Thivolle, 2008), so it suits for describing the property of concurrent systems. Indeed, MCL language provides high-level operators facilitating the construction of formulas. It allows to handle in a natural way the data values present in the LTSs and to reason about systems described in value-passing process algebras such as LOTOS.

The MCL formulas are logical formulas built over regular expressions using boolean operators, modalities operators (the necessity operator denoted by [ ] and the possibility operator denoted by $\langle\ \rangle$), maximal fixed point operator (denoted by μ) and data-handling constructs inspired from functional programming languages. From LTS's point of view, a transition sequence starting at the initial state and satisfying a regular formula ε can be expressed in MCL either as an example for the $\langle\varepsilon\rangle$`true` formula, or as a counterexample for the $[\varepsilon]$`false` formula. For specifying transition sequences, MCL uses regular formulas.

A regular formula is a logical formula built from action formulas, traditional and extended regular expression operators, namely concatenation (.), choice (|), and transitive- reflexive closure (*).

An action formula is a logical formula built from action predicates which includes action patterns, and the "**tau**" constant operator. Action pattern can either action for matching values denoted by $\{\alpha\,!e_1 \ldots !e_n\}$, or action for extracting and storing values denoted by $\{\alpha\,?x_1 : T_1 \ldots ?x_n : T_n\}$ where α is an action name, e_i is an expression, variable name or function name, x_i is a variable name and T_i is a basic data type. It is important to note that the usage of ! and ? symbols in MCL specification has different meaning from the notation introduced in LTS models. They enable to match a given value against an expression or to extract and store it in a variable. MCL also uses other specific notations: the "**true**" constant is used to match a value of any action formula and the wildcard clause "**any**" matches a value of any type.

4.2 Encoding Test Objectives into MCL Formula

We define the transformation of IF test objectives into MCL specification by associating to each test objective a MCL formula expressing a liveness property. Consider TO a test objective:

$TO = ''process : instance = \{proc\}id''$
$\wedge ''state : source = s'' \wedge ''input\ action : \alpha_1(x_1,..,x_n)''$
$\wedge ''output\ action : \alpha_2(x'_1,..,x'_n) \wedge ''state : destination = s''''$

The encoding of test objective TO is the following MCL property pattern:

$$[\texttt{true}*.\{\alpha_1\,?x_1 : T_1 \ldots ?x_n : T_n\}]\ \texttt{inev}(\{\alpha_2\,\dagger x'_1 \ldots \dagger x'_n\})$$

where $\{\alpha_1\,?x_1 : T_1 \ldots ?x_n : T_n\}$ and $\{\alpha_2\,\dagger x'_1 \ldots \dagger x'_n\}$ are the input action and the output action resp. Note that for dataless actions, brackets can be omitted. And such that $\dagger x$ can be either $!x$ or $?x : T$ depending on whether x'_i consists of matching values with data x_i (encoded $!x_i$) or extracting and storing them in typed variables (encoded $?x_i : T_i$). MCL uses the usual datatypes e.g., *bool, nat, string.*

The predicate inevitability of an action α denoted inev(α) expresses that a transition labelled with α is eventually reached from the current state. It can be defined in MCL using fixed point operator by the following macro definition:

```
macro inev(α)=
 μ X.(<true> true and [not (α)] X)
```

`macro_end`

meaning that as long as there has been no α action, there is always an execution leading to α.

It is important to note that our translation does not care about the conditions on the states which are involved in test objectives. For the formal verification, the pointing of the states of a system is useless information. Because the approach checks properties by means of an exhaustive search of all possible states that the system could reach.

Concerning the variables that we have not considered in the given general test objective. They are also translated but not in completely automatic manner. The translation requires sometimes the intervention of the user. As mentioned earlier, such as a programming language, MCL offers constructors to facilitate the handling of data values. Thus, for encoding the variable conditions that are not empty, we use such constructors, in particular the **where** constructor. The action pattern ending with the optional clause "**where b**" means that the pattern matches an action if and only if the guard (boolean expression) **b** is true. The guard can be the equality check, i.e, like **where** $v_i = val_i$.

5 EXPERIMENTS

5.1 ETCS System

Our work has been experimented on a formal model of the European Train Control System (ETCS). The ETCS is a part of the European standard that defines the European Railway Traffic Management System[1].

