A Formal Verification Tool for UML Behavioral Diagrams

Luciana Brasil Rebelo dos Santos, Eduardo Rohde Eras, Valdivino Alexandre de Santiago Júnior, and Nandamudi Lankalapalli Vijaykumar

Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, SP, Brazil

Abstract. Unified Modeling Language (UML) is considered a standard for modeling object-oriented software. It supports several different diagrams that can be used to model behavior and structure of the software. With respect to formal verification, particularly Model Checking, the existing approaches are usually restricted to a single UML diagram. This paper presents a tool to convert UML behavioral diagrams (sequence, activity, and state machine) into Transition Systems to support software Model Checking. A peculiar feature of our tool is that it is developed as part of a larger effort to allow Model Checking of software built in accordance with UML, including several UML behavioral diagrams. We demonstrate the effectiveness of our approach by applying it to a classic case study and also to a real case study (embedded software) in the space domain.

1 Introduction

A major challenge in software and systems development process is to advance defect detection at early stages of their life-cycles. Formal methods offer a large potential to obtain an early integration of verification in the design process, and to provide more effective verification techniques [1]. Besides, formal verification methods, such as Model Checking, are best applied in early stages of system design, when costs are low and benefits can be high, increasing the quality of systems. However, formal methods require mathematical background and their use is restricted, as users privilege the simplicity of other notations, rather than more formal means. Thus, adoption of formal methods will be easier when they can be applied within standard development processes and when they are based on standard notation [2].

Unified Modeling Language - (UML) [3] is currently accepted as the standard for modeling (object-oriented) software and it has received attention from researchers as well as practitioners. It presents diagrams that represent the static structure of a system, and also defines diagrams to model the dynamic behavior of systems. We make use of UML behavioral diagrams since we are interested in verifying the system behavior. In particular, dynamic aspects of system behavior can be specified by interactions (i.e. sequence diagrams), and activity diagrams give a view of the system that is associated with instances of classes.

Transition System, also called finite-state model, is a standard class of models to represent hardware and software systems [1]. They are often used as models to describe the behavior of systems. Basically, they are directed graphs where nodes represents *states*, and edges model *transitions*, i.e, state changes. Such a system evolves through its state space assuming different configurations, where a configuration can be understood as the set of states to which the system abides at any particular moment [4]. Model Checking is a formal automatic verification technique for finite state systems that checks temporal logic specifications on a given model. In the context of verifying design models expressed as UML activity diagrams, Eshuis and Wieringa [5] explore an idea similar to Transition Systems. However, transition system concept has a general nature and a broad range of behavioral diagrams, such as activity, sequence, and behavioral state machine can be conveniently adapted to use this concept [4].

This work presents a tool that allows the use of Formal Verification on projects developed using UML diagrams. Currently, the tool is able to transform two UML behavioral diagrams (sequence and activity) into individual Transition Systems (TS) to support software Model Checking. We consider properties generated from use case descriptions, which represent the requirements; and the TS translated from behavioral diagrams. The analyst specifies a use case, with its narrative description, and a sequence or activity diagram. The diagram is translated into an individual Transition System and then to the input language of the model checker. The requirement is fed into the model checker. If the requirement is false, the model checker gives a counterexample in the form of a trace.

The rest of the paper is organized as follows. In the next section we give a review of UML behavioral diagrams and Model Checking. In Section 3, the proposed methodology is exposed. We explain the tool including its architecture in Section 4. Section 5 shows the usability aspects of the tool. In Section 6 we apply our tool to one scenario of ATM (Automated Teller Machine) case study and on two scenarios of a space software product. Related work is discussed in Section 7. Finally, Section 8 concludes the paper.

2 Fundamentals

Given a UML behavioral diagram, it is possible to generate the corresponding TS provided that the elements of the diagram (messages, activities, states) are both established and understood and there exists (and is defined) a step relation enabling to systematically compute the next configuration(s) of the diagram from any given configuration [4]. In this section we briefly discuss UML and Model Checking.

2.1 UML

UML [3] is a visual language that has been developed to support the design of complex object-oriented systems. UML diagrams can be divided into two broad categories: structural and behavioral diagrams. The UML structural diagrams are used to model the static organization of the different elements in the system, whereas behavioral diagrams focus on the dynamic aspects of the system [6]. This section discusses only the UML diagrams that are relevant in the context of this work.

Use case: describes how a user interacts with the system by defining the steps required to accomplish a specific goal [7]. Use case diagrams help to determine the functionality and features of the software from the user's perspective. A use case comprises different possible sequences of interactions between the user and the computer. Each specific sequence of interactions in a use case is called a subscenario.

Sequence diagram: a sequence diagram (SD) is used to show the dynamic communications between objects during the execution of a task. An SD describes how groups of objects collaborate on some behavior over time. It registers the behavior of a single use case and displays objects and messages passed between these objects in the use case. The sequence diagram follows the approach based on temporal order of the messages, that is, the emphasis is on the temporal distribution of messages. The system requirements are represented in the use cases, i.e., use cases model **what is** the problem. The sequence diagrams show **how** the model will get the desired objective.

Activity diagram: activity diagram (AD) depicts the dynamic behavior of a system (or part of a system) by means of the flow of control between actions that the system performs [7]. It is similar to a flowchart and can show concurrent flows. Activity diagrams are an extension to the original idea of state machines. They evaluate better the conditions by which the instances come to certain decisions. An activity is a procedure executed while the AD is in a particular state and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transition, then these must be identified through conditions. ADs typically support description of sequencing, conditional dependency, parallel activities, and synchronization aspects involved in different activities.

