Modeling and Testing Interruptions in Reactive Systems Using Symbolic Models*

Wilkerson L. Andrade[1], **Patrícia D. L. Machado**[1]

[1] Formal Methods Group/DSC
Federal University of Campina Grande
Campina Grande, Brazil

{wilker,patricia}@dsc.ufcg.edu.br

Abstract. *In the context of reactive systems, several conformance testing strategies have been developed based on variations of labeled transition systems (LTS). However, these models are not suitable when the specification uses large or infinite data domains because each value in the data domain is represented as a system state, leading to an explosion of the state space. This problem is even bigger at interruption testing level, where the possible number of combinations of interruptions at different points of an execution path is huge. Symbolic transition systems (STS) are models where variables and parameters are explicitly represented and these information are treated in a symbolic way. In this work, we present an approach to modeling and testing from STS models of reactive systems with interruptions, and thus make possible the automatic test generation through direct manipulation of high-level specifications avoiding state space enumeration.*

1. Introduction

Testing is one of the most popular validation techniques used today and if it is used in an effective way, it can provide important evidences of quality and reliability of a product. Among several aspects of a system we can test its functionality, performance, time constraints, robustness, and so on, but this paper focuses on conformance testing of reactive systems where interruptions are allowed. In conformance testing, the testing process is performed in order to check if a given implementation is in conformance with its specification. This is based on a conformance relation and the hypothesis that the implementation can be abstracted (but not explicitly constructed) as a formal model so that valid conclusions can be reached on testing results.

In the last years, several theories and techniques of test case generation have been developed through specifications modeled by variations of the classic *Labeled Transition System* (LTS) [Tretmans 1996, Jard and Jéron 2005, Andrade et al. 2007, Cartaxo et al. 2008]. However, LTS models are not suitable when the specification uses large or infinite data domains because each value in the data domain is represented as a system state, leading to the classical state space explosion problem. Consequently, many tools can only be used in very restricted and finite domains.

*Project supported by FAPESQ/CNPq (Project 060/03 and CNPq Proc. 550946/2007-1). Cooperation between Motorola/UFPE/UFCG.

In practice, test cases (written, for example, using the *Testing and Test Control Notation*[1] (TTCN)) can be real programs with parameters and variables. Thus, powerful models are needed, where variables and parameters are explicitly modeled and these information are treated in a symbolic way. Considering this context, there are some approaches whose goal is to provide a symbolic approach for testing reactive systems [Rusu et al. 2000, Clarke et al. 2002].

In order to develop an effective solution to the interruption testing is essential to define a symbolic test model capable of representing such interruptions, and thus make possible the automatic test cases generation through direct manipulation of the high-level specifications without the need of state space enumeration. It is also important to consider that any interruption may be allowed to occur at any time and there are usually infinite possibilities of occurrences.

This paper presents an approach to modeling and testing from STS models of reactive systems with interruptions. This is based on the model proposed by Rusu et al. in [Rusu et al. 2000]. We show how interruptions can be represented and how test cases can be generated and selected based on the use of the STG tool [Clarke et al. 2002]. A case study in presented in the domain of feature testing of mobile phone applications.

This paper is structured as follows. Section 2 gives a quick description of the theory related to the symbolic models. Section 3 presents the symbolic test model structure used to model interruptions. Section 4 introduces the STG tool and discusses how to use it in order to generate interruption test cases. A case study is presented in Section 5. Finally, Section 6 presents related work and Section 7 concluding remarks.