The normative documents describe ETCS as a train control standard, based on in-cab equipment, an On-Board Unit (OBU) able to supervise train movements and to stop it according to the permitted speed at each line section, along with calculation and supervision of the maximum train speed at all times (ERTMS Commission Group - European Commission, 2017). The information is received from the ETCS equipment beside the track. For that purpose, the OBU runs concurrently with a Radio Block Center (RBC). Basically, this standard is proposed in order to improve the safety in European railways. The trains running limits are stated by movement authorities.

The train control system ensures the reception of messages like safety distance, speed limitations and controls the driving according to these limitations. Secondly, the safety is increased by the supervision of train driving. As illustrated in the Figure 1, data are used by the on-board ETCS equipment to supervise the train drivers[2]. Therefore, the on-board equipment has to know both information regarding the route as well as information regarding the train.

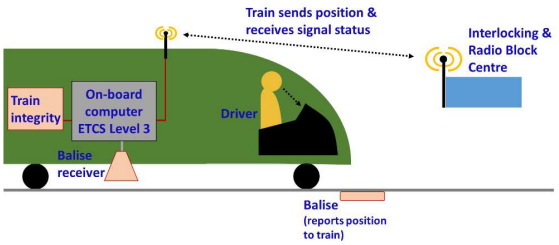


Figure 1: The ETCS system.

This train data is introduced by the train driver before starting the journey. Based on the track data and on the data entered by the driver, the on-board subsystem calculates a dynamic speed profile, calculates a set of braking curves for train movement supervision and commands the brake application, if necessary[3]. A high performance is given by an increasing speed and capacity due to a track-train transmission system and the on-board equipment knowledge. The track-train transmission system uses precise information about running limits and consequently, supervises a train permanently to avoid that the speed limits exceed. The on-board equipment knowledge about train running limits is used to inform drivers through displays, allowing the railways to increase the running speeds without worrying about shortening the time period for track side signal observation.

5.2 ETCS Formal Specification

The Figure 2 shows the semantic model of our use case, the signature of interfaces, and the behaviour of its components and its methods. The overall architecture of the system consists of:

- The OBU component receiving the information about the current speed, the current location and maximum authorized speed, and makes a decision to the issue to brake or not,
- The RBC component that manages the exchange of data required for a safe train travel,
- A behavioural specification of methods GETSPEED and GETRELEASESPEED that returns respectively at any time the current speed of the train and the maximum authorized speed under which the train must to respect.

[1] http://www.ertms.net/

[2] https://medium.com/@POST_UK/moving-block-signalling-b9b0b9f498c2

[3] https://ec.europa.eu/transport/modes/rail/ertms_en

In this section we describe only the structure of the pNets, the communication among them and the behaviour of the local methods. The local methods compute regularly the new value of the speed. They are naturally encoded as an infinite loop of actions that returns a random value in the range $[0 \ldots max_value]$ such that *max_value* is the maximum value of speed attainable by each of them. The internal behaviour of OBU and RBC components are represented by the LTSs given in Figure 3.
The ETCS system as specified in the European standard uses a clock. Each component owns a clock which is synchronized with clocks of the other components. Particularly, the RBC component receives a location of a danger point for a certain period of time. In our models, we abstract away this kind of detail and represent the receiving of a location of danger point as an infinite loop.

The OBU Component. The OBU receives the current estimated location (?ELocation(l)) of the train from the environment. This location is encapsulated in a request sending to the RBC (!MARequest(id,l)). Thus, it receives in return a release speed consisting of a limited speed under which the train must to respect (?MA(id,rs)). The local method GETSPEED will fill the current speed with the calculated value (?ESpeed(s)). If the current speed s of the train is less than or equals to the limited speed, it can continue to operate. Otherwise, if the current speed s exceeds the maximum authorized speed, an emergency brake is applied (!EBcmd(1)). The brake is hold until the train totally stops. The OBU sends a Driver Machine Interface command to display speed information to the driver (!DMIspeed(rs)).

The RBC Component. The RBC receives the Danger locations that are sent from the environment (?DLocation(l)). Based on this information and the estimated location report received from the OBU, the RBC computes release speeds by calling the method RELEASESPEED (!call_computation(rs)). The release speed is then sent to OBU via the (!MA(id,rs)) message.

