Towards a hybrid formal analysis technique for safety-critical software architectures

Ammar Boucherit*

Computer Science Department,
University of El-Oued, Algeria
Email: ammar-boucherit@univ-eloued.dz
*Corresponding author

Laura M. Castro

MADS Research Group,
A Coruña University, Spain
Email: lcastro@udc.es

Osman Hasan

SEECS,
National University of Sciences and Technology (NUST), Pakistan
Email: osman.hasan@seecs.nust.edu.pk

Abdallah Khababa

Computer Science Department,
Ferhat Abbas University,
Setif, Algeria
Email: akhababa@univ-setif.dz

Abstract: Given the catastrophic damage that bugs in critical systems can inflict on human life and its socio-economic environment, the use of rigorous analysis techniques while developing such systems is getting more and more important especially with the increasingly growing complexity of their architecture. However, the aforementioned growing complexity of such systems architecture leads to many scalability issues for the existing formal specification and verification approaches. This paper presents a novel and scalable formal development approach for critical system software architectures. In particular, our proposal is based on rewriting logic and combines both model checking and property-based testing techniques to bridge the gap between these complementary techniques, and hence overcome the drawbacks of previous attempts to ensure the absence of undesired or unexpected behaviour in the specification and implementation of a critical system.

Keywords:

Copyright 20XX Inderscience Enterprises Ltd.

Reference to this paper should be made as follows: Boucherit, A., Castro, L.M., Hasan, O. and Khababa, A. (xxxx) 'Towards a hybrid formal analysis technique for safety-critical software architectures', *Int. J. Critical Computer-Based Systems*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes:

This paper is a revised and expanded version of a paper entitled [title] presented at [name, location and date of conference].

1 Introduction

Critical systems are increasingly being used in almost every aspect of our daily lives in fields such as avionics, power plants, transportation, electronic voting and customer accounting systems. Such systems can be broadly classified according to the aspects of criticality; such as safety critical, mission critical and business critical systems (Hinchey and Coyle, 2010; Puntambekar, 2008). Therefore, depending on the severity and criticality of each system, failure consequences may lead to loss of money, information, and trust to more serious examples that lead to significant environmental disasters, and even loss of human lives. Thus, formal methods-based analyses have been integrated in the critical systems design process to reduce costs and ensure their safety (Becker, 2018; Nyberg et al., 2018; Kamburjan et al., 2018; Ter Beek et al., 2018). However, the increasing complexity of modern architectures makes the verification of safety-critical systems a very challenging task, and an active research area.

Software architecture plays a central role to bridge the gap between system requirements and implementation. The architecture description of a system is a major concern during the development process and it refers to the set of standards, techniques and notations that are commonly used to better represent and analyse software architectures. Various formalisms have been used to formalise such descriptions and we think that they can be classified into three main categories: Petri nets (Ding, 2016; Zhang et al., 2017; Zhu et al., 2018), unified modelling language (UML) diagrams (Thapaliya et al., 2017; Li et al., 2017a; Ozkaya and Kose, 2018) and architecture description languages (ADLs) (Li et al., 2017b; Batista et al., 2018; Liu et al., 2019a). Various mixed approaches have also been proposed in which UML has been combined with Z (Singh et al., 2016b) and/or transformed into Petri nets (Cortellessa et al., 2018; Noulamo et al., 2018; Kocı and Janoušek, 2017; Custódio Soares, 2017) to better define the architecture elements and associate a formal semantics to the software architecture. Similarly, formal method have been widely used to specify a software architecture and to prove rigorously that it satisfies the desired structural and behavioural properties (Taoufik et al., 2017; Giammarco and Giles, 2018; Gerhart et al., 2012; Zhang et al., 2010).

The main objective of this work is to propose a new generic approach that goes beyond the architecture description stage and covers all the development stages: specification, verification and testing for the formal analysis of safety critical systems software architectures. In particular, the proposed approach combines both model checking and property-based testing (PBT) techniques. Such combination of techniques can be motivated by the fact that even if the system model is fully verified with the model-checking technique, we can never be 100% sure that the corresponding software system will also be fault-free and/or fault-tolerant. Therefore, the development of critical systems involves a combination of model checking and PBT approaches (Aoki et al., 2017; Bucchiarone et al., 2004). While the objective of the verification process is to ensure completeness and correctness of a model specification, testing process aims to check the correctness of system implementation properties. In fact, this proposition is an extension of our previous work (Boucherit et al., 2018a), and then, we use rewriting logic as a unifying semantic framework in the specification stage to further enlarge the set of supported formalisms in the modelling stage. In addition, Maude's model checker is used to verify the model properties, and finally PBT technique to raise the confidence level on a given, specific system implementation.

The remainder of this paper is structured as follows: Section 2 presents an introduction to rewriting logic and software architecture basic concepts, model checking, and property-based testing. Then, Section 3 presents the proposed approach for the analysis of critical systems software architectures, which is illustrated on a simple ticket machine (TM) on Section 4. Section 5 discusses related work in the context of our contribution, and finally, Section 6 concludes the paper while identifying some future work.

2 Scientific background

This section provides a brief introduction about the main concepts behind the proposed approach. These foundations include software architecture basic concepts, rewriting

logic (Meseguer, 2012; Ölveczky, 2018), model checking, and PBT (Paraskevopoulou et al., 2015; Hebert, 2019).

2.1 Rewriting logic and architecture description

Maude (Clavel et al., 2007) is an algebraic programming language and system based on rewriting logic (Meseguer et al., 1992), i.e., a formalism that has been shown to be able for the specification of concurrent systems. Within this scope, a concurrent system can be specified by a rewrite theory T, where the dynamic parts of the system are described by a set of labelled – optionally conditional – rewrite rules (L, R) and its set of states (structural part) are described by the signature (Σ, E). Maude supports both equational and rewriting logic computation, and provides a very powerful interpreter that is able to execute Maude specifications that are organised by modules. Maude has been extended to deal with some aspects that have not been considered in the former version. Therefore, real-time Maude (Ölveczky and Meseguer, 2001) is built on top of the Maude to support the formal specification and analysis of real-time and hybrid systems. Practically, Maude and rewriting logic have been widely used in many actual applications and systems analysis (Rosu, 2015; Liu et al., 2019b; Berger et al., 2018; Meseguer, 2018; Xie et al., 2018).

Due to the influence of architectural decisions on almost all phases of system development, a large number of ADLs have been introduced to help designers in the development challenges of safety-critical systems (Feiler et al., 2006; Cuenot et al., 2007; Mens et al., 2010; Sari and Reuss, 2016; Oquendo, 2016; Kang, 2019). Indeed, ADLs provide both a conceptual framework and a concrete syntax for characterising software architectures, and allow a better understanding of the architecture through early error detection, and quality of service analysis.

Within this research field, rewriting logic has taken its place among the different formalisms used to specify software architecture, and many attempts have been made to develop a rewriting-logic-based ADL, such as CBabel (Braga and Sztajnberg, 2004). Similarly, ADLs have been translated into rewriting logic to benefit from its formal semantics (Bae et al., 2012; Ölveczky et al., 2010). Moreover, Table 1 summarises the proposition of Jerad and Barkaoui (2005) for the correspondence between software architecture concepts and real-time Maude constructs.

Table 1 Concepts correspondence between software architecture and real-time Maude

Software architecture concept	*Real-time Maude concept*
Component	Class
Component interface	Set of terms with the sort `service` on top
Component computation	Set of rewrite rules
Connector	Set of rewrite rules
Types	Sorts
Communication events	Message exchange
Configuration	Term with the sort `system` on top
Compositionality	Sub-class relationship

2.2 Model checking

Model checking (Clarke et al., 2018) is a formal verification technique that can automatically determine whether a model M of finite state adheres to a specification Φ. This principle can be expressed briefly as follows: $M \models \varphi$. In fact, M represents a *reduction* of the real system model (which is typically too complex, and present a non-finite number of possible runtime states) that is used in model checking for automatically determining whether it adheres to its specifications $\varphi_1, \varphi_2, \ldots, \varphi_n$, which are defined formally as propositions in temporal logic (Clarke and Emerson, 1982; Pnueli, 1981). The process of verification is done automatically through a model checker that explores (partially or exhaustively) the reachable state space of M and then returns a positive or negative result about the correctness of the property φ. In the case of a negative result, the model checker presents a counterexample showing how the predicate is false.