2.2 Model Checking

According to [1], Model Checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

The Model Checking approach can be viewed in Figure 1. There are properties obtained from the requirements (in our case, from use case descriptions) that reveals what the system should do and not to do. The properties are formalized using some sort of temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) (we formalize properties by means of specification patterns [8]). A model describes the system behavior (we generate the model from UML behavioral diagrams). The model checker examines all system states to check whether they satisfy the desired property. If a state violates the property under consideration, the model checker provides a counterexample showing a trace that indicates the violation.

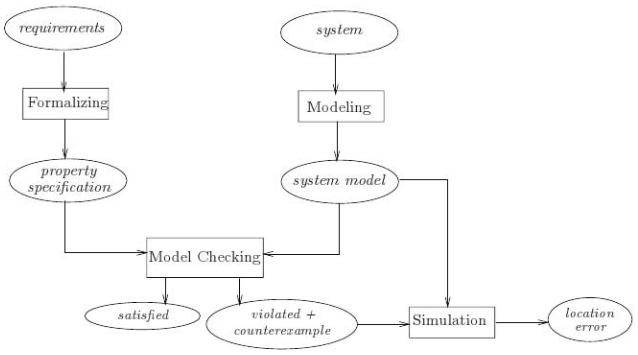


Fig. 1. Schematic view of Model Checking. Source: [1]

3 Proposed Approach

A prerequisite for Model Checking is a property to be checked and a model of the system under consideration. In Figure 2, we show our approach aiming at Model Checking of software developed in accordance with UML. In the following, we explain the main activities that the workflow performs:

(i) **identify scenarios** by looking at use case models. A use case can be viewed as a scenario. Each scenario is a set of related subscenarios tied together by a common goal. The mainline sequence ('main success scenario' [9]) and each of the variations ('extensions and sub-variations') are the scenarios identified by our approach; (ii) **formalize properties**. For each selected scenario, we extract requirements from the textual description of use cases. After that, we formalize properties by means of specification patterns [8]; (iii) **generate TS**. Based on the available UML behavioral diagrams, we generate individual TSs and then a unified TS (the unified TS is being developed). This is the main activity to achieve the objective of our approach. As we will explain later, our approach does not demand that all three UML behavioral diagrams (sequence, activity, behavioral state machines) exist: it is enough to have the sequence diagram and one of the other two to generate the TS; (iv) **generate model checker notation**. Then, we translate the created TS to a model checker. Our first idea is to use the NuSMV model checker [10]; (v) Finally, we **apply Model Checking** to realize about defects in the behavioral description of the system represented by the UML diagrams. We repeat activities from (ii) to (v) for each selected scenario. Also note that activities (ii) and (iii)/(iv) may be accomplished in parallel. Relating to acitivity (ii) the requirements are identified using use case descriptions and then, formalized using using specification patterns as proposed by [8]. This activity of obtaining the formalized properties depends on the human factor and the professional knowledge of the requirements to identify which is the best pattern/pattern scope. More details can be found on [8].

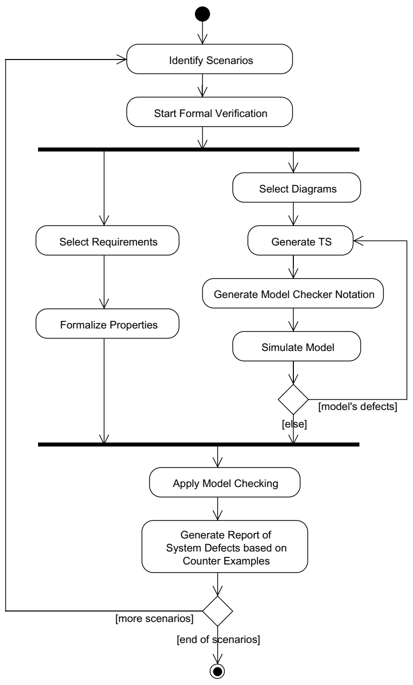


Fig. 2. Workflow of the proposed approach

4 Software Tool

We are developing a tool aiming to support the approach that is detailed in Section 3. It has been implemented using Java. We have considered UML 2.0 specifications and have used Papyrus [11] to produce the design artifacts. The design artifacts are then exported into XMI (XML Metadata Interchange) format, and are inputted to our tool. The high level architecture of our tool is shown in Figure 3. The three major modules of our implementation are: a **reader** module, a **converter** module, and **TUTS** (The Unified Transition System). Blue lines show the part that is already implemented. The state machine diagram converter, the TUTS, as well as the translation to the model checker input language are under development.

The reader module is fed with three XMI files, relating to UML sequence, state machines, and activity diagrams. The reader module puts all the information of each file in linked lists that are directed to the converter. The converter translates the linked lists to individual transition systems. Finally, the TUTS joins the three diagrams into a Unified Transition System.