Test Objectives for Automatic Verification. A critical property of the ETCS that is particularly crucial to be verified is the ability of the system of taking over control if the driver appears to be going too fast. Among scenarios describing this property we consider the following:

Example 1. *Scenario in which the train is in the indication state (encoded s_5), running at 120km/h while the release speed is 80km/h, thus* OBU *has to generate a brake command and to pass to the intervention state (encoded s_8) – by traversing the normal state (encoded s_3)–. The associated test objective is formulated in the IF description (where {OBU}0 is used to identify the first instance of process* OBU*) as follows:*

$$TO = "process:instance=\{OBU\}0" \wedge "variable\ rs = 80" \wedge "state:source = s_5" \wedge "variable\ s = 120" \wedge "state:source = s_8" \wedge "output\ action:\texttt{EBCMD1}"$$

As detailed in previous works (Nguyen et al., 2014), TestGen-IF tool can be used to generate efficient test cases for testing and validating of a system from such test objectives. However, as the system deal with data variables over a domain the authors show that the number of test cases generated by exhaustive strategy grows too large even for relatively small domains.

5.3 Results

The source specification was written in the intermediate format Fiacre language (Berthomieu et al., 2012). The Fiacre language provides syntax for data types and expressions, definition of LTSs, and a form of composition of processes by synchronization on channels. Then we run a combination of CADP tools (Garavel et al., 2011), the most important ones are ceasar.open for generating transition systems from Fiacre programs, ocis the interactive simulator, and Evaluator4, the model-checker that deals with the MCL logic. All the tools provided by the CADP toolbox are command-line tools, but also integrated into graphical user interface (GUI). Through the Xeuca interface (see Figure 4) CADP toolbox allows an easy access to the offered functionalities.

From a finite model pNet of ETCS we have computed the LTS of the global system. Choosing small values for the domain of parameters, i.e. [0..4] intervals for all data, we obtain an LTS with 662 states and 3615 transitions.

To formally verify the correct execution of the different scenarios, we generated several properties in MCL in precise and generic way; they express various facets of the system. Some properties express global correctness of the application, seen from the (external) ETCS point of view, and that reveals the feasibility of several scenarios or the impossibility of some errors.

First, we started by verifying usual properties the system is deadlock-free. As well, we verified a property expressing that each scenario is acyclic, i.e. specifying the absence of unfair execution actions, which is characterized using the infinite looping operator (denoted by @ operator):

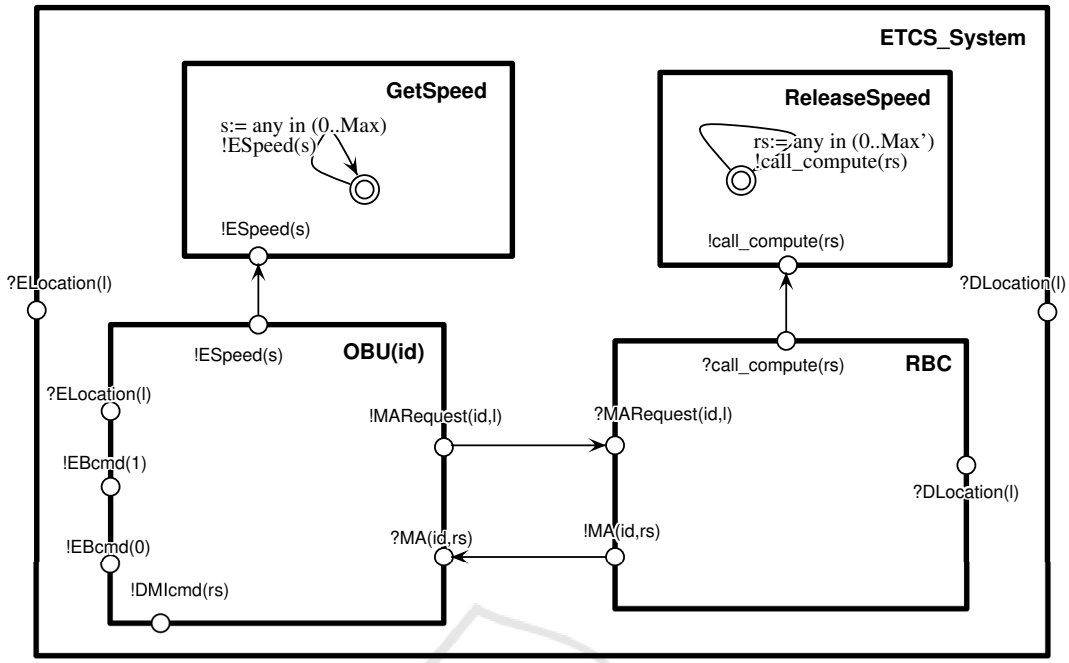
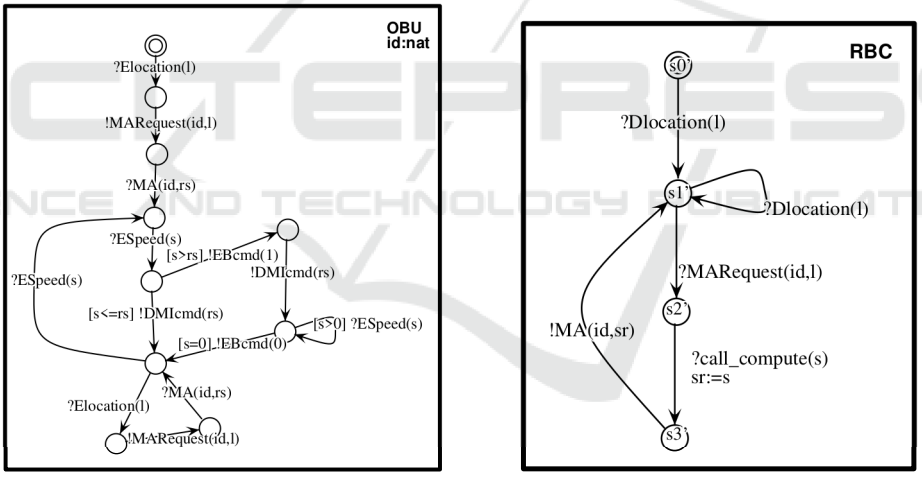