In this context, Maude features a very powerful linear temporal logic (LTL) model checker. This LTL model checker was designed with the goal of combining a very expressive and general-purpose system specification language with 'on-the-fly' model checking, which is one of the most commonly used approaches for model checking.

2.3 Property-based testing

Even if the model of a system has been formally verified and that its specification has been automatically generated, there is no guarantees that the realised system will not exhibit any inconsistent behaviour. This is because of the possibility of errors in the implementation phase. Therefore, 100% certification requires test sets checking the conformance of the implementation with the specification of the system. Thus, we propose to use PBT as a complementary technique to eliminate errors that may appear in the implementation phase. Unlike the more classical manually implemented, case-by-base specified testing, PBT requires the developer to write universally quantified expressions that characterise the behaviour of the system under test (SUT) functionality, i.e., instead of choosing specific input data, data generators are used to define the input data types, and the expected outcome is specified on the basis of a trusted model (i.e., a less efficient alternative implementation), an oracle, or by exploring the outcome's features.

Thereafter, a PBT testing tool can use data generators (either provided by a data library or customised by the developer) to randomly produce acceptable input data to indeed test those universally quantified expressions, running as many concrete scenarios as desired, and diagnosing each of those scenarios offering counterexamples for test failures (i.e., violations of the universal quantification).

3 Software architectures analysis approach

The software design process is often viewed as a sequence of phases that transforms a set of informal specifications into a detailed technical specification that can be used for the development. All the intermediate phases are characterised by a transformation from a more abstract description to a more detailed one.

Our approach for the development of safety-critical systems contemplates the following phases of software development: modelling, specification, validation, and testing. We start from an abstract description of the system using the appropriate formalism and advance in sequence from one phase to the next. Note that our intent is not to propose a methodological development lifecycle, but a systematic approach to systems analysis. As such, our proposal could potentially fit any desired project management cycle.

The following subsections describe the steps of our approach, that covers the software development process from the abstract specification to the executable implementation.

3.1 Modelling phase

The goal of this phase is to create the first model of the critical system architecture. Depending on the studied system, a system may have multiple operational phases, typically: start-up, initialisation, normal processing, special operations and shutdown.

Generally, the system model is composed of two kinds of descriptions: static and dynamic. The static description aims to identify the different elements that collectively form the main architecture of the system and what is the high-level goal of their cooperation. The static description is completed with a dynamic description that concerns principally the system elements configuration, that is to say, the definition of the rules governing the system behaviour in terms of the possible actions.

Once the relevant architectural elements (components, connectors, and data) are identified, the software architecture model for each important property in the system is defined by a configuration constrained in their relationships. Then, according to the property or aspect to be verified, system architecture must be formalised using a convenient formalism (i.e., ADLs, Petri nets, UML, etc.). This step can be repeated several times until all the interesting actions in the system are well represented. At the end of this phase, a model of high quality for each property of interest is created.

3.2 Specification phase

During this second phase, the software architecture descriptions – the corresponding model for each important system property – obtained in the previous phase are subjected to rewriting logic-based specification. Such a specification may be prepared directly by the designer or by using mapping rules from the used formalism to rewriting logic. Details about formalisms having well-defined mapping rules into rewriting logic can be found in Ölveczky et al. (2010), Bae et al. (2012), Stehr et al. (2001), Mokhati et al. (2006), Jerad and Barkaoui (2005) and Boucherit et al. (2018b).

We propose to use Maude for this purpose. The main reason for this is the ease of specification in Maude, thanks to the flexibility and generality of its underlying rewriting logic-based formalism that supports the specification of a wide range of concurrency models. In addition, rewriting logic via the real-time Maude (Ölveczky and Meseguer, 2001) permits the description of software architecture components and system models directly by means of a single system module specifying a real-time rewrite theory and allows a choice between discrete and continuous time domains.

Finally, Maude supports modular programming, module importation and parameterised programming techniques (Goguen et al., 2000) and, thus, permits the

specification of large and complex systems. These important characteristics give the designer the ability to easily represent hierarchically-composed architectures, facilitating the construction of new specifications from already existing ones. Therefore, Maude specifications become reusable and easier to be understood, debugged and managed (Clavel et al., 2011).

3.3 Verification phase

Given our focus on critical systems design, the integration of a verification phase is essential to our approach, in order to ensure the desired safety properties of the system. This third phase can help the designer to assess that the given system/components are built correctly, and the choice of architecture fulfills the traced requirements.

In our approach, the verification is performed by using the Maude model checking tool. This tool is typically based on two specification levels:

1. *System specification level*, in which the critical system behaviour is formalised using a rewrite theory (covered by the previous phase of our approach).
2. *Property specification level*, where two categories of properties have to be specified (covered in this phase of our approach). The first category is composed of the set of desired system properties to be verified on the rewrite theory. The second category is composed of the properties exhibited by the model (i.e., by design), to be used as a reference in the process of verification by the model checking tool.

The result of the verification process may lead to a positive outcome, meaning that the model of the system satisfies the desired requirements, or to a counterexample, thus reporting any found inconsistencies. Here, a counterexample is an explanation, at the model level that demonstrates how a non-allowed state is reached.

In addition, our approach allows combining the Maude's model checker with the Maude's reachability tool (if necessary) to detect the existence of critical states in the system. The set of these critical states is generally identified by the experts in the field of study. The main advantage of complementing the use of the Maude model-checking tool with the reachability tool is to confirm the existence/absence of some critical states starting from pre-defined initial states. In fact, this possibility minimises the time and state space to be explored and allows us to better understand and analyse the system.

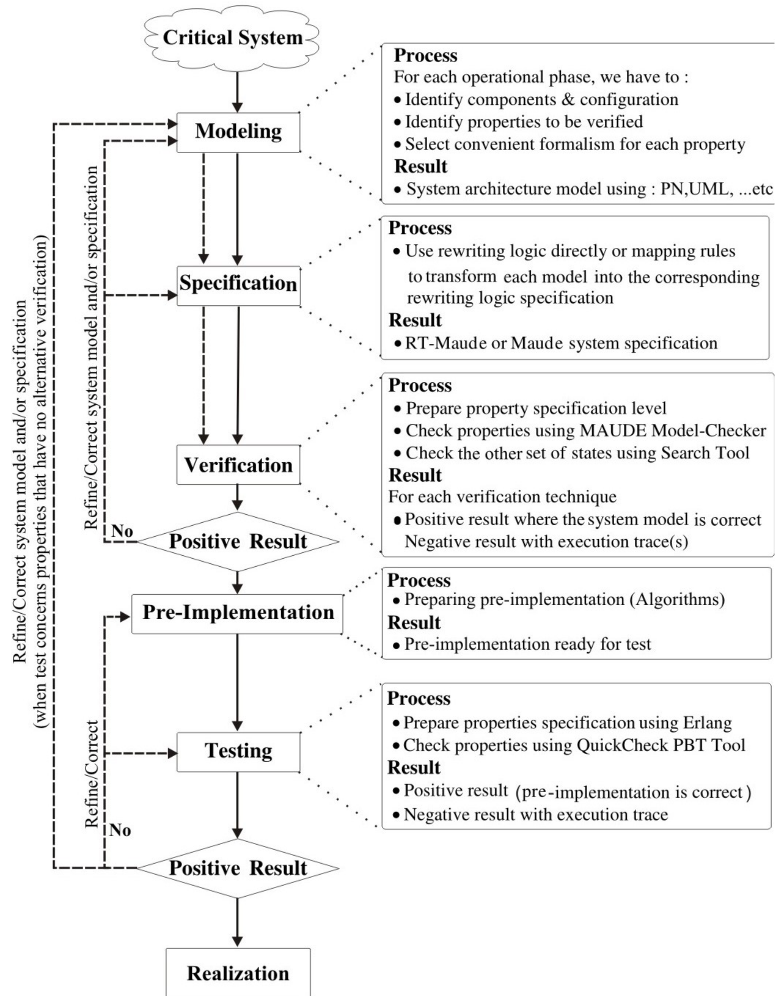
Finally, in the case of infinite-state systems, these techniques are consequently only semi-decidable, but they can still uncover many errors. Maude supports such model checking analyses with a library of search and model checking strategies, including model checking for a useful class of timed temporal logic formulas. Furthermore, due to the reflective nature of its logic and its implementation, such a library can be easily extended by the user with new model checking strategies (Ölveczky, 2007; Rubio et al., 2018).