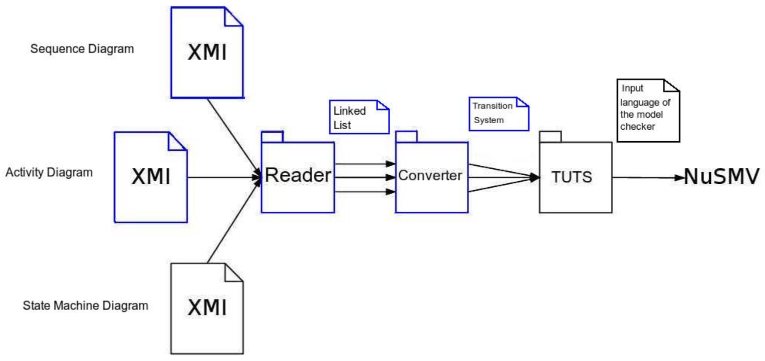
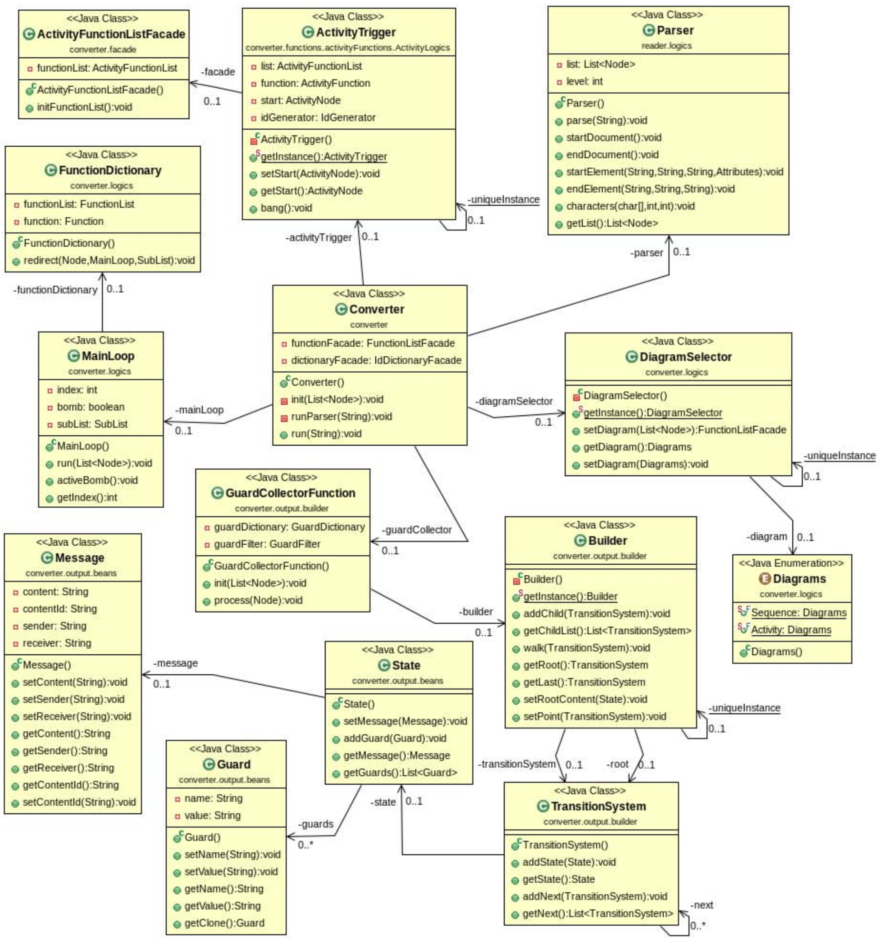


Fig. 3. Tool Architecture

After its creation, the unified TS can be used to systematically generate its corresponding encoding into the model checker input language by constructing declarative divisions [4]. It is important to emphasize that once a formal unified TS was generated from UML behavioral diagrams, we see the possibility of transforming it into several different languages of available model checkers such as SPIN [12], UPPAAL [13], and NuSMV [10]. In our current approach, we chose the NuSMV model checker because it is open source, it has a widespread use in academia, and it accepts properties formalized not only in Computation Tree Logic (CTL) but also in Linear Temporal Logic (LTL) [1], two logics that are well known and have mappings defined in the specification patterns [8].

The **reader** module uses an API (Application Programming Interface) to parse the XMI input file called SAX (Simple API for XML). A Java class (*Parser* class, in our tool) that extends the SAX API implementing its functions is used. While reading the file, the API triggers an event to every tag of the XMI input. This event is captured by the functions implemented in the reading class and makes the treatment for each case. Each tag read is then saved in a linked list of objects that contains all information relevant to their future processing.

The input to the **converter** module (*Converter* class) is precisely the list generated by the reader module. This list contains all the tags of the XMI input sorted in the order they appear in the document. To process this list, there is a main loop (*MainLoop* class) that goes through each item (tag) from this list and directs its contents to a dictionary of functions (*FunctionDictionary* class). The dictionary of functions contains functions specific to the treatment of each tag that may appear in the XMI input file. Two dictionaries of functions were created, one for each diagram, due to differences between the XMI files of sequence and activity diagrams. However, the operation remains the same: for each item read by the main loop it is called a function of the dictionary which will treat that item. Once called the function, each rendered element generates a state for the final output, the individual Transition System (TS). These states are created and stored by the *Builder* class in a linked list. This is the output of the converter module. Figure 4 shows the classes diagram for the reader and converter modules. In the bottom of Figure 4, we can see the basic classes *Message* and *Guard*, which compose the class *State*, which, in turn, composes the class *TransitionSystem*. We can also see the classes *DiagramSelector* and *Diagrams*, which selects the type of diagram to process.



<<Java Class>> ActivityFunctionListFacade converter.facade
functionList: ActivityFunctionList
ActivityFunctionListFacade()
initFunctionList():void
-facade 0..1
<<Java Class>> ActivityTrigger converter.functions.activityFunctions.ActivityLogics
list: ActivityFunctionList
function: ActivityFunction
start: ActivityNode
idGenerator: IdGenerator
ActivityTrigger()
getInstance():ActivityTrigger
setStart(ActivityNode):void
getStart():ActivityNode
bang():void
-uniqueInstance 0..1
<<Java Class>> Parser reader.logics
list: List<Node>
level: int
Parser()
parse(String):void
startDocument():void
endDocument():void
startElement(String,String,String,Attributes):void
endElement(String,String,String):void
characters(char[],int,int):void
getList():List<Node>
<<Java Class>> FunctionDictionary converter.logics
functionList: FunctionList
function: Function
FunctionDictionary()
redirect(Node,MainLoop,SubList):void
-functionDictionary 0..1
-activityTrigger 0..1
-parser 0..1
<<Java Class>> Converter converter
functionFacade: FunctionListFacade
dictionaryFacade: IdDictionaryFacade
Converter()
init(List<Node>):void
runParser(String):void
run(String):void
<<Java Class>> MainLoop converter.logics
index: int
bomb: boolean
subList: SubList
MainLoop()
run(List<Node>):void
activeBomb():void
getIndex():int
-mainLoop 0..1
-diagramSelector 0..1
<<Java Class>> DiagramSelector converter.logics
DiagramSelector()
getInstance():DiagramSelector
setDiagram(List<Node>):FunctionListFacade
getDiagram():Diagrams
setDiagram(Diagrams):void
-uniqueInstance 0..1
<<Java Class>> GuardCollectorFunction converter.output.builder
guardDictionary: GuardDictionary
guardFilter: GuardFilter
GuardCollectorFunction()
init(List<Node>):void
process(Node):void
-guardCollector 0..1
<<Java Class>> Builder converter.output.builder
Builder()
getInstance():Builder
addChild(TransitionSystem):void
getChildList():List<TransitionSystem>
walk(TransitionSystem):void
getRoot():TransitionSystem
getLast():TransitionSystem
setRootContent(State):void
setPoint(TransitionSystem):void
-builder 0..1
-uniqueInstance 0..1
-diagram 0..1
<<Java Enumeration>> Diagrams converter.logics
Sequence: Diagrams
Activity: Diagrams
Diagrams()
<<Java Class>> Message converter.output.beans
content: String
contentId: String
sender: String
receiver: String
Message()
setContent(String):void
setSender(String):void
setReceiver(String):void
getContent():String
getSender():String
getReceiver():String
getContentId():String
setContentId(String):void
-message 0..1
<<Java Class>> State converter.output.beans
State()
setMessage(Message):void
addGuard(Guard):void
getMessage():Message
getGuards():List<Guard>
-transitionSystem 0..1
-root 0..1
<<Java Class>> Guard converter.output.beans
name: String
value: String
Guard()
setName(String):void
setValue(String):void
getName():String
getValue():String
getClone():Guard
-guards 0..*
-state 0..1
<<Java Class>> TransitionSystem converter.output.builder
TransitionSystem()
addState(State):void
getState():State
addNext(TransitionSystem):void
getNext():List<TransitionSystem>
-next 0..*

Fig. 4. Summarized class diagram of reader and converter modules

With respect to sequence diagrams, combined fragments can appear. They are repetition structures (loops, alt, opt, ...). When found by the dictionary of functions, these elements access specific functions for their treatment. These functions control the main loop, initiating new instances of it or ending the execution of the loop in order to drive the conversion of the XMI file, as the repetition fragment requires. During the conversion of activity diagrams, the main loop is used only to sort the elements that arrive out of order in XMI. Once sorted, a class *Trigger* triggers the first function for reading the first element. Then, when the element is read and processed the next element is called, as well as its function. Each element acts on its own without the need of the main loop. Decision-making structures and repetition are treated here by the tasks that make your calls according to the read element. Treatment of parallelism in activity diagram is based on the generation of all possibilities. These description can be seen in Figure 5 as an activity diagram, showing activities (representing the flow of the classes), as well as the objects created.

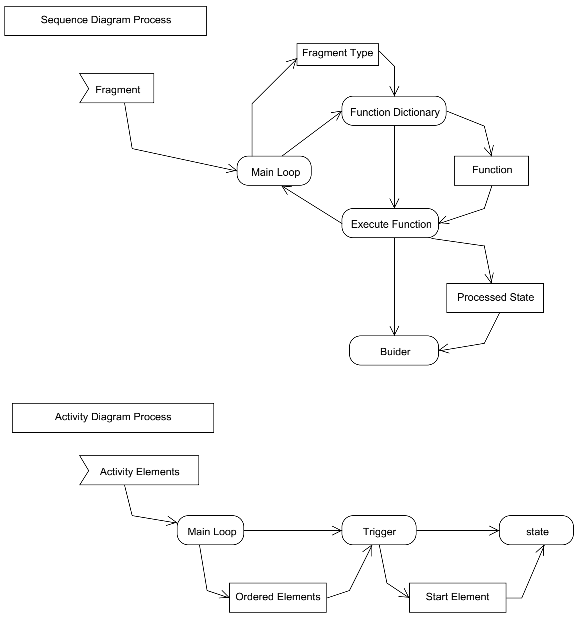


Fig. 5. Activity diagram of the tool behavior when processing sequence and activity diagrams

5 Usability Aspects

Observing the proposed approach in Section 3, on one side we have the properties, which are manually specified, and on the other side we have the system model, which is automatically generated. Regarding the generation of the system model, all the user needs to do is to model the UML diagrams using Papyrus [11], exactly as shown in Figure 6.

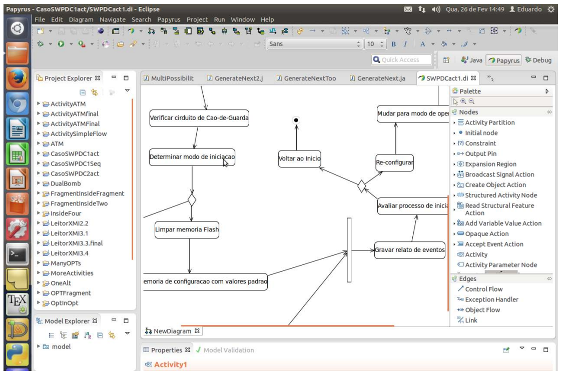

Papyrus - CasoSWPDC1act/SWPDCact1.di - Eclipse
File Edit Diagram Navigate Search Papyrus Project Run Window Help
Project Explorer
Model Explorer
Palette
Nodes
Edges
Verificar cirduito de Cao-de-Guarda
Determinar modo de iniciacao
Limpar memoria Flash
Voltar ao Inicio
Mudar para modo de ope
Re-configurar
Avaliar processo de inicia
Gravar relato de eventos
emoria de configuracao com valores padrao
Properties
Activity1

Fig. 6. Screen of Papyrus

We are using eclipse [14], one of the most used IDE for Java programming, and Papyrus is a plugin for eclipse. Papyrus is aiming at providing an integrated environment for editing any kind of EMF (Eclipse Modeling Framework) model and particularly supporting UML and related modeling languages such as SysML and MARTE. It automatically generates the XMI file related to the UML diagram that we use as input for the reader module.

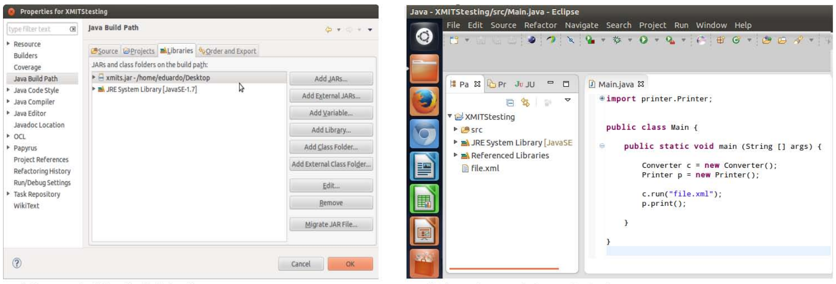
After the diagrams are modeled in Papyrus, it is necessary to generate an XMI file for each one of them. Our tool is distributed in a Java package. This package should be added to the *build path* of the project, as can be seen in Figure 7 a). Once the package is added to the project, it is sufficient to import our tool in a java class, as follows:

```
import converter.converter;
```

Then, one must save the XMI file in the project root and pass its name as a parameter in the *run()* function of the converter object. In Figure 7 b) it is possible to see a Java class, which contains all the described steps to run the tool, as well as the directory in the left, containing the XMI file and the project (Papyrus generates XMI files with extension XML).

```
c.run("arquivo.xml");
```

When executing this command, the tool will run with the mentioned file and display the output on the console. The output is exactly an individual Transition System. When the tool is finished, the input language of the model checker will be automatically generated, as well.


Properties for XMITStesting
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
Cancel
OK
Java - XMITStesting/src/Main.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
XMITStesting
src
JRE System Library [JavaSE
Referenced Libraries
file.xml
Main.java

```
xmits.jar - /home/eduardo/Desktop
JRE System Library [JavaSE-1.7]
```

```java
import printer.Printer;

public class Main {

    public static void main (String [] args) {

        Converter c = new Converter();
        Printer p = new Printer();

        c.run("file.xml");
        p.print();

    }

}
```

a) Screen of adding the build path

b) Java class ready to run the tool

Fig. 7. Screens of Eclipse

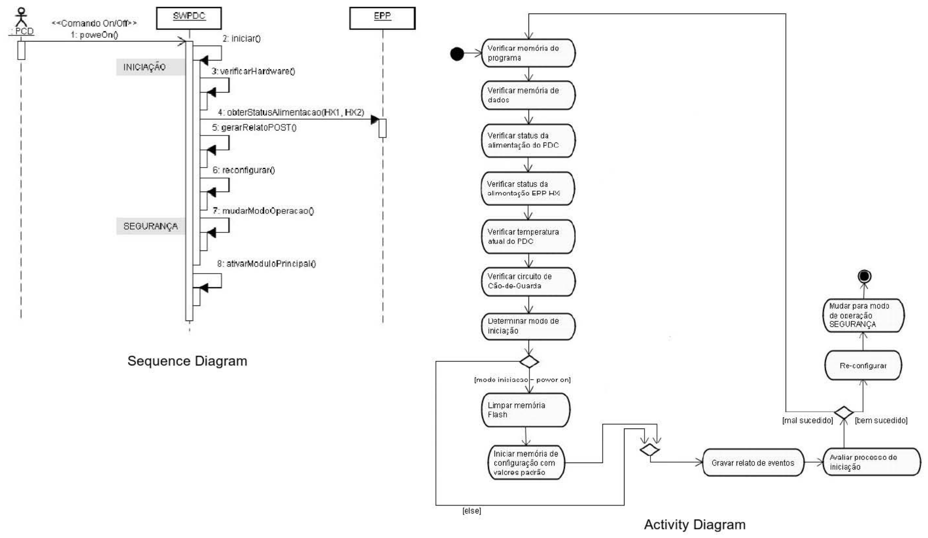
6 Case Studies

In order to verify the effectiveness of our approach, we have carried out two case studies. In our experiments we have considered five different diagrams: two from ATM (where the ATM interacts with a potential customer via a specific interface and communicates with the bank over an appropriate communication link), and three from SWPDC (Software for the Payload Data Handling Computer), a real space software product. As we have already stated, the tool is not totally implemented and we will describe all steps we have performed to get the results, including some mannual activities that will be automated when the tool is totally finished. Due to space constraints, we will detail only one scenario, showing how to execute all activities proposed in the approach, until the final results. With respect to other scenarios, we will describe only the final results.

SWPDC is a space application software product developed in the context of the QSEE (Quality of Space Application Embedded Software) research project [15]. QSEE was an experience in outsourcing the development of software embedded in satellite payload. INPE was the customer and there were two SWPDC's suppliers: INPE itself and a brazilian software company. SWPDC has the following computing units: Payload Data Handling Computer (PDC), Event Pre-Processors (EPPs), and On-Board Data Handling (OBDH) Computer.

In accordance with our approach, we must **identify scenarios**, observing use cases. The scenario we detail is *Startup of PDC*. In this scenario, the main actor is the PCD (Power Conditioning Unit) that switches the PDC computer on. The flow involves: to accomplish hardware verification, to obtain current PDC temperature, to generate startup report, to configure PDC state with standard values, and to divert the control to the main module when in safety operation mode. Two diagrams are related to this scenario, sequence and activity, as shown in Figure 8.

Then, we **start Formal Verification**. For this, we **generate TS** and **select requirements**. The tool outputs for the sequence and activity diagrams are shown in Figure 9. In the left, we can see the individual TS obtained from sequence diagram, with 10 reachable states. In the right, Figure 9 exhibits part of



: PCD
<<Comando On/Off>>
1: poweOn()
SWPDC
EPP
2: iniciar()
INICIAÇÃO
3: verificarHardware()
4: obterStatusAlimentacao(HX1, HX2)
5: gerarRelatoPOST()
6: reconfigurar()
7: mudarModoOperacao()
SEGURANÇA
8: ativarModuloPrincipal()
Sequence Diagram
Verificar memória do programa
Verificar memória de dados
Verificar status da alimentação do PDC
Verificar status da alimentação EPP HX
Verificar temperatura atual do PDC
Verificar circuito de Cão-de-Guarda
Determinar modo de iniciação
[modo iniciacao = power on]
Limpar memória Flash
Iniciar memória de configuração com valores padrão
[else]
Gravar relato de eventos
Avaliar processo de iniciação
[mal sucedido]
[bem sucedido]
Re-configurar
Mudar para modo de operação SEGURANÇA
Activity Diagram

Fig. 8. Sequence and activity diagrams for scenario *Startup of PDC*

the individual TS obtained from activity diagram. In total, we have 135 states, 26 of which are reachable states when running NuSMV model checker. Each state is characterized by the values of the variables (**generate model checker notation**). We have identified three main variables that characterize the TS obtained from activity diagram: (i) $State = \{Start, Verificarmemoriadeprograma, ...\}$; (ii) $bemSucedido = \{dc, false, true\}$. *bemSucedido* represents the evaluation of the startup process; and (iii) $modoIniciacaoPowerOn = \{dc, false, true\}$. *modoIniciacaoPowerOn* represents the determination of the startup mode (power on or reset command). *bemSucedido* and *modoIniciacaoPowerOn* comes from the guards identified within the activity diagram. Regarding the TS obtained from sequence diagram, we have identified only one variable: $State = \{Start, powerOn, ...\}$.

Continuing the approach, we can extract four relevant user-defined properties for this scenario, related to requirements. To proceed with the Formal Verification, it is necessary to **formalize the properties** to be checked. We chose Computation Tree Logic (CTL) [1] to formalize the properties. Note that the properties could be formalized using LTL as well, considering that NuSMV supports such logic.

Requirement 1: *The POST (Power-On Self Test) shall comprise: (i) Power status of the PDC itself; (ii) Power satus of the two EPP-HXI sets; (iii) Current internal temperature of the PDC; (iv) Coherent information of the PDC Program Memory; (v) Reading of SRAM (Data Memory) and; (vi) Correct operation of the watchdog timer circuit.* This requirement can be mapped into several properties wich can be formalized using the **Existence Pattern and Globally Scope** proposed by [8], in CTL, as follows[1]:

[1] We used the NUSMV's syntax to write the property in CTL.

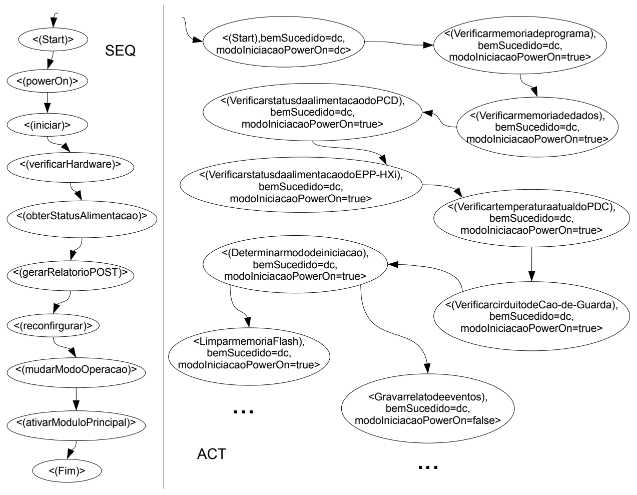

SEQ
<(Start)>
<(powerOn)>
<(iniciar)>
<(verificarHardware)>
<(obterStatusAlimentacao)>
<(gerarRelatorioPOST)>
<(reconfirgurar)>
<(mudarModoOperacao)>
<(ativarModuloPrincipal)>
<(Fim)>
ACT
<(Start),bemSucedido=dc, modoIniciacaoPowerOn=dc>
<(Verificarmemoriadeprograma), bemSucedido=dc, modoIniciacaoPowerOn=true>
<(Verificarmemoriadedados), bemSucedido=dc, modoIniciacaoPowerOn=true>
<(VerificarstatusdaalimentacaodoPCD), bemSucedido=dc, modoIniciacaoPowerOn=true>
<(VerificarstatusdaalimentacaodoEPP-HXi), bemSucedido=dc, modoIniciacaoPowerOn=true>
<(VerificartemperaturaatualdoPDC), bemSucedido=dc, modoIniciacaoPowerOn=true>
<(VerificarcirduitodeCao-de-Guarda), bemSucedido=dc, modoIniciacaoPowerOn=true>
<(Determinarmododeiniciacao), bemSucedido=dc, modoIniciacaoPowerOn=true>
<LimparmemoriaFlash), bemSucedido=dc, modoIniciacaoPowerOn=true>
<Gravarrelatodeeventos), bemSucedido=dc, modoIniciacaoPowerOn=false>
...
...

Fig. 9. TS obtained from sequence diagram and part of the TS obtained from activity diagram of *Startup via PDC* scenario

$AF(State = Verificarmemoriadeprograma)$
$AF(State = Verificarmemoriadedados)$
$AF(State = VerificarstatusdaalimentacaodoPCD)$
$AF(State = VerificarstatusdaalimentacaodoEPP - HXi)$
$AF(State = VerificartemperaturaatualdoPDC)$
$AF(State = VerificarcirduitodeCao - de - Guarda)$

To formalize the properties, it is necessary to see the TSs generated by the tool and its variables. Each one of the six items related to requirement 1 are represented by one state in the TS obtained from activity diagram. When running NuSMV (**apply Model Checking**), these properties are all true. Now, to the TS obtained from sequence diagram, four of the six items are false, that is, the diagram does not reflect this requirement.

Requirement 2: *The SWPDC should know how to distinguish between a power-on process and a reset process.* This property can be formalized using the **Existence Pattern and Scope After Q** proposed by [8], in CTL, as follows:

$!E[!((State = Determinarmododeiniciacao \& modoIniciacaoPowerOn = true) \&$
$AF(State = LimparmemoriaFlash))U((State = Determinarmododeiniciacao \&$
$modoIniciacaoPowerOn = true) \& !((State = Determinarmododeiniciacao \&$
$modoIniciacaoPowerOn = true) \& AF(State = LimparmemoriaFlash)))]$

The TS obtained from activity diagram has a variable that distinguish between a power-on process and a reset process (*modoIniciacaoPowerOn*). After the state *Determinarmododeiniciacao* if *modoIniciacaoPowerOn=true* the next state should be *LimparmemoriaFlash*. When running NuSMV, this property is true. Relating to the TS obtained from sequence diagram, it is not possible to distinguish between the two startup modes. The diagram does not meet this requirement.

Requirement 3: *The SWPDC should report processing of the POST through reports of events.* This property can be formalized using the **Existence Pattern and Globally Scope** proposed by [8], in CTL, as follows:
$AF(State = Gravarrelatodeeventos)$
$AF(State = gerarRelatorioPOST)$
The first is for the TS obtained from the activity diagram. The second is for the TS obtained from sequence diagram. Both diagrams meet this requirement.

Requirement 4: *In the case of any unrecoverable problem not being identified in the PDC after the startup process, the PDC shall automatically enter into the safety operation mode.* This property can be formalized using the **Response Pattern and Globally Scope** proposed by [8], in CTL, as follows:
$AG((State = Avaliarprocessodeiniciacao \& bemSucedido = true) - > AF(State = MudarparamododeoperacaoSEGURANCA))$
In the TS obtained from activity diagram, there is a variable that indicates if the startup process is successful (*bemSucedido=true*) or not (*bemSucedido=false*). After state *Avaliarprocessodeiniciacao*, in the case of *bemSucedido=true*, it is possible to reach state *MudarparamododeoperacaoSEGURANCA*, which represents that the PDC enters the safety operation mode. Otherwise, the PDC remains in the startup operation mode. When running NuSMV, this property is true. For the TS obtained from sequence diagram, this property is false. The variable that indicates if the startup process is successful (*bemSucedido*) is not available for this diagram. Regardless of whether or not the startup has problems, the PDC enters in the safety operation mode. Thus, the diagram does not meet this requirement.

We can note that the TS obtained from the activity diagram meets all requirements verified. However, the TS obtained from the sequence diagram reflects only requirement 3, and does not meet all the other three requirements verified.

We have also worked with two other scenarios: one from SWPDC (*Activation of the main module*), with one activity diagram, and one scenario from ATM (*Perform Transaction*), with two diagrams: a sequence and an activity diagram. For the first scenario, we have obtained 351 states, 85 of which are reachable states when running NuSMV model checker. Three requirements were verified and for one of them, the TS did not meet the requirement. For the second scenario, we have obtained 513 states, 53 of which are reachable states when running NuSMV model checker for the sequence diagram, and we have obtained 1218 states, 86 of which are reachable states when running NuSMV model checker for the activity diagram. The two TSs obtained from the two diagrams meet the requirements verified.

In this section, we have shown the use of our tool, by means of two case studies, generating individual TSs from sequence and activity UML diagrams, and we introduced policies to transform the TSs into the NuSMV input language. Besides, after running the model checker for the explained scenario, we have found that the sequence diagram modeling the space application product does not comply with all the specified requirements.

7 Related Work

This section presents some of the research literature related to this paper, showing (not exhaustive) tool approaches that use Formal Verification and UML.

Mikk [16] and Latella [17] translated Statecharts into PROMELA, the input language of SPIN verification system. Lam [18] formally analyzed activity diagrams using NuSMV model checker. The objective was determining the correctness of activity diagrams. Eshuis [5] presented two translations from activity diagrams to NuSMV. The aim was to assess the activity diagrams from the point of view of requirements and also from the point of view of implementation, which represents the current system behavior. Dubrovin [19] implemented a tool that translates UML hierarchical state machine models to the input language of NuSMV. Uchitel [20] proposes translation of scenarios, specified as Message Sequence Charts (MSCs), into a specification in the form of Finite Sequential Processes. This can then be fed to the Labelled Transition System Analyser model checker to support system requirements validation. All these previous studies deal with a single UML or UML-like diagram to perform Formal Verification. Rather, our tool allows to work with up to two UML behavioral diagrams (and more diagrams in the future). In addition, it is not clear if in the previous studies the authors used specification patterns to formalize the properties. Specification patterns provide clear guidelines to such formalization.

Baresi [21] developed a tool to carry out Formal Verification of UML-based models, mainly interested in the timing aspects of systems. It is composed of: static part (class diagrams); dynamic aspects and behavior are rendered through: (a) state diagrams and activity; (b) sequence diagrams; and (c) interaction overview diagrams, used to relate different sequence diagrams; Clocks (and time diagrams) are used to add the time dimension to systems. All these diagrams seem to be required to construct the approach.

Cortellessa [22] proposes a methodology Performance Incremental Validation in UML (PRIMA-UML) aimed at generating a queueing network based performance model from UML diagrams that are usually available early in the software lifecycle (use case, sequence, and deployment).

The main motivation of our approach is the practical use of formal methods in software development, through automation. The idea is that the user feeds the tool with UML behavioral diagrams and it shows the defects. This can be done throughout the lifecycle, even before software coding. [22] suggested an interesting approach to encompass performance validation task as an integrated activity within the development process. We aim at detecting design defects within the solution, but considering functional requirements of the software product. We are proposing, rather than a tool, an approach to detect defects in the design of software developed in accordance with UML. Besides, our approach suggests to use different behavioral diagrams, which represent complementary views of system behavior and are often used in different phases of software specification and design, allowing thus a wider system range to be verified. Most of the studies we mentioned deal with a specific type of UML diagram. On the other hand, Baresi [21] seems to require an assorted number of diagrams (structural, behavioral and timing diagrams), which are not always available on the documentation.

8 Conclusions

In this paper we presented a tool that, ultimately, is another initiative in order to facilitate and thus increase the use of formal methods in software real projects. We draw on two facts to the development of this work. First, UML is a language widely used in various application domains, including the aerospace one. Second, Formal Verification and formal methods in general, despite all the benefits already presented by academic community, have not seen widespread adoption as a routine part of systems development practice [23].

We presented two case studies (containing three scenarios) running our tool. We have detailed one scenario, showing how to perform the activities proposed in the approach. We have shown the results for the three scenarios.

It is noteworthy that our tool supports the automation of a methodology that aims to add value to software products, which are specified and designed by means of UML. As seen in Figure 2 and detailed in the case studies section (Section 6), our methodology not only involves the automatic transformation of multiple perspectives of behavioral modeling (sequence, activity, behavioral state machines) in order to accomplish Model Checking, but also this methodology provides a way of formalizing the properties through specification patterns. From a practical standpoint, hints of how to formalize properties are of great value to find defects in design solutions for critical systems.

Future directions follow. We are currently developing the other modules of the tool. When State Machine Diagram transformation is finished we will begin the construction of the final part, the unified TS generation. Another direction is to automatically identify in the UML diagrams a problem (inconsistency between diagrams, incorrect behavior) when a counterexample is detected by running NuSMV. We will dedicate efforts in transforming the unified TS to other model checkers such as SPIN and UPPAAL.

Acknowledgement. This work is supported in part by Financiadora de Estudos e Projetos (FINEP) under Project Number 01.10.0233.00, and by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) under Process Number 2012/23767-2.

References

[1] Baier, C., Katoen, J.P.: Principles of model checking, p. 975. The MIT Press, Cambridge (2008)

[2] Schäfer, T., Knapp, A., Merz, S.: Model checking uml state machines and collaborations. Electronic Notes in Theoretical Computer Science 55(3), 357–369 (2001)

[3] OMG, T.O.M.G.: Omg - unified modeling language (omg uml) (1997)

[4] Debbabi, M., Hassaïne, F., Jarraya, Y., Soeanu, A., Alawneh, L.: Verification and Validation in Systems Engineering, p. 270. Springer, Heidelberg (2010)

[5] Eshuis, R., Wieringa, R.: Tool support for verifying uml activity diagrams. IEEE Transactions on Software Engineering 30(7), 437–447 (2004)

[6] Sarma, M., Mall, R.: Automatic generation of test specifications for coverage of system state transitions. Information and Software Technology 51(2), 418–432 (2009)

[7] Pressman, R.S.: Software Engineering: A Practitioner's Approach, 7th edn. McGrawHill, New York (2010)

[8] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 411–420. ACM Press, New York (1999)

[9] Cockburn, A.: Writing Effective Use Cases, p. 304. Addison-Wesley Professional, US (2000)

[10] Kessler, F.B.: Nusmv home page (2011)

[11] eclipse.org: Papyrus (2014)

[12] Holzmann, G.: The SPIN model checker: Primer and reference manual, vol. 1003. Addison-Wesley (2004)

[13] Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

[14] eclipse.org: Eclipse (2014)

[15] Santiago, V., Mattiello-Francisco, M., Costa, R., Silva, W., Ambrosio, A.: Qsee project: an experience in outsourcing software development for space applications. In: The 19th International Conference on Software Engineering & Knowledge Engineering (SEKE), pp. 51–56 (2007)

[16] Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.: Implementing statecharts in promela/spin. In: Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, pp. 90–101. IEEE (1998)

[17] Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. Formal Aspects of Computing 11(6), 637–664 (1999)

[18] Lam, V.: A formalism for reasoning about uml activity diagrams. Nordic Journal of Computing 14(1), 43–64 (2007)

[19] Dubrovin, J., Junttila, T.: Symbolic model checking of hierarchical uml state machines. In: ACSD 2008. 8th International Conference on Application of Concurrency to System Design, pp. 108–117. IEEE (2008)

[20] Uchitel, S., Kramer, J.: A workbench for synthesising behaviour models from scenarios. In: Proceedings of the 23rd Intern. Conference on Software Engineering, pp. 188–197. IEEE Computer Society (2001)

[21] Baresi, L., Morzenti, A., Motta, A., Rossi, M.: Towards the UML-based formal verification of timed systems. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 267–286. Springer, Heidelberg (2011)

[22] Cortellessa, V., Mirandola, R.: Prima-uml: a performance validation incremental methodology on early uml diagrams. Science of Computer Programming 44(1), 101–129 (2002)

[23] Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Computing Surveys 41(4), 19:1–19:36 (2009)