2. Background

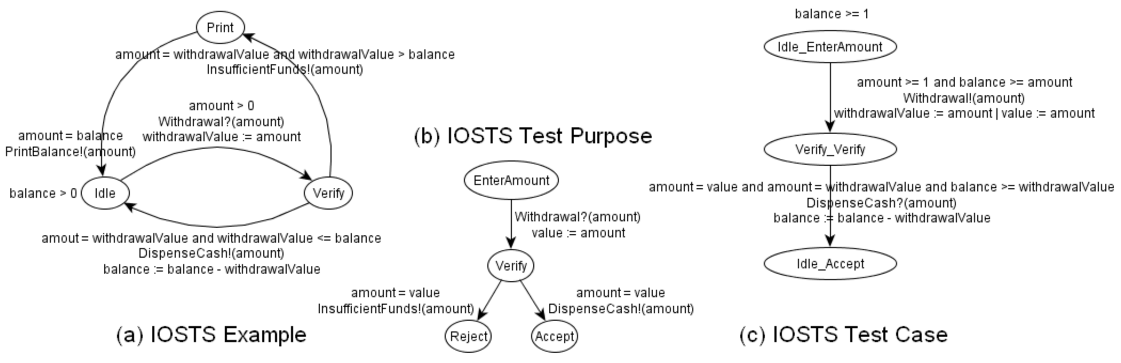
In this section, we present the theory related to the symbolic models. In this work we consider the *Input/Output Symbolic Transition Systems* (IOSTS) that is a model of extended labeled transition systems [Rusu et al. 2000].

Syntax of IOSTS. An IOSTS is a symbolic automata with a finite set of locations, typed variables, and the communication with its environment is performed through actions carrying parameters.

Definition of IOSTS. *Formally, an IOSTS is tuple $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$, where: (1) V is a countable set of typed variables; (2) P is a countable set of parameters. For $x \in V \cup P$, $type(x)$ denotes the type of x; (3) Θ is the initial condition, a predicate with variables in $V \cup P$; (4) L is a countable, non-empty set of locations; (5) $l^0 \in L$ is the initial location; (6) $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is a countable, non-empty alphabet, where $\Sigma^?$ is a countable set of input actions, $\Sigma^!$ is a countable set of output actions, and Σ^τ is a countable set of internal actions. Each action $a \in \Sigma$ has a signature $sig(a) = \langle t_1, ..., t_k \rangle$, that is a tuple of distinct parameters. The signature of internal actions is the empty tuple; (7) $\mathcal{T}$ is a set of transitions, where each transition consists of: a location $l \in L$, called the origin of the transition; an action $a \in \Sigma$, called the action of the transition; a predicate G with variables in $V \cup P \cup sig(a)$, called the guard; an assignment A, such that for each variable $x \in V$ there is exactly one assignment in A, of the form $x := A^x$, where A^x is an expression on $V \cup P \cup sig(a)$; a location $l' \in L$, called the destination of the transition.*

[1] http://www.ttcn-3.org/

Figure 1(a) shows an example of an IOSTS. In graphical representations, input actions are followed by the "?" symbol and output actions are followed by the "!" symbol. These symbols are used only as notation, they are not part of the action's name. The simple IOSTS from Figure 1(a) models a withdrawal transaction in an ATM system, where this transaction has a precondition (the initial condition) that says that the current *balance* must be strictly positive. At the beginning, the system is in the *Idle* location. Next, the system expects the *Withdrawal* input carrying a strictly positive integer parameter *amount* and saves the value of *amount* into the variable *withdrawalValue*. Then, considering that the value of *withdrawalValue* is less than or equal to the *balance*, the ATM system dispenses the cash through the output *DispenseCash* carrying the parameter *amount* (the guard $amount = withdrawalValue$ and $withdrawalValue <= balance$ means "choose a value for the parameter *amount* that, with the value of the variable *withdrawalValue*, satisfies the guard"), the variable balance is decreased by the withdrawn value, and the system returns to the *Idle* location. Otherwise, if the account does not have sufficient funds, the system emits the invalid withdrawal value through the output *InsufficientFunds* carrying the parameter *amount* (the guard $amount = withdrawalValue$ and $withdrawalValue > balance$ has a similar meaning to that of the previous guard), after, the current balance is emitted through the output *PrintBalance* (the guard $amount = balance$ means "choose a value for the parameter *amount* such that it is equal to the value of the parameter *balance*"), and the system returns to the *Idle* location.