3.4 Testing phase

The main interest of this phase is to ensure the correctness and quality of the proposed implementation (pre-implementation) before system realisation. On the other hand, the

more complex a software system is, the more difficult (or even unfeasible) it is to verify it formally, since there may be properties which simply cannot be represented using LTL, or that would require an unacceptable effort to define. To cater for such cases, we advocate to apply the testing phase to check those properties for which we have no other alternative left in the verification phase. Contrary to the formal verification and model checking, testing offers no absolute guarantees, though. This is why we propose to use PBT as a strategy as a last resort in our testing phase.

Figure 1 General description of our proposal



Following this approach, we define all desired properties as universally quantified expressions, and devote as much time as available to automatically generate, run, and evaluate them as test cases. Here we will take advantage of QuickCheck's capabilities to produce shunk (i.e., simplified) counterexamples when a specific test case fails, to show how that the realisation of the system violates a given property. In this case, a counterexample is an execution trace that leads to the falsification of the said property.

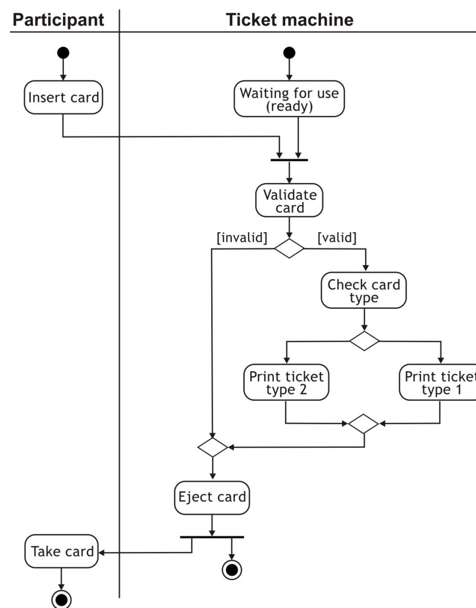
At the end of this section, Figure 1 displays an overview of all the stages of our approach.

4 Case study

4.1 General system description

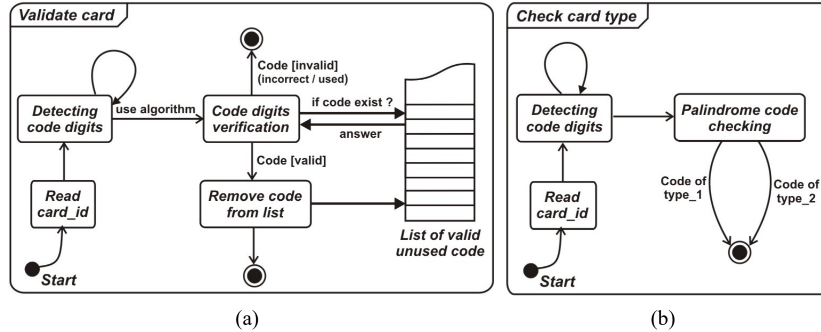
In this section, a simple TM – that delivers various items to users once a special identity card is pressed into the machine – is presented to show the usefulness of the proposed approach. In fact, TM is a type of automated machine (vending machine), which has been often treated as a critical system (Carvalho and Meira, 2019; Schneidermeier et al., 2013; Poppleton, 2007). The proposed TM is to be used by participants in two training workshops that are held on the same date and location. Since the workshops are different in scope and price, there is a recurrent need for two types of tickets: for meals, scientific trips, toolkits and these two types of tickets can be delivered by the TM. The machine is programmed to deliver one ticket at each time of use (for instance, morning, coffee break, noon and at the end of day). The participant's ticket type is determined on the basis of their identity card that is provided during the registration procedure. For instance, in the morning, the TM dispenses toolkit tickets (to take the material bag for a workshop). After some time, the TM delivers tickets for the coffee break. At noon, meal tickets are delivered. At the end of the day, the participants take tickets identifying their bus for their scientific trips.

Figure 2 The UML activity diagram of our case study TM



In our case, TM has to be programmed to avoid money loss, avoid any malfunctions in conference management as well as the direction of the participants.

Figure 3 The UML state diagrams of TM's composite states



(a) (b)

Figure 2 shows a simple activity diagram describing the functionality of the considered TM. First, a participant needs to enter their identity card. In case of failure to identify the person, the participant's card is rejected. If the card is validated, the TM automatically determines the participant type and then prints the corresponding ticket. States `Validate card` and `Check cardtype` are both composite states detailed Figure 3.

In a real scenario, a simple algorithm based on hamming code to detect code error and/or a white-list could be used for such states.

4.2 System and properties specifications

4.2.1 System specification level

The rewrite theory specifying the activity diagram of Figure 2 is given as follows:

```
fmod TMACHINE is
 protecting BOOL .
 sorts Ticket Participant Machine Marking .
 subsorts Participant Machine Ticket < Marking .
 sorts InternalState Action .
 ops ready validating checking printing rejecting : -> Action .
 ops free busy : -> InternalState .
 sorts Attribute AttributeSet .
 subsort Attribute < AttributeSet .
 op none : -> AttributeSet  [ctor] .
 op _,_ : AttributeSet AttributeSet -> AttributeSet [ctor assoc comm id: none] .
 sorts TTicket TCode PName .
 ops T1 T2 : -> TTicket .
 ops npal pal : -> TCode .
 ops P1 P2 P3 P4 : -> PName .
 op none : -> Marking .
 op ticket :_ : TTicket -> Attribute [ctor gather(&)] .
 op [_] : Attribute -> Ticket [ctor] .
 op TM :_ : InternalState -> Attribute [ctor gather(&)] .
 op TMState :_ : Action -> Attribute [ctor gather(&)] .
 op <_> : AttributeSet -> Machine [ctor] .
 op Code :_ : TCode -> Attribute [ctor gather(&)] .
 op Inserted :_ : Bool -> Attribute [ctor gather(&)] .
 op Served :_ : Bool -> Attribute [ctor gather(&)] .
 op <_|_> : PName AttributeSet -> Participant [ctor] .
```

```
 op __ : Marking Marking -> Marking [ctor assoc comm id: none].
endfm

mod TMACHINE-DIAGRAM is
inc TMACHINE .
 op initial : -> Marking .
 op init-machine : -> Marking .
 op init-participant : -> Marking .
 eq init-machine = < TM : free,TMState : ready > .
 eq init-participant = < P1 | Code : npal,Inserted : false,Served : false >
    < P2 | Code : pal,Inserted : false,Served : false > < P3 | Code : npal,
    Inserted : false,Served : false > < P4 | Code : pal,Inserted : false,Served
    : false > .
 eq initial = init-machine init-participant .

 var P : PName .
 var C : TCode .
 var T : TTicket .

 rl[add-new-P] : < P | Code : C,Inserted : false,Served : true > => < P | Code :
               C,Inserted : false,Served : false > .
 rl[P-use-TM] : < TM : free,TMState : ready > < P | Code : C,Inserted : false,
              Served : false > => < P | Code : C,Inserted : true,Served : false >
              < TM : busy,TMState : validating > .
 rl[TM-validating] : < TM : busy,TMState : validating > < P | Code : C,Inserted :
              true,Served : false > => if ((C == pal) or (C == npal)) then (< TM :
              busy,TMState : checking > < P | Code : C,Inserted : true,Served :
              false >) else ( < TM : busy,TMState : rejecting > < P | Code : C,
              Inserted : false,Served : false > ) fi .
 rl[reject-TM-ready] : < TM : busy,TMState : rejecting > => < TM : free,TMState :
              ready > .
 rl[TM-printing] : < TM : busy,TMState : checking > < P | Code : C,Inserted : true,
              Served : false > => < TM : busy,TMState : printing > < P | Code : C,
              Inserted : true,Served : true > if (C == pal) then [ ticket : T1 ]
              else [ ticket : T2 ] fi .
 rl[P-taking-Ticket] : < TM : busy,TMState : printing > < P | Code : C,Inserted :
              true,Served : true > [ ticket : T ] => < P | Code : C,Inserted :
              false,Served : true > < TM : busy,TMState : rejecting > .
 endm
```

4.2.2 Properties specification level

In order to pass to the verification stage, we have to prepare two additional modules. The first module TMACHINE_PREDS must contain the set of properties that are assumed to be verified by Maude LTL model-checker. Such properties may also be used in the second module TM_CHECK to express the set of relevant properties to be checked in the system.

The corresponding modules for our UML activity diagram are given as follows:

```
mod TMACHINE_PREDS is
protecting TMACHINE-DIAGRAM .
including SATISFACTION .
subsort Marking < State .
ops Machine-busy Machine-free : -> Prop .
ops New-Participant(_) Participant-served(_) : PName -> Prop .
var A : Marking . var x : Action .
var P : PName .   var C : TCode .
*** ---------- SOME IMPORTANT PROPERTIES OF THE STUDIED SYSTEM -----------
eq < TM : busy,TMState : x > A |= Machine-busy = true .
```

```
eq < TM : free,TMState : x > A |= Machine-free = true .
eq < P | Code : C,Inserted : true,Served : false > A |= New-Participant(P) = true .
eq < P | Code : C,Inserted : true,Served : true > A |= Participant-served(P) = true .
endm
mod TM-CHECK is
 inc TMACHINE_PREDS .
 inc MODEL-CHECKER .
 inc LTL-SIMPLIFIER .
 op initial : -> Marking .
 op init-machine : -> Marking .
 op init-participant : -> Marking .
 eq init-machine = < TM : free,TMState : ready > .
 eq init-participant = < P1 | Code : npal,Inserted : false,Served : false > < P2 |
  Code : pal,Inserted : false,Served : false > < P3 | Code : npal,Inserted : false,
  Served : false > < P4 | Code : pal,Inserted : false,Served : false > .
 eq initial = init-machine init-participant .
 ops no-deadlock : -> Prop .
 op serving-participant(_) : PName -> Prop .
 var P : PName .
 eq serving-participant(P) = ([](New-Participant(P) -> <>(Participant-served(P)))) .
 eq no-deadlock = [] ((Machine-free -> <> (Machine-busy)) /\ (Machine-busy -> <>
  (Machine-free))) .
endm
```

4.2.3 Verification

The following two properties are expressed in the module TM_CHECK and are verified using the Maude LTL model checker.

1. *System activity and deadlock absence:* This property ensures that the system is always functional and never reaches a deadlock state. This can be done by checking that whenever the TM is busy (resp., free) it is eventually followed by a free (resp., busy) state to ensure that the TM is always active. This property is expressed in LTL as follows:

```
no-deadlock = [] ((Machine-free -> <> (Machine-busy)) /\ (Machine-busy -> <>
(Machine-free)))
```

After verification, we have obtained the following positive results as follows:

```
Maude> reduce in TM-CHECK : modelCheck(initial, no-deadlock) .
      rewrites: 855 in 42175075451ms cpu (15ms real) (~ rewrites/second)
      result Bool: true
```

2. *Serving participants:* This property ensures that whenever a participant is present with a valid card and their card is inserted, they will be surely served, either by a ticket of type 1 or type 2. This property is expressed as follows:

```
serving-participant(P) = ([](New-Participant(P) -> <>(Participant-served(P))))
```

The checking of such property has confirmed their correctness in the proposed activity diagram as follows:

```
Maude> reduce in TM-CHECK : modelCheck(initial, serving-participant(P)) .
      rewrites: 666 in 42051663328ms cpu (0ms real) (~ rewrites/second)
      result Bool: true
```

4.2.4 Testing

Although that we have used a well-known formal verification technique in the previous phase, there are still other important properties that are related to the implementation and could not be verified.

1 *Card validation algorithm*

The first composite state `Validate card` given in Figure 3(a) is composed of two steps. In the first step, we assume that a participant card contains a bar code with 14 digits where the addition of digits in Positions 1, 2, 4 and 8 (starting from right to left) equals to 20. Secondly, the card codes of the registered participants are inserted in a list into the memory of the machine for each service and when a valid participant card is used, it is automatically deleted from this list and it will not be valid again for such time of use.

For the sake of simplicity, we will show here a sample implementation of the first step, although our PBT model below does include both steps. The proposed C function for checking the validity of the card is given as follows:

```
int Checking_Validity(long Code)
{ int digit, t, pos = 0,count = 0, ad = 0;
    while(Code != 0)
    {
        t = Code; Code /= 10;
        digit = t-Code*10;
        pos = ++count ;
        if(pos==1 || pos==2 || pos==4 || pos==8) ad += digit;
    }
   if (ad == 20) return (0); else return (-1);
}
```

2 *Card classification algorithm*

The second composite state `Check card type` given in Figure 3(b) uses a simple algorithm that checks if a valid code is palindrome or not. The corresponding C function for such algorithm is given as follows:

```
int Checking_Palindrome(long Code)
{ char tabcode[]; char reversed_code[];
  int i;
  itoa(Code,tabcode,10);  // convert Code into string
  int h = strlen(tabcode); // getting tabcode length
  for (i = h - 1; i >= 0 ; i--)
  {
   reversed_code[h - 1 - i] = tabcode[i]; // inverting tabcode characters
  }
  (strcmp(tabcode,reversed_code) == 0) ? return 0 : return 1;
}
```

Now, the more common testing activities would write specific test cases for these two C functions, in which a few developer-selected inputs are used. The appropriate selection of representative cases and corner cases entirely depends on the expertise and ability of the developer. Also, these functionalities are usually tested in isolation (i.e., using unit tests) and the presence of integration tests where the same code is used for testing both algorithms, will likely depend on the complexity of the software, time available, and again, developer expertise.

Instead, we advocate the use of PBT, which adds formalism and increases the expressive power of our testing simultaneously. Using PBT, we describe one general property to test conformity of the implementation with the specification in Figure 2:

```
prop_ticket_machine_prop() ->
  ?FORALL(Cmds, commands(ticket_machine_spec),
  begin
     {_History, _FinalState, Result} = run_commands(Cmds),
     Result == ok
  end).
```

This property is a universally-quantified declarative implementation stating that all *commands* (in PBT terminology, actual calls to the implementation of source code, like the algorithm implementations above) should *run* so that the final result of the considered execution is *ok* (that is to say, free of errors). In other words, given that we specify which calls we want to make, the PBT tool will generate sequence of invocations to be used as test cases, run them, and evaluate them. Sequences of invocations are generated according to both data generation specification and *command* preconditions. Then, their evaluation is based on both actual call execution and *command* postcondition checking.

In the TM case study, we have two commands: `validate_card` (which invokes the `Checking_Validity` function), and `check_card_type` (that invokes `Checking_Palindrome`). These two commands are implemented in the `ticket_machine_spec` file referred by our general property. The implementation of the `validate_card` commands features the following key parts:

```
%% Argument generator
validate_card_args(_S) ->
    [oneof([long(20), % a card number is either any 20-digit long
                    % or a 20-digit long which digits 1, 2, 4 and 8 sum up 20
          ?LET({D1, D2, D3, D4}, digits_that_sum_up_20(),
               to_long([D1, D2, digit(), D3, digit(), digit(), digit(), D4,
                        digit(), digit(), digit(), digit(), digit(), digit()]))])].

digits_that_sum_up_20() ->
    ?SUCHTHAT({D1, D2, D3, D4},
              {digit(), digit(), digit(), digit()}
              D1 + D2 + D3 + D4 == 20.)
```

While we could use only randomly generated 20-digit long inputs as card number, the chances of getting a *valid* card (one where the sum of the digits in Positions 1, 2, 4 and 8 is 20) would be significantly low. This would lead to a lot of non-significant tests. So instead, we force that for each test case, *one of* a totally random, or a random-but-valid-by-construction card number is used. This way our automatic testing will feature both positive and negative tests.

Next, we see that the precondition for the `validate_card` is always true (provided we have generated a `_Card` number), and that we invoke the actual implementation under test when we define the command itself:

```
%% Precondition for validate_card
validate_card_pre(_S, [_Card]) ->
    true.

%% Invokes the actual operation on the SUT
validate_card(Card) ->
    call_cnode('Checking_Validity', Card).

%% Postcondition for validate_card
validate_card_post(_S, [Card], 0) ->
    is_valid_card(Card);
validate_card_post(_S, [Card], -1) ->
    not is_valid_card(Card).

is_valid_card({P1,P2,_,P4,_,_,_,P8,_,_,_,_,_,_}) ->
    (P1+P2+P4+P8 == 20).
```

We now observe how the evaluation of this command is performed. A *true* postcondition marks a passed test case, which can be because the `Card` *is a valid card* and the implementation returns 0, or because it is *not* and the implementation, consequently, returns –1. Any mismatch to this expectation would mean a failure in the software has been detected.

Respectively, the implementation of the `check_card_type` commands looks as shown below. The most notable differences are the absence of an argument generator, since a `check_card_type` command has to be always preceded by a `validate_card` command, which already generates a `Card`. Instead, we do have a precondition, which also checks that the card has not been already used (i.e., has not been blacklisted yet).

```
%% Precondition for check_card_type
check_card_type_pre([], _) -> % empty list means we did not pass validate_card
    false;                  % for any cards yet
check_card_type_pre(ListOfCards, [_Card]) ->
    not list:member(Card, ListOfCards).

%% Argument generator just relays the card already in use
check_card_type_args([Card]) ->
    [Card].

%% Invokes the actual operation on the SUT
check_card_type(Card) ->
    call_cnode('Checking_Palindrome', Card).

%% Postcondition for check_card_type
check_card_type_post(_S, [Card], 0) ->
    is_palindrome(Card);
check_card_type_post(_S, [Card], 1) ->
    not is_palindrome(Card).

is_palindrome(Card) ->
    CardAsList = digits_to_list(Card),
    CardAsList == lists:reverse(CardAsList).
```

As for the postcondition, again we check the coherence between the implementation's responses (0 or 1) with the corresponding classification of the card number as

palindrome or not. Figure 4 shows the successful execution of 100,000 different test cases.

Figure 4 PBT execution report

```
Erlang/OTP 22 [erts-10.4.3] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1]

Eshell V10.4.3  (abort with ^G)
1> eqc:start().
Starting Quviq QuickCheck version 1.44.1
   (compiled for R21 at {{2018,7,3},{9,13,55}})
   (Warning: You are using R22)
Licence for University of A Coruna reserved until {{2019,7,12},{19,11,57}}
ok
2> eqc:quickcheck(eqc:numtests(100000,ticket_machine_prop:prop_ticket_machine_prop())).
.................................................................................................(x10).
................................................................................................(x100).
................................................................................................(x1000)
.......................................................................................(x100).........
OK, passed 100000 tests

67.8377% {ticket_machine_prop,validate_card,1}
32.1623% {ticket_machine_prop,check_card_type,1}

length:    Count: 100000   Min: 0   Max: 203   Avg: 15.630710   StdDev: 16.843149   Total: 1563071
```

Interesting data reported by the QuickCheck tool about these test cases is a maximum test sequence length of 203 (i.e., different cards used in the same test, that would account for number of workshop attendees), an average of 15.6, and a total of more than 1.5 million of invocations to the actual implementations. Very few testing techniques can compete, with the modest effort involved in describing the PBT model (commands, preconditions, postconditions, data generators), with this sort of testing power.

5 Related work

Safety-critical systems have to be developed carefully to prevent loss of life and resources due to system failures. Therefore, specification, verification, validation, and testing are key concepts in the process of their development. In this section, we compare the foundations, advantages and contributions of our approach for the formal analysis of critical systems software architecture with other related research.

Regarding the use of formal methods for the development of critical systems, a substantial amount of work has been done that can be categorised according to the modelling formalism, the specification language as well as the verification and testing techniques used.

On the modelling stage, different approaches have been presented to facilitate the design of critical systems software architectures using UML (Weissnegger et al., 2015, 2016; Kocataş et al., 2016) and Petri nets (Song and Schnieder, 2018; Wu and Zheng, 2018; Singh and Singh, 2019). However, UML lacks a precise semantics which is a serious drawback of UML-based techniques to ensure and guarantee the reliability of the system. Therefore, some works on the transformation (resp, the combination) of UML diagrams into (resp, with) Petri nets (Bennama and Bouabana-Tebibel, 2016; Ding et al., 2016; Meghzili et al., 2017; Shapiro et al., 2002), and other rigorous formalisms (Singh et al., 2016a; Vistbakka and Troubitsyna, 2018; Abbas et al., 2018; Ozkaya and Kose, 2018) have been proposed to fill the gaps of UML, especially in the case of real-time systems specification. On the other hand, Petri nets still generally have some limitations (Ballarini et al., 2011; Kolagari, 2002), as they are combined

or transformed into more formal languages such as B machines (Boudi et al., 2015a, 2015b), Z (Jaidka et al., 2017) as well as they are verified with many model checkers (Chaichompoo et al., 2017; Wolf, 2018; Nigro et al., 2019) for better analysis quality results. In addition, many ADLs have been introduced in the last decades to define and model system architecture prior to system implementation, and can also be used to analyse software architectures. Nevertheless, most of them are informal, domain specific and do not support behavioural descriptions or dynamic verification of architectural properties (De Sanctis et al., 2017; Haider et al., 2018). Therefore, many authors aimed to transform ADLs into more formal formalisms (Blouin and Giese, 2016; Han and Wang, 2016; Mkaouar et al., 2015; Franco et al., 2016) to cover these limitations. The common deficiency of all these works is that they are restricted by utilising just a single formalism at the modelling stage, which limits their utilisation. Therefore, we advocate using rewriting logic for the specification stage to offer a generic approach allowing the use of multi-formalisms at the modelling stage. For instance, rewriting logic has shown its ability and adequateness both as a semantic framework for software architecture description (Jerad et al., 2007; Bruni, 2009; Belala et al., 2010), and as a meta-logic in which many software architecture description formalisms can be represented or translated such as UML diagrams (Durán et al., 2014; Chama et al., 2013; Djaoui et al., 2018a, 2018b), Petri nets (Boucherit et al., 2018b) and ADLs (Ölveczky et al., 2010; Bae et al., 2012). In addition, the use of Real-Time Maude in this context has proved to be very advantageous compared to well-known ADLs (Jerad et al., 2008) and that is why some authors have proposed rewriting logic-based ADLs (Braga and Sztajnberg, 2004; Garlan et al., 2009; Rademaker et al., 2005; Bouanaka and Belala, 2008; Smaali et al., 2013).

As compared to other verification techniques, model checking is a well-established technique that has been widely used for automatically verifying safety-critical systems (Meng, 2019; Ray et al., 2019; Parquier et al., 2016; Ammar et al., 2016) and allowing designers to determine whether or not a system model meets its specification. In order to deal with the state-space explosion problem, the Maude LTL model checker leverages upon both the explicit-state verification and the on-the-fly algorithms category. Therefore, its performance is comparable to that of others well-known tools such as SPIN, UPPAAL and Kronos (Eker et al., 2003; Ölveczky and Meseguer, 2007). Besides being able to use and extends the LTL model-checker with user defined strategies (see Subsection 3.3) to verify the model properties and cater the space-state problem, Maude also provides several analysis tools, including an inductive theorem prover (ITP) (Hendrix et al., 2008; Clavel et al., 2006), timed rewriting for simulation purposes, timed search and time-bounded linear temporal logic model checking as all-in-one toolkit to be used for the verification stage.

Last but not the least, testing is often a mandatory step in the development process of safety-critical systems as it is not possible to absolutely ensure the correctness of the implemented system even if the system model has been formally verified and that its specification has been automatically generated, because of the possibility of errors in its implementation. Thus, we advocate the use of PBT for the testing stage as it has been proved to be an effective automatic technique – based on random generation of data – to reveal implementation errors for safety-critical systems (Fredlund et al., 2015; Santos et al., 2018). Similarly, PBT has also been previously combined with model checking such as in Aichernig and Schumi (2017, 2016). However, these works differ from our approach since both techniques have been defined in a specific programming language

and/or used on specific industrial applications such as in Bentahar and Dssouli (2018) and Enoiu et al. (2016).

Consequently, we can say that we have proposed a generic approach that covers all system development life-cycle stages, in which we can capitalise the complementary nature of various formalisms. In doing so, we have used rewriting logic as a unifying semantic framework in the specification stage. The Maude's LTL model checking tool is used to check model properties, and finally QuickCheck's PBT at the implementation level, to reduce costs, save time and back up the confidence on a given, specific system realisation.

6 Conclusions and future work

The main goal of the present work is to provide a generic approach for the formal analysis of critical systems software architecture that supports multi formalisms and goes beyond the architecture description stage to cover all the development stages: specification, verification and testing for the formal analysis of safety critical systems software architectures. To do so, we start the process of development and analysis by using one of the most appropriate formalisms to describe the software architecture of the studied system, and translate it into rewriting logic. Then, we have proposed to use the Maude model-checker to check the absence of any abnormal behaviour in the proposed architecture of the studied system. In addition, by using QuickCheck specifications and PBT, we intend to verify the sustained compliance of actual proposed code (algorithm) before (or during) its real implementation.

A specific objective of our future work is to develop a complete framework which allows automatic translation of the software architecture models into the rewriting logic and properties to avoid human errors in handling Maude specifications, as well as QuickCheck properties.

Acknowledgements

The present work would be quite incomplete if it was not financially supported by the Grant No. 034/PNE/ENS/Spain/13-14 received through the Algerian ministry of higher education and scientific research. In addition, we would like to thank all MADS research group members at A Coruña University for their help and support to accomplish this work.

References

Abbas, M., Ben-Yelles, C-B. and Rioboo, R. (2018) 'Modelling UML state machines with focalize', *International Journal of Information and Communication Technology*, Vol. 13, No. 1, pp.34–54.

Aichernig, B.K. and Schumi, R. (2016) 'Towards integrating statistical model checking into property-based testing', in *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, IEEE, pp.71–76.

Aichernig, B.K. and Schumi, R. (2017) 'Statistical model checking meets property-based testing', in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, pp.390–400.

Ammar, M., Hamad, G.B., Mohamed, O.A. and Savaria, Y. (2016) 'Efficient probabilistic fault tree analysis of safety critical systems via probabilistic model checking', in *2016 Forum on Specification and Design Languages (FDL)*, IEEE, pp.1–8.

Aoki, T., Satoh, M., Tani, M., Yatake, K. and Kishi, T. (2017) 'Combined model checking and testing create confidence – a case on commercial automotive operating system', in *Cyber-Physical System Design from an Architecture Analysis Viewpoint*, pp.109–132, Springer.

Bae, K., Ölveczky, P.C., Meseguer, J. and Al-Nayeem, A. (2012) 'The SynchAADL2Maude tool', in *Fundamental Approaches to Software Engineering*, pp.59–62, Springer.

Ballarini, P., Djafri, H., Duflot, M., Haddad, S. and Pekergin, N. (2011) 'Petri nets compositional modeling and verification of flexible manufacturing systems', in *2011 IEEE Conference on Automation Science and Engineering (CASE)*, IEEE, pp.588–593.

Batista, T., Oquendo, F. and Leite, J. (2018) 'Modeling and executing software architecture using SysADL', in *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, IEEE, pp.4–5.

Becker, K., Voss, S. and Schätz, B. (2018) 'Formal analysis of feature degradation in fault-tolerant automotive systems', *Science of Computer Programming*, Vol. 154, pp.89–133.

Belala, F., Barkaoui, K. et al. (2010) 'A tile logic based approach for software architecture description analysis', *Journal of Software Engineering and Applications*, Vol. 3, No. 11, p.1067.

Bennama, M. and Bouabana-Tebibel, T. (2016) 'A CTL-based OCL extension using CPN ML for UML validation', *International Journal of Critical Computer-Based Systems*, Vol. 6, No. 4, pp.302–321.

Bentahar, J. and Dssouli, R. (2018) 'Model-based verification and testing methodology for safety-critical airborne systems', in *Proceedings New Trends in Model and Data Engineering: MEDI 2018 International Workshops, DETECT, MEDI4SG, IWCFS, REMEDY*, Springer, Marrakesh, Morocco, 24–26 October, Vol. 929, pp.63–74.

Berger, U., James, P., Lawrence, A., Roggenbach, M. and Seisenberger, M. (2018) 'Verification of the european rail traffic management system in real-time Maude', *Science of Computer Programming*, Vol. 154, pp.61–88.

Blouin, D. and Giese, H. (2016) 'Combining requirements, use case maps and AADL models for safety-critical systems design', in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, pp.266–274.

Bouanaka, C. and Belala, F. (2008) 'Towards a mobile architecture description language', in *IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2008*, IEEE, pp.743–748.

Boucherit, A., Castro, L.M., Khababa, A. and Hasan, O. (2018a) 'Towards the formal development of software based systems: access control system as a case study', *Information Technology and Control*, Vol. 47, No. 3, pp.393–405.

Boucherit, A., Khababa, A. and Castro, L.M. (2018b) 'Automatic generating algorithm of rewriting logic specification for multi-agent system models based on Petri nets', *Multiagent and Grid Systems*, Vol. 14, No. 4, pp.403–418.

Boudi, Z., Collart-Dutilleul, S. et al. (2015a) 'Colored Petri nets formal transformation to b machines for safety critical software development', in *2015 International Conference on Industrial Engineering and Systems Management*, IEEE, pp.12–18.

Boudi, Z., Miloudi, E., Koursi, E. and Collart-Dutilleul, S. (2015b) 'From place/transition Petri nets to b abstract machines for safety critical systems', *IFAC-PapersOnLine*, Vol. 48, No. 21, pp.332–338.

Braga, C. and Sztajnberg, A. (2004) 'Towards a rewriting semantics for a software architecture description language', *Electronic Notes in Theoretical Computer Science*, Vol. 95, pp.149–168.

Bruni, R., Lafuente, A.L. and Montanari, U. (2009) 'Hierarchical design rewriting with Maude', *Electronic Notes in Theoretical Computer Science*, Vol. 238, No. 3, pp.45–62.

Bucchiarone, A., Muccini, H., Pelliccione, P. and Pierini, P. (2004) 'Model-checking plus testing: from software architecture analysis to code testing', in *International Conference on Formal Techniques for Networked and Distributed Systems*, Springer, pp.351–365.

Carvalho, G. and Meira, I. (2019) *Modelling and Testing Timed Data-Flow Reactive Systems in COQ from Controlled Natural-Language Requirements*, arXiv preprint arXiv:1910.13553.

Chaichompoo, O., Thongtak, A. and Vatanawood, W. (2017) 'Transformation of time Petri net into Promela', in *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, IEEE, pp.1–5.

Chama, W., Elmansouri, R. and Chaoui, A. (2013) 'Using graph transformation and Maude to simulate and verify UML models', in *2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE)*, IEEE, pp.459–464.

Clarke, E.M. and Emerson, E.A. (1982) *Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic*, Springer.

Clarke Jr., E.M., Grumberg, O., Kroening, D., Peled, D. and Veith, H. (2018) *Model Checking*, MIT Press.

Clavel, M., Palomino, M. and Riesco, A. (2006) 'Introducing the ITP tool: a tutorial', *J. UCS*, Vol. 12, No. 11, pp.1618–1650.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Talcott, C. (2007) *All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, Springer-Verlag.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J. and Talcott, C. (2011) *Maude Manual (Version 2.6)*, Vol. 1, No. 3, pp.4–6, University of Illinois, Urbana-Champaign.

Cortellessa, V., Eramo, R. and Tucci, M. (2018) 'Availability-driven architectural change propagation through bidirectional model transformations between UML and Petri net models', in *2018 IEEE International Conference on Software Architecture (ICSA)*, IEEE, pp.125–12509.

Cuenot, P., Frey, P., Johansson, R., Lönn, H., Papadopoulos, Y., Reiser, M-O., Sandberg, A., Servat, D., Kolagari, R.T., Törngren, M. et al. (2007) '11 the east-ADL architecture description language for automotive embedded software', in *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, Springer, pp.297–307.

Custódio Soares, J.A. (2017) 'Automatic model transformation from UML sequence diagrams to coloured Petri nets'.

De Sanctis, M., Trubiani, C., Cortellessa, V., Di Marco, A. and Flamminj, M. (2017) 'A model-driven approach to catch performance antipatterns in ADL specifications', *Information and Software Technology*, Vol. 83, pp.35–54.

Ding, J. (2016) 'An approach for modeling and analyzing dynamic software architectures', in *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, IEEE, pp.2086–2092.

Ding, Z., Jiang, M. and Zhou, M. (2016) 'Generating Petri net-based behavioral models from textual use cases and application in railway networks', *IEEE Transactions on Intelligent Transportation Systems*, Vol. 17, No. 12, pp.3330–3343.

Djaoui, C., Kerkouche, E., Chaoui, A. and Khalfaoui, K. (2018a) 'A graph transformation approach to generate analysable Maude specifications from UML interaction overview diagrams', in *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, IEEE, pp.511–517.

Djaoui, C., Kerkouche, E., Chaoui, A. and Khalfaoui, K. (2018b) 'Generating Maude specifications from UML interaction overview diagrams: a graph transformation based approach', in *2018 Fifth International Symposium on Innovation in Information and Communication Technology*, IEEE, pp.1–8.

Durán, F., Roldán, M., Moreno, A. and Álvarez, J.M. (2014) 'Dynamic validation of Maude prototypes of UML models', in *Specification, Algebra, and Software*, pp.212–228, Springer.

Eker, S., Meseguer, J. and Sridharanarayanan, A. (2003) 'The maude LTL model checker and its implementation', in *International SPIN Workshop on Model Checking of Software*, Springer, pp.230–234.

Enoiu, E.P., Čaušević, A., Ostrand, T.J., Weyuker, E.J., Sundmark, D. and Pettersson, P. (2016) 'Automated test generation using model checking: an industrial evaluation', *International Journal on Software Tools for Technology Transfer*, Vol. 18, No. 3, pp.335–353.

Feiler, P.H., Lewis, B.A. and Vestal, S. (2006) 'The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems', in *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, IEEE, pp.1206–1211.

Franco, J.M., Correia, F., Barbosa, R., Zenha-Rela, M., Schmerl, B. and Garlan, D. (2016) 'Improving self-adaptation planning through software architecture-based stochastic modeling', *Journal of Systems and Software*, Vol. 115, pp.42–60.

Fredlund, L-Å., Herranz, Á. and Mariño, J. (2015) 'Applying property-based testing in teaching safety-critical system programming', in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, IEEE, pp.309–316.

Garlan, D., Schmerl, B.R. and Cheng, S-W. (2009) 'Software architecture-based self-adaptation', *Autonomic Computing and Networking*, Vol. 1, pp.31–55.

Gerhart, S., Craigen, D. and Ralston, T. (2012) 'Experience with formal methods in critical systems', *High-Integrity System Specification and Design*, Vol. 413.

Giammarco, K. and Giles, K. (2018) 'Verification and validation of behavior models using lightweight formal methods', in *Disciplinary Convergence in Systems Engineering Research*, pp.431–447, Springer.

Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K. and Jouannaud, J-P. (200) *Introducing OBJ*, Springer.

Haider, U., Woods, E. and Bashroush, R. (2018) 'Representing variability in software architecture: a systematic literature review', *International Journal of Software Engineering and Computer Systems*, Vol. 4, No. 2, pp.19–37.

Han, R. and Wang, S. (2016) 'Transformation rules from AADL to improved colored GSPN for integrated modular avionics', in *2016 11th International Conference on Reliability, Maintainability and Safety*, IEEE, pp.1–6.

Hebert, F. (2019) *Property-Based Testing with PropEr, Erlang, and Elixir: Find Bugs Before Your Users Do*, Pragmatic Programmers, LLC.

Hendrix, J., Meseguer, J. and Sasse, R. (2008) *Maude ITP2.0 Tutorial*, Technical Report, University of Illinois at Urbana-Champaign.

Hinchey, M. and Coyle, L. (2010) 'Evolving critical systems: a research agenda for computer-based systems', in *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, IEEE, pp.430–435.

Jaidka, S., Reeves, S. and Bowen, J. (2017) 'Modelling safety-critical devices: coloured Petri nets and z', in *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM, pp.51–56.

Jerad, C. and Barkaoui, K. (2005) 'On the use of rewriting logic for verification of distributed software architecture description based LFP', in *Rapid System Prototyping*, IEEE, pp.202–208.

Jerad, C., Barkaoui, K. and Touzi, A.G. (2007) 'Hierarchical verification in Maude of LFP software architectures', in *European Conference on Software Architecture*, Springer, pp.156–170.

Jerad, C., Barkaoui, K. and Grissa-Touzi, A. (2008) 'On the use of real-time maude for architecture description and verification: a case study', in *BCS Int. Acad. Conf.*, pp.305–317.

Kamburjan, E.,. Hähnle, R. and Schön, S. (2018) 'Formal modeling and analysis of railway operations with active objects', *Science of Computer Programming*, Vol. 166, pp.167–193.

Kang, E-Y. (2019) *A Formal Verification Technique for Architecture-based Embedded Systems in East-ADL*, arXiv preprint arXiv:1903.06241.

Kocı, R. and Janoušek, V. (2017) 'Specification of requirements using unified modeling language and Petri nets', *International Journal on Advances in Software*, Vol. 10, Nos. 1/2.

Kocataş, A.T., Can, M. and Doğru, A.H. (2016) 'Lightweight realization of UML ports for safety-critical real-time embedded software', in *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, IEEE, pp.258–265.

Kolagari, A.R.T. (2002) *Transformation of Open and Algebraic High-level Petri Net Classes*, Technische Universität Berlin, Fakultät IV, Elektrotechnik und Informatik.

Li, B., Liao, L. and Yu, X. (2017a) 'A verification-based approach to evaluate software architecture evolution', *Chinese Journal of Electronics*, Vol. 26, No. 3, pp.485–492.

Li, C., Yang, H-J. and Liu, H-X. (2017b) 'An approach to modelling and analysing reliability of breeze/ADL-based software architecture', *International Journal of Automation and Computing*, Vol. 14, No. 3, pp.275–284.

Liu, J., Li, T., Ding, Z., Qian, Y., Sun, H. and He, J. (2019a) 'AADL+: a simulation-based methodology for cyber-physical systems', *Frontiers of Computer Science*, Vol. 13, No. 3, pp.516–538.

Liu, S., Ölveczky, P.C., Zhang, M., Wang, Q. and Meseguer, J. (2019b) 'Automatic analysis of consistency properties of distributed transaction systems in Maude', in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp.40–57.

Meghzili, S., Chaoui, A., Strecker, M. and Kerkouche, E. (2017) 'On the verification of UML state machine diagrams to colored Petri nets transformation using Isabelle/HOL', in *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, IEEE, pp.419–426.

Meng, M. (2019) *Method for Model Checking on the Design of Security Checking Software of Safety-Critical Distributed Storage System*, 18 April [online] US Patent App. 16/103, 222.

Mens, T., Magee, J. and Rumpe, B. (2010) 'Evolving software architecture descriptions of critical systems', *Computer*, No. 5, pp.42–48.

Meseguer, J. (2012) 'Twenty years of rewriting logic', *The Journal of Logic and Algebraic Programming*, Vol. 81, No. 7, pp.721–781.

Meseguer, J. (2018) 'Formal design of cloud computing systems in Maude', in *Brazilian Symposium on Formal Methods*, pp.5–19, Springer.

Meseguer, J. (1992) 'Conditional rewriting logic as a unified model of concurrency', *Theoretical Computer Science*, Vol. 96, No. 1, pp.73–155.

Mkaouar, H., Zalila, B., Hugues, J. and Jmaiel, M. (2015) 'From ADLL model to LNT specification', in *Ada-Europe International Conference on Reliable Software Technologies*, Springer, pp.146–161.

Mokhati, F., Badri, M. and Gagnon, P. (2006) 'Translating UML diagrams into Maude formal specifications: a systematic approach', in *SEKE*, Vol. 6, pp.572–577.

Nigro, C., Nigro, L. and Sciammarella, P.F. (2019) 'Modelling and analysis of partially stochastic time Petri nets using UPPAAL model checkers', in *Intelligent Computing-Proceedings of the Computing Conference*, Springer, pp.975–993.

Noulamo, T., Tanyi, E., Nkenlifack, M., Lienou, J-P. and Djimeli, A. (2018) 'Formalization method of the UML statechart by transformation toward Petri nets', *IAENG International Journal of Computer Science*, Vol. 45, No. 4.

Nyberg, M., Gurov, D., Lidström, C., Rasmusson, A. and Westman, J. (2018) 'Formal verification in automotive industry: enablers and obstacles', in *International Symposium on Leveraging Applications of Formal Methods*, pp.139–158, Springer.

Ölveczky, P.C. (2007) *Real-Time Maude 2.3 Manual*.

Ölveczky, P.C. (2018) *Designing Reliable Distributed Systems: A Formal Methods Approach Based on Executable Modeling in Maude*, Springer.

Ölveczky, P.C. and Meseguer, J. (2001) *Specifying and Analyzing Real-Time Object Systems in Real-Time Maude*, Manuscript, Computer Science Laboratory, SRI International.

Ölveczky, P.C. and Meseguer, J. (2007) 'Semantics and pragmatics of real-time Maude', *Higher-Order and Symbolic Computation*, Vol. 20, Nos. 1–2, pp.161–196.

Ölveczky, P.C., Boronat, A. and Meseguer, J. (2010) 'Formal semantics and analysis of behavioral AADL models in real-time Maude', in *Formal Techniques for Distributed Systems*, pp.47–62, Springer.

Oquendo, F. (2016) 'Formally describing the software architecture of systems-of-systems with SosADL', in *2016 11th System of Systems Engineering Conference (SoSE)*, IEEE, pp.1–6.

Ozkaya, M. and Kose, M.A. (2018) 'SAwUML-UML-based, contractual software architectures and their formal analysis using spin', *Computer Languages, Systems & Structures*, Vol. 54, pp.71–94.

Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L. and Pierce, B.C. (2015) 'Foundational property-based testing', in *International Conference on Interactive Theorem Proving*, pp.325–343, Springer.

Parquier, B., Rioux, L., Henia, R., Soulat, R., Roux, O.H., Lime, D. and André, É. (2016) 'Applying parametric model-checking techniques for reusing real-time critical systems', in *International Workshop on Formal Techniques for Safety-Critical Systems*, pp.129–144, Springer.

Pnueli, A. (1981) 'The temporal semantics of concurrent programs', *Theoretical Computer Science*, Vol. 13, No. 1, pp.45–60.

Poppleton, M.R. (2007) 'Towards feature-oriented specification and development with event-b', in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, Springer, pp.367–381.

Puntambekar, A.A. (2008) *Software Engineering: Analytical, Focuses on Results, Provides Advice*, Technical Publications, Pune.

Rademaker, A., Braga, C. and Sztajnberg, A. (2005) 'A rewriting semantics for a software architecture description language', *Electronic Notes in Theoretical Computer Science*, Vol. 130, pp.345–377.

Ray, S., Ghosh, N., Masti, R.J., Kanuparthi, A. and Fung, J.M. (2019) 'Formal verification of security critical hardware-firmware interactions in commercial SoCs', in *Proceedings of the 56th Annual Design Automation Conference 2019*, ACM, pp.43–46.

Rosu, G. (2015) 'From rewriting logic, to programming language semantics, to program verification', in *Logic, Rewriting, and Concurrency*, pp.598–616, Springer.

Rubio, R., Martí-Oliet, N., Pita, I. and Verdejo, A. (2018) 'Parameterized strategies specification in Maude', in *International Workshop on Algebraic Development Techniques*, Springer, pp.27–44.

Santos, A., Cunha, A. and Macedo, N. (2018) 'Property-based testing for the robot operating system', in *A-TEST@ ESEC/SIGSOFT FSE*, pp.56–62.

Sari, B. and Reuss, H-C. (2016) 'A model-driven approach for the development of safety-critical functions using modified architecture description language (ADL)', in *2016 International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles & International Transportation Electrification Conference (ESARS-ITEC)*, IEEE, pp.1–5.

Schneidermeier, T., Burghardt, M. and Wolff, C. (2013) 'Design guidelines for coffee vending machines', in *International Conference of Design, User Experience, and Usability*, Springer, pp.432–440.

Shapiro, S., Lespérance, Y. and Levesque, H.J. (2002) 'The cognitive agents specification language and verification environment for multiagent systems', in *Autonomous Agents and Multiagent Systems: Part 1*, ACM, pp.19–26.

Singh, P. and Singh, L. (2019) 'Verification of safety critical and control systems of nuclear power plants using Petri nets', *Annals of Nuclear Energy*, Vol. 132, pp.584–592.

Singh, M., Sharma, A.K. and Saxena, R. (2016a) 'Formal transformation of UML diagram: use case, class, sequence diagram with z notation for representing the static and dynamic perspectives of system', in *Proceedings of International Conference on ICT for Sustainable Development*, Springer, pp.25–38.

Singh, M., Sharma, A.K. and Saxena, R. (2016b) 'An UML+Z framework for validating and verifying the static aspect of safety critical system', *Procedia Computer Science*, Vol. 85, pp.352–361.

Smaali, S., Choutri, A. and Belala, F. (2013) 'K semantics for dynamic software architectures', in *Proceedings of the International Arab Conference on Information Technology (ACIT'2013)*.

Song, H. and Schnieder, E. (2018) 'Evaluating fault tree by means of colored Petri nets to analyze the railway system dependability', *Safety Science*, Vol. 110, pp.313–323.

Stehr, M-O., Meseguer, J. and Ölveczky, P.C. (2001) 'Rewriting logic as a unifying framework for Petri nets', in *Unifying Petri Nets*, pp.250–303, Springer.

Taoufik, S.R., Tahar, B.M., Layth, S. and Mourad, K. (2017) 'Towards a formal approach for the verification of SCA/BPEL software architectures', in *2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA)*, IEEE, pp.1–6.

Ter Beek, M.H., Gnesi, S. and Knapp, A. (2018) 'Formal methods and automated verification of critical systems'.

Thapaliya, A., Jeong, D. and Kwon, G. (2017) 'Failure analysis in safety critical systems using failure state machine', in *Advances in Computer Science and Ubiquitous Computing*, pp.540–545, Springer.

Vistbakka, I. and Troubitsyna, E. (2018) *Towards Integrated Modelling of Dynamic Access Control with UML and Event-b*, arXiv preprint arXiv:1805.05521.

Weissnegger, R., Kreiner, C., Pistauer, M., Römer, K. and Steger, C. (2015) 'A novel design method for automotive safety-critical systems based on UML/MARTE', in *Proceedings of the 2015 Forum on Specification & Design Languages*, ECI, pp.177–184.

Weissnegger, R., Pistauer, M., Kreiner, C., Schuß, M., Römer, K. and Steger, C. (2016) 'Automatic testbench generation for simulation-based verification of safety-critical systems in UML', in *PECCS*, pp.70–75.

Wolf, K. (2018) 'Petri net model checking with LoLA 2', in *International Conference on Applications and Theory of Petri Nets and Concurrency*, Springer, pp.351–362.

Wu, D. and Zheng, W. (2018) 'Formal model-based quantitative safety analysis using timed coloured Petri nets', *Reliability Engineering & System Safety*, Vol. 176, pp.62–79.

Xie, W., Zhu, H., Zhang, M., Lu, G. and Fang, Y. (2018) 'Formalization and verification of mobile systems calculus using the rewriting engine Maude', in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, Vol. 1, pp.213–218.

Zhang, P., Muccini, H. and Li, B. (2010) 'A classification and comparison of model checking software architecture techniques', *Journal of Systems and Software*, Vol. 83, No. 5, pp.723–744.

Zhang, C., Ma, Y., Wang, X. and Wang, R. (2017) 'Software architecture modeling and reliability evaluation based on Petri net', in *2017 International Conference on Dependable Systems and Their Applications (DSA)*, IEEE, pp.51–56.

Zhu, D., Tan, H. and Yao, S. (2018) 'Petri nets-based method to elicit component-interaction related safety requirements in safety-critical systems', *Computers & Electrical Engineering*, Vol. 71, pp.162–172.