Figure 2: pNet model for the ETCS system.



(a) LTS of OBU

(b) LTS of RBC

Figure 3: Model for ETCS System.

[true∗] <true> ⓒ

Not surprisingly, this property does not hold for all scenarios. By using ocis simulator (as depicted in Figure 4), we visualize this is because the RBC processor can stay in its state after receiving a ?DLocation(l). Indeed, since we abstracted away the clock, the RBC process can be engaged in a circular receiving loop of danger locations that are sent from the environment. However, under the hypothesis that this action is performed over a time period, such a cycle is executed at most a finite number of times. Thus, cycles of this form should not be considered a problem, and the model is refined, for instance by allowing only a finite number of actions.

Next, we proved a formula that checks the reachability of the emergency brake command:

[true∗."EBCMD1"] true

This property is evaluated to TRUE meaning that the break command is reachable over all computations paths.

Afterwards, we proved properties that we generated from test objectives. For instance, consider the following test objective checking global correction of

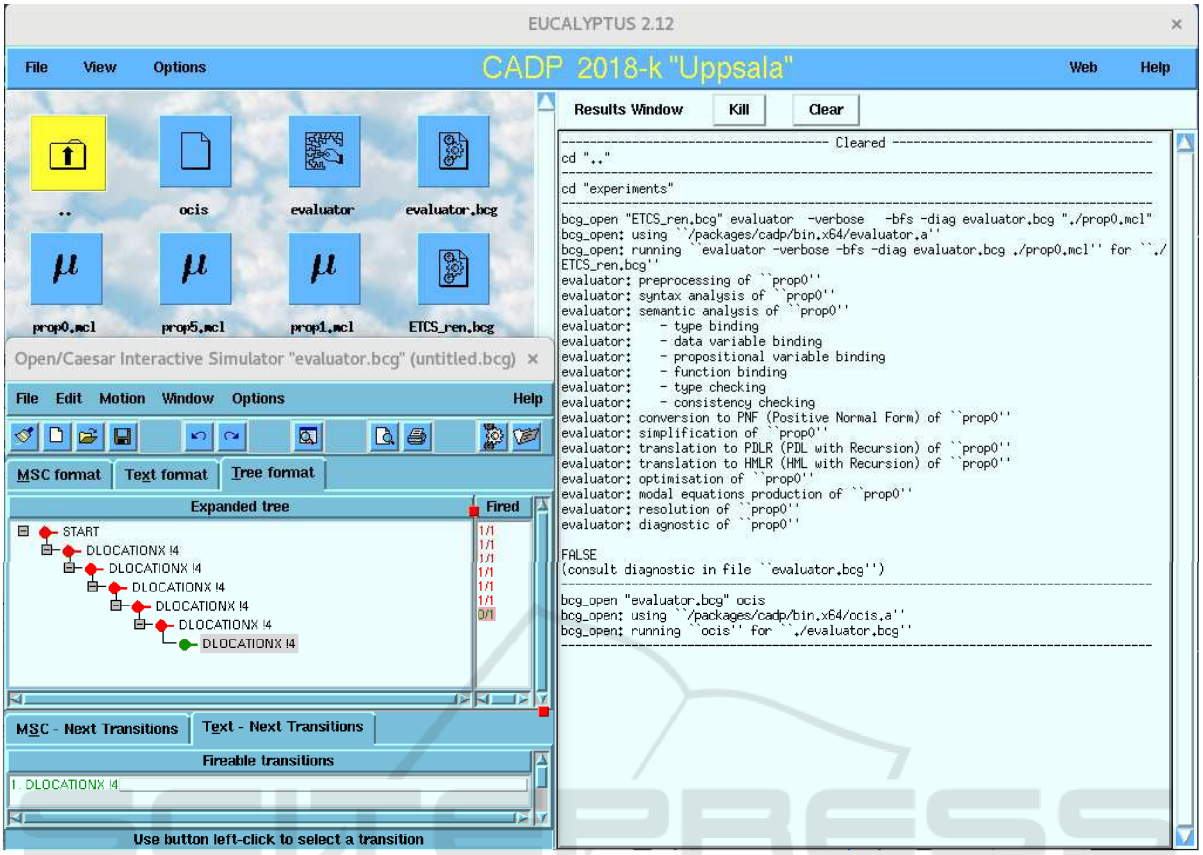


Figure 4: Graphical user-interface of CADP.

the ETCS system:

$$TO = "process:instance=\{RBC\}0" \wedge "state:source=s'_0" \wedge "input\ action:\texttt{MAREQUEST}(id)" \wedge "output\ action:\texttt{MA}(id,sr)" \wedge "state:destination=s'_1"$$

It formalises the scenario in which the train is in the initial state (encoded s'_0) issues a MAREQUEST, thus the RBC has to send MA and to pass to the idle state (encoded s'_1).

Based on the property pattern, our framework derives from the test objective the following MCL formula:

[**true**∗.({MAREQUEST ?id:**nat**})] inev({MA !id ?**any**})

Note how the identifier *id* of the OBU is extracted from a transition label by the first action predicate {MAREQUEST ?id:nat} (by wildcard symbol ?) and is used subsequently in the property. This property is evaluated to TRUE meaning that for each possible request of a train (*id* being the identifier of an OBU), the return of the corresponding movement authority permission is reachable with some returned value of speed (denoted **any**).

From the test objective given in the Example 1, our framework generates also a MCL formula although in a less systematic way:

[**true**∗.{MA ?**any** ?v_1:**nat**}.(**not**{ESPEED ?**any**})∗.{ESPEED ?v_2:**nat where** $v_2 > v_1$}] inev(”EBCMD 1”)