Print
amount = withdrawalValue and withdrawalValue > balance
InsufficientFunds!(amount)
amount > 0
Withdrawal?(amount)
withdrawalValue := amount
amount = balance
PrintBalance!(amount)
balance > 0
Idle
Verify
amout = withdrawalValue and withdrawalValue <= balance
DispenseCash!(amount)
balance := balance - withdrawalValue
(a) IOSTS Example
(b) IOSTS Test Purpose
EnterAmount
Withdrawal?(amount)
value := amount
Verify
amount = value
InsufficientFunds!(amount)
amount = value
DispenseCash!(amount)
Reject
Accept
balance >= 1
Idle_EnterAmount
amount >= 1 and balance >= amount
Withdrawal!(amount)
withdrawalValue := amount | value := amount
Verify_Verify
amount = value and amount = withdrawalValue and balance >= withdrawalValue
DispenseCash?(amount)
balance := balance - withdrawalValue
Idle_Accept
(c) IOSTS Test Case

Figure 1. IOSTS Examples

Semantics of IOSTS. The semantics of IOSTS is defined through an *Input-Output Labeled Transition Systems* (IOLTS). An IOLTS is a variant of the classic LTS that makes distinction between events of the system that are controllable by the environment (the inputs) and those that are only observable (the outputs) [Tretmans 1996]. For more details about the semantics of the IOSTS see [Rusu et al. 2000].

Conformance Testing with IOSTS. Conformance testing is a kind of testing used to verify if the implementation is in accordance with the specification. It consists in checking a conformance relation between a specification and an implementation through the execution of a set of test cases and the evaluation of the obtained verdicts [Rusu et al. 2000]. Next, we instantiates the main concepts related to the conformance testing like specifications, implementations, test purposes, test cases, and a conformance relation.

Specifications and Implementations. A specification is an IOSTS without cycles of internal actions and an implementation is any application that can be modeled by an IOSTS.

Test Purposes. Test purposes are used in order to verify if the implementation exhibits a desired behavior. In this context, a test purpose is an IOSTS that describes a specific scenario to be verified. Figure 1(b) presents an example of a test purpose for the ATM withdrawal transaction specification. It is used to select the scenarios where the user successfully performs a withdrawal transaction. The *Reject* location is used to discard all other scenarios where the system does not exhibit the desired behavior. This test purpose is not complete (i.e., all input actions are not enabled all the time) but it is implicity completed by the test case generation tool, so the activity of defining test purpose is simplified by allowing to focus only on the desired behavior.

Test Cases. Test cases are used to assign verdicts to implementations. A test case is an input-complete, deterministic IOSTS with three disjoints sets of locations: *Pass*, *Inconclusive*, and *Fail*. An example of a test case is shown in Figure 1(c). It starts by performing a withdrawal transaction in the ATM system. Then it expects to receive the money. If the ATM system dispenses the expected money the verdict is *Pass*, that is, the implementation is in conformance with the specification and the test purpose. Finally, if some other input is received, the verdict is *Fail*, that is, the implementation emitted an unspecified output.

Conformance. The conformance relation links an implementation to the specification. Intuitively, an implementation conforms to a specification if for all traces of the specification, the set of output actions of the implementation is contained in the set of output actions of the specification.

3. Modeling Features Using Symbolic Models

This section presents the symbolic test model structure capable of representing interruptions. The structure of the proposed test model is illustrated by an application in the mobile phone domain. In this domain, different interruptions (e.g. incoming calls, incoming messages, alarm clocks, and so on) can occur at any time, pausing other features for a moment. In the mobile phone applications context, features are sets of individual requirements that describe a cohesive, identifiable unit of functionality. As a feature example, the Message feature can be cited since it has, amongst other requirements, the requirements of sending and receiving messages.

Figure 2 presents a symbolic model that represents the behavior of removing a message from inbox. The message can only be removed if it is not blocked, otherwise, the message cannot be removed. This behavior is represented only by nodes from 1 to 12. This same model also represents the occurrence of interruptions (nodes from 13 to 16). The Incoming Alert interruption specifies the arrival of a new kind of text messages where the text appears to the user inside a dialog box.

As we can see, in Figure 2, the interruption model is connected to the feature that can be interrupted (the main flow) using a special action named *Interrupt* carrying a parameter (*intCode*) that identifies the place where the interruption is allowed. Then, the value of the parameter *intCode* is saved in the variable *choice*. Each point where an interruption is allowed has a different integer value associated to it. Other important information is in the last action of the interruption, there is a guard used to guarantee that the main flow continues its execution from the same point where it had been interrupted. For instance, if an interruption begins with the parameter *intCode* equals to 1, then it must

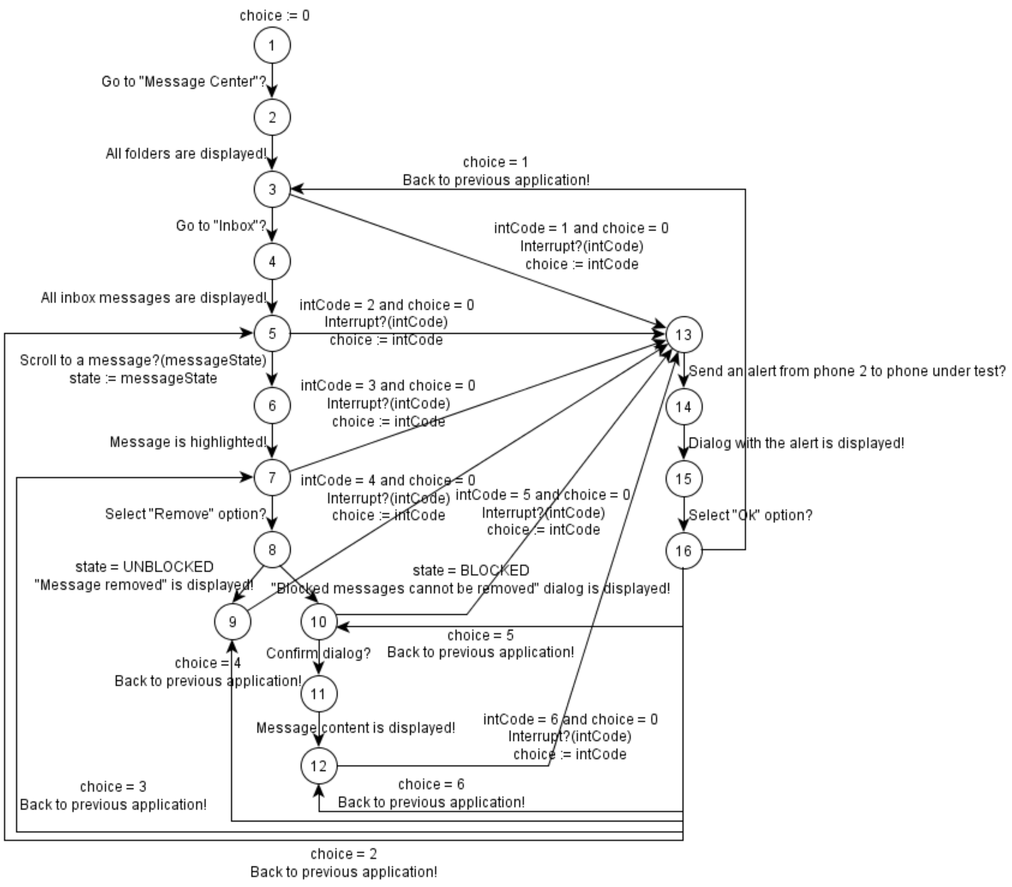

choice := 0
1
Go to "Message Center"?
2
All folders are displayed!
choice = 1
Back to previous application!
3
Go to "Inbox"?
intCode = 1 and choice = 0
Interrupt?(intCode)
choice := intCode
4
All inbox messages are displayed!
intCode = 2 and choice = 0
Interrupt?(intCode)
choice := intCode
5
13
Scroll to a message?(messageState)
state := messageState
Send an alert from phone 2 to phone under test?
intCode = 3 and choice = 0
Interrupt?(intCode)
choice := intCode
6
14
Message is highlighted!
Dialog with the alert is displayed!
7
15
intCode = 4 and choice = 0
Interrupt?(intCode)
choice := intCode
intCode = 5 and choice = 0
Interrupt?(intCode)
choice := intCode
Select "Remove" option?
Select "Ok" option?
8
16
state = UNBLOCKED
"Message removed" is displayed!
state = BLOCKED
"Blocked messages cannot be removed" dialog is displayed!
9
10
choice = 5
Back to previous application!
choice = 4
Back to previous application!
Confirm dialog?
11
Message content is displayed!
intCode = 6 and choice = 0
Interrupt?(intCode)
choice := intCode
12
choice = 3
Back to previous application!
choice = 6
Back to previous application!
choice = 2
Back to previous application!

Figure 2. Remove Message Behavior with Interruptions

finish performing the action that has the following guard: *choice = 1*.

Some hints of how to link a model with an interruption are: (1) identify the point where the interruption must occur; (2) link this point to the interruption behavior using a transition labeled as follows: the guard is *intCode = X and choice = 0*, where *X* uniquely identifies this point of interruption; the action is *Interrupt?(intCode)*; the assignment is *choice := intCode*; (3) connect the last action of the interruption behavior to the same point where the interruption started using a transition labeled with the guard *choice = X*, where *X* is the same value that uniquely identifies this point of interruption.

Considering the interruption testing in the mobile phone applications context, the more interesting test cases are those where there is only one interruption for each test case. This happens for four reasons: (1) failures are rarely the result of the simultaneous occurrence of two (or more) faults; (2) the majority of the test cases are manually executed; (3) there is usually only one common resource, for instance, the processor, the screen, etc; (4) good test cases must facilitate the fault localization. Considering that each failure is associated with only one fault, it is enough that each test case verifies the interruptions in a single point at a time, thus making fault localization easier.

The test case generation strategy where only one interruption is allowed for each test case is reached because of the second part of the guard (*choice = 0*) associated to the *Interrupt* actions. When an interruption is allowed, the value of the variable *choice* is changed to any value different from zero, then all other interruptions are automatically discarded during the test case generation.

Considering a continuous time, an interruption can occur at infinite points during the system execution. But considering the tester's point of view, each possibility of interruption can only be observed after each system response. This happens due to the fact that it is impractical to reproduce a scenario where an interruption occurs between an input action and the system response. Thus, the proposed test model only represents those interruptions that occur immediately after the system responses. In this case, Figure 2 represents all possibilities of interruption from tester's point of view.

In practice, this interruption test model should not be written by hand because it is tiresome and not cost-effective. It must be generated directly from abstract specifications. As default, any interruption can occur virtually at any point of execution. However, depending on the intended behavior at some points some of them or all may not be allowed. By taking these issues into account, Figueiredo et al [de Figueiredo et al. 2006] presents an approach for specifying feature and interruptions as use cases. The model presented in this section can be automatically generated from this approach.

4. Test Case Generation and Selection with STG

STG (Symbolic Test Generation) [Clarke et al. 2002] is a tool which allows for the automatic synthesis of conformance test cases from symbolic models. It is based on that theory presented in Section 2 and it is a potential candidate for interruption test case generation. In this section, STG will be presented in order to show how to use it for interruption test case generation and selection.

An exhaustive interruption test case generation is impractical due to the huge amount of generated test cases. Particularly, in mobile phone applications context the majority of test cases are manually executed. In this scenario, test case selection strategies are needed. The strategy used to reduce the test suite is based on test purposes.

Then, once the system is specified using IOSTS, the next step is to define test purposes in order to verify a particular system functionality or a specific interruption at some point. For our proposed test model, it is very simple to define test purposes where an interruption can occur. Given that the point where the interruption must occur was chosen, it is enough to use the *Interrupt* action in the test purpose carrying the integer that identifies the selected point.

Considering that specification from Figure 2, a test purpose can be defined in order to verify the scenario where an alert appears when the user is accessing the inbox folder. As the selected interruption point corresponds to the second output of the specification, the action *Interrupt* must carry the integer 2. Next, the last action of the selected behavior (the scenario where the message is removed) is appended to the test purpose. In the mobile phone context, test cases are paths from the initial node to any leaf node or, in case of cycles, the longest paths with non-repeated actions. Therefore, it is important to keep the last action of the chosen scenario in the test purpose, so the generated test case contains all steps of the selected path. Figure 3(a) presents the test purpose defined above and the Figure 3(b) shows the test case generated by the STG tool. Note that in the generated test case (Figure 3(b)) the interruption occurs only in one specific point.

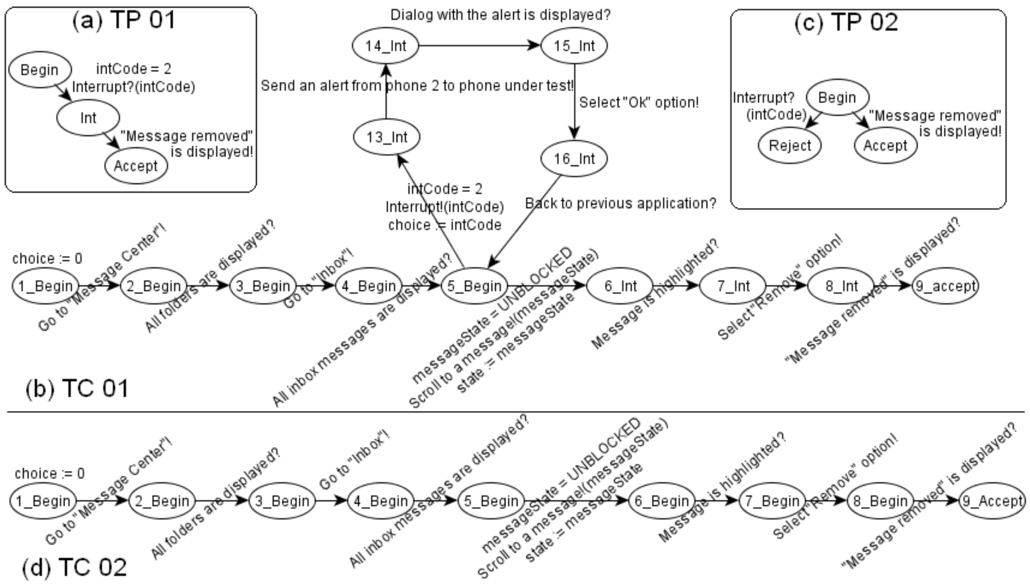

(a) TP 01
Begin
intCode = 2
Interrupt?(intCode)
Int
"Message removed" is displayed!
Accept
(b) TC 01
(c) TP 02
Interrupt? (intCode)
Reject
"Message removed" is displayed!
(d) TC 02
Dialog with the alert is displayed?
Send an alert from phone 2 to phone under test!
Select "Ok" option!
Back to previous application?
choice := 0
Go to "Message Center"!
All folders are displayed?
Go to "Inbox"!
All inbox messages are displayed?
messageState = UNBLOCKED
Scroll to a message!(messageState)
state := messageState
Message is highlighted?
Select "Remove" option!
"Message removed" is displayed?

Figure 3. Test Purposes and Test Cases

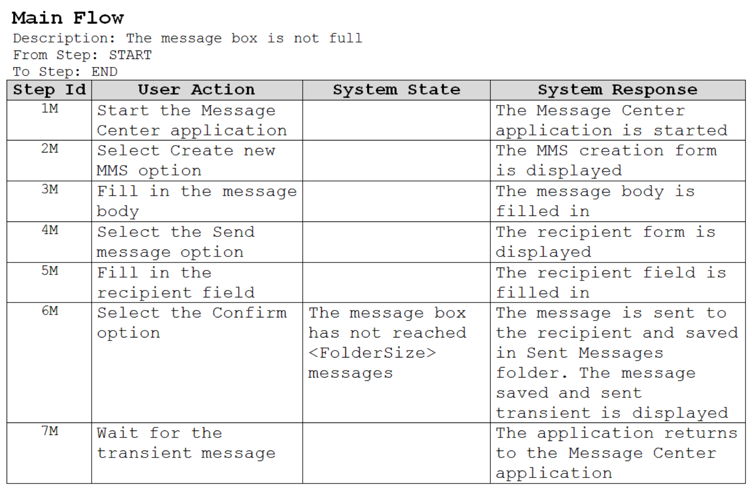
Considering that specification from Figure 2, another test purpose can be defined in order to verify a scenario where a message is removed and all interruptions are not allowed. This test purpose is presented in Figure 3(c). Note that to prohibit all interruptions it is enough to take the *Interrupt* action to the *Reject* location. The other action of the test purpose (*"Message removed" is displayed!*) is used to select the scenario where the message is removed. The generated test case is shown in Figure 3(d).

5. Case Study

This section presents a case study to illustrate the application of the proposed test model for interruption testing. The main goal is to compare the use of IOLTS, a variation of the classic labeled transition system, with the use of IOSTS, a kind of symbolic model, to represent interruptions, and consequently allow the automatic generation of test cases. This case study was performed using an application of the mobile phone domain.

The chosen features are Multimedia Messages and Incoming Alert. The former feature means a new kind of message capable of containing multimedia items like image, audio, and video. For this case study only one use case of the Multimedia Messages feature was considered: the action of creating and sending a multimedia message. The last feature was chosen because it is responsible for causing interruptions.

Basically, features and interruptions are specified as use cases using a controlled natural language [de Figueiredo et al. 2006]. In this kind of specification, each use case has a main flow and some alternative flows. Flows are described through steps including a user action and the respective system response. Each step has a condition (system state) that determines if the system response will happen or not. Due to the lack of space, Figure 4 shows only the main flow of the Multimedia Messages specification. Note that the specification can contain variables to make it more abstract and reusable, for example the message box size (<*FolderSize*> variable) was not explicitly specified because this value depends on the phone memory size. In this case study we assume that there are three possible values for <*FolderSize*>: 1000, 1500, and 2000.



Main Flow

Description: The message box is not full

From Step: START

To Step: END

Step Id	User Action	System State	System Response
1M	Start the Message Center application		The Message Center application is started
2M	Select Create new MMS option		The MMS creation form is displayed
3M	Fill in the message body		The message body is filled in
4M	Select the Send message option		The recipient form is displayed
5M	Fill in the recipient field		The recipient field is filled in
6M	Select the Confirm option	The message box has not reached <FolderSize> messages	The message is sent to the recipient and saved in Sent Messages folder. The message saved and sent transient is displayed
7M	Wait for the transient message		The application returns to the Message Center application

Figure 4. *Multimedia Messages* Feature

The first step of the case study was to represent the specification of the Multimedia Messages and Incoming Alert features using both models IOLTS and IOSTS. In the first model (IOLTS) we do not have variables then we must explicitly represent all values of the variables leading to a bigger model than the other (IOSTS). Due to the lack of space, the models will not be shown. The developed IOLTS had 84 states and 100 edges while the IOSTS model had only 26 states and 47 edges. Symbolic models are usually simpler than labeled transition systems.

One of the main advantages of using symbolic models is in the test purpose definition activity. A test purpose was defined using both models in order to verify a scenario where an interruption occurs when the user is typing a message, after that, the user decides to include an image into the message, and finally, the message is sent and saved into the message box. The defined IOLTS test purpose had 20 states and 27 edges while the defined IOSTS test purpose had only 5 states and 4 edges.

The IOSTS test purposes are usually very simple leading to a less error-prone activity. Moreover, symbolic models allow the generation of abstract test cases, that is, test cases where the variables are instantiated only during the test execution. Abstract test cases are ideal in the mobile phone context mainly because the development process follows the strategy of software product lines. In this case, test cases can be generated in an abstract way and they are instantiated depending on the kind of product under test.

6. Related Work

To the best of our knowledge, approaches developed in order to take into account symbolic models as the underlying model for interruption test case generation are practically nonexistent. So this section presents some approaches that could be followed in order to provide an effective solution to the interruption testing. In the context of symbolic models, the principle is to adapt some existing approach to, beyond representing the system behavior, represent the system data without enumerate the data values. There is still very few work in this context and the most of them use (variations of) state machines or labeled transition systems as the underlying model.

State machines tend to be used in synchronous context, where inputs and outputs appear together in a single transition. Thus, they are unsuitable for representing some characteristics of reactive systems like interruptions, isolated input and outputs, etc. Considering the use of state machines, there is a tool named GAST [Koopman and Plasmeijer 2003] that was extended to deal with Extended Finite State Machines (EFSM) specifications. In this tool, properties and data types are expressed in first order logic, and based on this information, test data are automatically generated. This tool is less suitable for testing because the concept of state is not present in a clear manner, even thus it is possible to represent states defining explicitly a complex data structure that represents the state space. The GAST's algorithm unfolds, in an on-the-fly way, the data type structure in order to select a path in the EFSM.

The use of (variations of) labeled transition systems is more common in the literature. Lestiennes and Gaudel [Lestiennes and Gaudel 2002] developed a strategy of test generation and selection based on selection hypotheses combined with an operation of unfolding algebraic data types and predicate resolution. Frantzen et al. [Frantzen et al. 2006] extended the ioco theory presented in [Tretmans 1996] to support the software testing based on symbolic models. The symbolic framework developed by Frantzen et al. uses concepts from first order logic as underlying theory for dealing with guards and variables, quiescence is taken into account, and some ideas about coverage is discussed. As disadvantages, the symbolic framework does not consider test purposes and it still need to be automated. Calamé et al. [Calamé et al. 2007] proposes an approach combining a model similar to IOSTS, data abstraction, and constraint solving to generate test cases. The main idea is to apply data abstraction to abstract the model in a finite state one, use the TGV tool [Jard and Jéron 2005] to generate abstract test cases, and finally, constraint solving is applied to instantiate the test cases.

The chosen approach to develop an effective solution to interruption testing is based on a symbolic test case generation theory (introduced in Section 2). This theory avoids the state explosion problem due to the enumeration of data values. The test case generation process is reduced to syntactic operations on the models and an analysis of which states take the system to acceptance states. Next, abstract test cases are automatically generated by the STG tool, and finally, during the test execution, test cases are instantiated using a constraint solving.

7. Concluding Remarks

This work proposes a symbolic test model capable of representing interruptions for reactive systems. This model is based on the *Input/Output Symbolic Transition Systems* theory. The IOSTS theory is supported by the STG tool [Clarke et al. 2002]. Since that test selection is a crucial requirement for interruption testing, some hints of how to define test purposes in order to verify specific interruptions were presented. In practice, given the specification (UML models, use case templates, and so on), a symbolic test model must be automatically generated. Finally, the symbolic model must be combined with test purposes for the interruption test case generation.

The use of symbolic models allows the generation of abstract test cases. In the mobile phone applications context, abstract test cases are very suitable because the development process follows the strategy of software product lines. Thus, test cases can be generated in an abstract way and they are instantiated depending on the kind of product under test, making the test case management process easier. Furthermore, as the presented strategy is based on an IOSTS test model, different abstract specification formalisms can be used. As further work, this model is going to be extended to include timing requirements for extending its application to real-time systems.

References

Andrade, W. L., Neto, F. G. O., and Machado, P. D. L. (2007). Geração de casos de teste de interrupção para aplicações de celulares. In *VIII Test and Fault Tolerance Workshop (WTF 2007)*, pages 129–142, Porto Alegre, RS, Brazil.

Calamé, J. R., Ioustinova, N., and van de Pol, J. (2007). Automatic model-based generation of parameterized test cases using data abstraction. In *Proceedings of the Doctoral Symposium affiliated with the Fifth Integrated Formal Methods Conference (IFM 2005)*, volume 191 of *ENCS*, pages 25–48. Elsevier.

Cartaxo, E. G., Andrade, W. L., Neto, F. G. O., and Machado, P. D. L. (2008). LTSBT: A tool to generate and select functional test cases for embedded systems. In *SAC'08: Proceedings of the 2008 ACM symposium on Applied computing*, volume 2, pages 1540–1544, New York, NY, USA. ACM Press.

Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2002). STG: A symbolic test generation tool. In *TACAS'02: Proc. of the Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 151–173. Springer.

de Figueiredo, A. L. L., Andrade, W. L., and Machado, P. D. L. (2006). Generating interaction test cases for mobile phone systems from use case specifications. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10. Proceedings of the AMOST'2006.

Frantzen, L., Tretmans, J., and Willemse, T. (2006). A Symbolic Framework for Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in LNCS, pages 40–54. Springer.

Jard, C. and Jéron, T. (2005). TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315.

Koopman, P. and Plasmeijer, R. (2003). Testing reactive systems with GAST. In Gilmore, S., editor, *Trends in Functional Programming*, volume 4 of *Trends in Functional Programming*, pages 111–129. Intellect.

Lestiennes, G. and Gaudel, M.-C. (2002). Testing processes from formal specifications with inputs, outputs and data types. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 3, Washington, DC, USA. IEEE Computer Society.

Rusu, V., du Bousquet, L., and Jéron, T. (2000). An approach to symbolic test generation. In *IFM'00: Proceedings of the Second International Conference on Integrated Formal Methods*, pages 338–357, London, UK. Springer-Verlag.

Tretmans, J. (1996). Test generation with inputs, outputs, and quiescence. In *TACAS'96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 127–146, London, UK. Springer-Verlag.