This formula expresses a general property for verifying that OBU issues a brake command at each state whenever speed is greater than releaseSpeed. Actually, the test objective does not specify the input action but the value of the variables s and rs ($s = 120$ and $rs = 80$), and the output action: EBCMD 1. It is specified that the speed of the train is greater than the release speed. By analysis of the models of the train, we look for the actions that set the value of these variables, we use them to express the assignment instead of the variables assignment. As it can be noted, the variable s is set by the action ESPEED sets and the variable rs is set by the action MA. Thereby, for these actions that are used in the formula, it is explicitly expressed that the argument of the first is greater than the argument of the second. Note that the values 80 and 120 of the variables are not set in the formula to express this safety property in a general form. It is evaluated to TRUE: once an RBC delivers a release speed, upon the first speed exceeding this limit the train always issues the brake command.

6 CONCLUSION AND PERSPECTIVES

This article provides a framework for the generation of logical properties from test objectives with the aim of verifying properties on complex distributed systems. A test objective provides a convenient description for generating test cases to be executed to achieve a particular software testing requirement. This paper describes the translation of such test objectives to MCL properties. MCL properties enable the exhaustive verification of applications; the correctness of applications can be proved by using the model checking technique.

Concerning future work, first we plan to refine the behavioural model by taking time into consideration. Preliminary modifications allow to encode implicitly the notion of time and go beyond the issues raised by its abstraction. However, from the test objective point of view, it would be interesting to study what could be the property pattern corresponding to the test objective involving the clock. Moreover, we will consider the eventual changes of the model parameters due to the clock phases over the time.

Finally, our framework could be extended to take into account other aspects in order to offer the ability to analyse non-functional properties.

REFERENCES

Ameur-Boulifa, R., Henrio, L., Kulankhina, O., Madelaine, E., and Savu, A. (2017). Behavioural semantics for asynchronous components. *J. Log. Algebr. Meth. Program.*, 89:1–40.

Asadollah, S. A., Inam, R., and Hansson, H. (2015). A survey on testing for cyber physical system. In *IFIP International Conference on Testing Software and Systems*, pages 194–207. Springer.

Belghiat, A. and Chaoui, A. (2015). A Pi-calculus-based approach for the verification of UML2 sequence diagrams. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 2, pages 1–8. IEEE.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., and Schnoebelen, P. (2013). *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media.

Berthomieu, B., Bodeveix, J., Filali, M., Garavel, H., Lang, F., Peres, F., Saad, R., Stoecker, J., and Vernadat, F. (2012). The syntax and semantics of FIACRE. In *Deliverable number F.3.2.11 of project TOPCASED*.

Bozga, M., Graf, S., and Mounier, L. (2002). If-2.0: A validation environment for component-based real-time systems. In *International Conference on Computer Aided Verification*, pages 343–348. Springer.

Bozga, M., Graf, S., Ober, I., Ober, I., and Sifakis, J. (2004). *The IF Toolset*, pages 237–267. Springer Berlin Heidelberg.

ERTMS Commission Group - European Commission (2017). Delivering an effective and interoperable european rail traffic management system (ERTMS) – the way ahead. Technical report, SWD(2017) 375.

Ferrante, O., Scholte, E., Rollini, S., North, R., Manica, L., and Senni, V. (2018). A methodology for formal requirements validation and automatic test generation and application to aerospace systems. Technical report, SAE Technical Paper.

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

Garavel, H., Lang, F., Mateescu, R., and Serve, W. (2011). CADP 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS'11*, volume 6605 of *LNCS*, Saarbrücken, Germany. Springer, Heidelberg.

Garousi, V., Felderer, M., Karapıçak, Ç. M., and Yılmaz, U. (2018). Testing embedded software: A survey of the literature. *Information and Software Technology*.

Ghazel, M. (2014). Formalizing a subset of ertms/etcs specifications for verification purposes. *Transportation research part C: emerging technologies*, 42:60–75.

Godefroid, P. (2016). Between testing and verification: Dynamic software model checking.

Hennessy, M. and Lin, H. (1995). Symbolic bisimulations. *Theor. Comput. Sci.*, 138(2):353–389.

Henrio, L., Madelaine, E., and Min, Z. (2015). pNets: An expressive model for parameterised networks of processes. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 492–496. IEEE.

Jesus Valdivia, L., Solas, G., Añorga, J., Arrizabalaga, S., Adin, I., and Mendizabal, J. (2017). Etcs on-board unit safety testing: Saboteurs, testing strategy and results. *Promet-Traffic&Transportation*, 29(2):213–223.

Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingel, J., and Varró, D. (2018). Survey and classification of model transformation tools. *Software & Systems Modeling*, pages 1–37.

Kapinski, J., Deshmukh, J. V., Jin, X., Ito, H., and Butts, K. (2016). Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. *IEEE Control Systems Magazine*, 36(6):45–64.

Karna, A. K., Chen, Y., Yu, H., Zhong, H., and Zhao, J. (2018). The role of model checking in software engineering. *Frontiers of Computer Science*, 12:642–668.

Mateescu, R. and Thivolle, D. (2008). A model checking language for concurrent value-passing systems. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, pages 148–164.

Mouttappa, P., Maag, S., and Cavalli, A. (2013). Using passive testing based on symbolic execution and slicing

techniques: Application to the validation of communication protocols. *Computer Networks*, 57(15):2992–3008.

Mubeen, S., Nolte, T., Sjödin, M., Lundbäck, J., and Lundbäck, K.-L. (2017). Supporting timing analysis of vehicular embedded systems through the refinement of timing constraints. *Software & Systems Modeling*, pages 1–31.

Nguyen, H. N., Zaïdi, F., and Cavalli, A. R. (2014). A framework for distributed testing of timed composite systems. In *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, pages 47–54.

Petiot, G., Kosmatov, N., Giorgetti, A., and Julliand, J. (2014). How test generation helps software specification and deductive verification in frama-c. In *International Conference on Tests and Proofs*, pages 204–211. Springer.

Platzer, A. and Quesel, J.-D. (2009). European train control system: A case study in formal verification. In *International Conference on Formal Engineering Methods*, pages 246–265. Springer.

Salem, M. O. B., Mosbahi, O., Khalgui, M., and Frey, G. (2015). R-UML: An UML profile for verification of flexible control systems. In *ICSOFT*, pages 118–136. Springer.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS