Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 184 (2007) 3–19
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Towards Property Oriented Testing

Patricia D. L. Machado[1,3]

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

Daniel A. Silva[2,4]

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

Alexandre C. Mota[5]

Formal Methods Laboratory/CIn
Federal University of Pernambuco
Recife, Brazil

Abstract

Conformance testing is a kind of functional testing where a formally verified specification is considered and test cases are generated so that conclusions can be established regarding the possibility of acceptance/rejection of conforming/non-conforming implementations. If the focus is on a complete specification, test suites may be impractical and even infinite with unclear relations between test cases and the specification. Property oriented testing focuses on particular, eventually critical, properties of interest. The specification of one or more properties drives the test process that checks whether they are satisfied by an implementation. Properties are often stated as test purposes, targeting testing at a particular functionality. This paper presents an overview of approaches to property oriented testing for reactive systems, focusing on labelled and symbolic transition systems.

Keywords: Property testing, test purpose, transition systems

1 Introduction

Testing from formal specifications has proved to be a convenient association between testing and formal methods. Test cases are derived from a formal specifica-

[1] Supported by FAPESQ/CNPq/Brazil - Project 060/03.
[2] Supported by CAPES/Brazil.
[3] patricia@dsc.ufcg.edu.br
[4] daguiar@dsc.ufcg.edu.br
[5] acm@cin.ufpe.br

doi:10.1016/j.entcs.2007.06.001

tion and an implementation is checked, by means of executing these test cases, to detect whether the implementation satisfies the specification. As a kind of functional (black-box) testing, this is usually called specification-based testing. However, in the formal methods community, conformance testing is more often used, where a formally verified specification is considered and test cases are generated so that important properties can be established regarding the possibility of acceptance/rejection of conforming/non-conforming implementations [15]. Furthermore, when the specification is given as a model, the term model-based testing is used [7].

Since they are based only on the observable behaviour (black-box view) of a system, conformance test cases are independent of a specific implementation. Thus they can be used in different versions of a given implementation. Moreover, they can be developed as soon as a specification is produced, either prior to or in parallel with the implementation. It is largely accepted that conformance testing performed in parallel with development helps to improve both the specification and the code produced and usually only a few faults are detected at test execution [22,21]. The resulting test suite is a valuable asset to cope with changes in further versions and a strong link between specification and code – usually not suitable to formal reasoning.

However, conformance testing presents some limitations. Depending on the coverage criteria, for instance, overall specification coverage, test suites may be too big and therefore impractical to handle and evolve with the implementation, even if automation is considered. This is particularly critical for distributed systems which are often complex and difficult to develop and test. Test generation algorithms can produce large test suites, even infinite, with unclear relations between a given test case and the specification. The purpose of test cases should be clear [6]. Also, they may present significant redundancies when a number of test cases have the same effect in detecting particular faults [17]. Testing is a very time-consuming and expensive activity, demanding optimal and minimal test cases to be selected. Furthermore, not all the specification is often implemented and the implementation may present additional behaviour.

Property oriented testing is a kind of conformance testing where the selection criteria is associated with a given property, verified on the specification, that needs to be checked. The goal is to focus on particular, eventually critical, properties of interest of the system, possibly not previously checked [10,8]. The specification of one or more properties drives the test process that checks whether they are satisfied by an implementation. Properties are often stated as a test purpose, targeting testing at a particular functionality.

Approaches for test case selection from test purposes have been developed, focusing on algorithms and tools, mostly based on labelled transition systems as specifications [6,16]. More recently, approaches based on more abstract views of a system are being considered [4,11]. In both cases, the approaches are usually inspired by the model checking technique, where a given (temporal logic) property is verified over a model. Model checking algorithms are adapted to test case generation and temporal logic properties are the basis of test purposes [8,28].

This paper presents an overview of approaches to property oriented testing for

reactive systems where properties are stated as test purposes. There is no intention to be exhaustive, but to present different and representative ones. Section 2 presents theoretical background on conformance testing and test purpose. Section 3 presents approaches for test purpose and test case selection based on labelled transition systems, whereas Section 4 presents approaches based on symbolic transition systems. Finally, Section 5 presents a comparison of approaches and Section 6 presents concluding remarks and perspectives.

2 Background

The next subsections give a quick description of the formal framework for conformance testing presented in [15,30] and the formal framework for test purposes presented in [6].

2.1 Formal Framework for Testing

Conformance testing relates a specification and an implementation under test (IUT) by the relation **conforms-to** $\subseteq IMPS \times SPECS$, where *IMPS* represents the universe of implementations and *SPECS* represents specifications.

Then, IUT **conforms-to** s if and only if IUT is a correct implementation of s.

However, such a relation is hard to be checked by testing. Also, implementations are usually unsuitable for formal reasoning. Therefore, we assume that any IUT can be modelled by a formal object $i_{\mathrm{IUT}} \in MODS$, where *MODS* represents the universe of models. This is known as test hypothesis [1]. Then, an implementation relation **imp** $\subseteq MODS \times SPECS$ is defined such that IUT **conforms-to** s if and only if i_{IUT} **imp** s.

Let *TESTS* be the domain of test cases and $t \in TESTS$ be a test case. Then EXEC(t,IUT) denotes the operational procedure of applying t to IUT (test execution). The result of this procedure is a set of observations in the domain *OBS*. It can be noticed that test execution may involve multiple runs of EXEC(t,IUT) due to non-determinism. Let an observation function that formally models EXEC(t,IUT) be defined as $obs : TESTS \times MODS \rightarrow \mathcal{P}(OBS)$. Then, $\forall$ IUT $\in IMPS$ $\exists i_{\mathrm{IUT}} \in MODS$ $\forall t \in TESTS \cdot$ EXEC(t,IUT) $= obs(t, i_{\mathrm{IUT}})$, according to the test hypothesis.

Let $v_t : \mathcal{P}(OBS) \rightarrow \{\mathbf{fail}, \mathbf{pass}\}$ be a family of verdict functions. This can be abbreviated to IUT **passes** $t \Longleftrightarrow_{def} v_t(\mathrm{EXEC}(t,\mathrm{IUT})) = \mathbf{pass}$. Then, for any test suite $T \subseteq TESTS$, IUT **passes** $T \Leftrightarrow \forall t \in T \cdot$ IUT **passes** t. Also, IUT **fails** $T \Leftrightarrow \neg$(IUT **passes** T). A test suite that can distinguish between all conforming and non-conforming implementations is called *complete*. Let $T_s \subseteq TESTS$ be complete. Then, IUT **conforms-to** s if and only if IUT **passes** T_s.

Since complete test suites are impractical, weaker requirements are needed. A test suite is *sound* when all correct implementations and possibly some incorrect implementations pass it. In other words, any detected faulty implementation is non-conforming, but not the other way around. Let $T \subseteq TESTS$ be sound. Then, IUT **conforms-to** $s \Rightarrow$ IUT **passes** T. The other direction of the implication

is called *exhaustiveness*, meaning that all non-conforming implementations will be detected.

Sound test suites are more commonly accepted in practice, since rejection of conforming implementations, by exhaustive test suites, may lead to unnecessary debugging. Let $der_{imp} : SPECS \rightarrow \mathcal{P}(TESTS)$ be a test suite derivation algorithm. Then, $der_{imp}(s)$ should only produce sound and/or complete test suites.

2.2 Formal Test Purposes

Testing for conformance and testing from test purposes have different goals. The former aims to accept/reject a given implementation. On the other hand, the latter aims to observe a desired behaviour that is not necessarily directly related to a required behaviour or correctness. If a behaviour is observed, then confidence on correctness may increase. Otherwise, no definite conclusion can be reached. Due to its overloaded use, test purpose is called observation objective in [6]. Nevertheless, the term test purpose is kept in this paper for the sake of presentation. This subsection extends concepts introduced in Subsection 2.1 for test purposes.

Test purposes describe desired observations that we wish to observe by testing an implementation. Test purposes are related to implementations that are able to exhibit them by a well chosen set of experiments. This is defined by the relation **exhibits** $\subseteq IMPS \times TOBS$, where $TOBS$ is the universe of test purposes. To reason about exhibition, we also need to consider the test hypothesis from Subsection 2.1 by defining the *reveal* relation **rev** $\subseteq MODS \times TOBS$, so that, for $e \in TOBS$, IUT **exhibits** e if and only if i_{IUT} **rev** e, with $i_{\text{IUT}} \in MODS$ of IUT.

A verdict function decides whether a test purpose is exhibited by an implementation: $H_e : \mathcal{P}(OBS) \rightarrow \{\textbf{hit}, \textbf{miss}\}$. Then, IUT **hits** e **by** $t_e =_{def} H_e(\text{EXEC}(t_e,\text{IUT})) =$ **hit**. This is extended to a test suite T_e as IUT **hits** e **by** $T_e =_{def} H_e(\bigcup\{\text{EXEC}(t,\text{IUT}) \mid t \in T_e\}) =$ **hit**, which differs from the **passes** abbreviation.

A test suite that is *e-complete* can distinguish among all exhibiting and non-exhibiting implementations, such that, IUT **exhibits** e if and only if IUT **hits** e **by** T_e. A test suite is *e-exhaustive* when it can only detect non-exhibiting implementations (IUT **exhibits** e implies IUT **hits** e **by** T_e), whereas a test suite is *e-sound* when it can only detect exhibiting implementations (IUT **exhibits** e if IUT **hits** e **by** T_e). Note that there is a similarity in purpose between *sound* test suites and *e-sound* test suites, even though the implications are relatively inverted. The former can reveal the presence of faults, whereas the latter can reveal intended behaviour.

Conformance and exhibition can be related. The goal is to consider test purposes in test selection to obtain test suites that are sound and e-complete. On one hand, *e-soundness* guaranties that a hit result always implies exhibition. On the other hand, *e-exhaustiveness* guaranties that implementations that exhibit are not rejected. Soundness provides us with the ability to detect non-conforming implementations. Contrary to complete test sets, *e-complete* test sets are more feasible. For instance, an algorithm is present in [6] for labelled transition systems.

Finally, it is important to remark that a test purpose may be revealed by both conforming and non-conforming implementations w.r.t. a given specification. An

ideal situation, though not practical, would be to only consider a test purpose e when i **rev** $e \supseteq i$ **passes** T, where $T \subseteq TESTS$. However, test purposes are chosen so that: $\{i \mid i \textbf{ rev } e\} \cap \{i \mid i \textbf{ imp } s\} \neq \emptyset$. In this case, a test execution with test case $T_{s,e}$ that is both sound and e-complete and that results in **fail** means non-conformity, since sound test cases do not reject conforming implementations and e-complete test cases distinguish between all exhibiting and non-exhibiting implementations. Also, if the result is $\{\textbf{pass}, \textbf{hit}\}$, confidence on correctness is increased, as the hit provides possible evidence of conformance.

3 Test Generation from Labelled Transition Systems

Labelled Transition Systems (LTS) are defined in terms of states and labelled transitions between states, where labels often represent observable interactions of the system. A test case is usually defined as an LTS that is deterministic and has finite behaviour, contains **pass** and **fail** as terminal states and any non-terminal state has either an input action or output actions or an action that observes quiescence. Such restrictions are often made to make the definition of sound and e-complete test cases possible.

LTS approaches to property testing are presented in the next subsections. These approaches are based on the specification of the visible behaviours of reactive systems. Specification consists of an LTS representing the IUT behaviour, in general, through input and output actions. Analysis over specifications, e.g. performed by traversal techniques, make possible the specification of the properties to be tested against the IUT and the establishment of the conformance relation w.r.t. the IUT.

3.1 Test Generation with Verification Technology

TGV (Test Generation with Verification technology) [16] is a black-box testing tool that provides automatic synthesis of conformance test cases for non-deterministic reactive systems. It has already been applied to industrial experiments [9]. The test cases synthesis is based on verification techniques such as synchronous product, on-the-fly verification and traversal algorithms.

TGV is based on the ioco theory presented in [29]. This theory defines a conformance relation between implementation and specification. It states that an IUT is ioco-correct with respect to its specification if either (*i*) the IUT can never produce an output which could not have been produced by the specification in the same situation (i.e. after the same sequence of inputs and outputs), or (*ii*) the IUT may only be quiescent if the specification can do so. Quiescence is defined as the absence of outputs, and it may be observed by timers.

As input, TGV receives the model of the system behaviour and a test purpose, both given as a variant of LTS called Input-Output Labelled Transition System (IOLTS). The IOLTS provides distinction between events of the system that are controllable by the environment and those that are only observable by the environment, i.e., inputs and outputs, respectively. Internal actions are also defined. The test purposes are equipped with two additional special states, the *accept* and *refuse*

states, which are used to define the targeted behaviours.

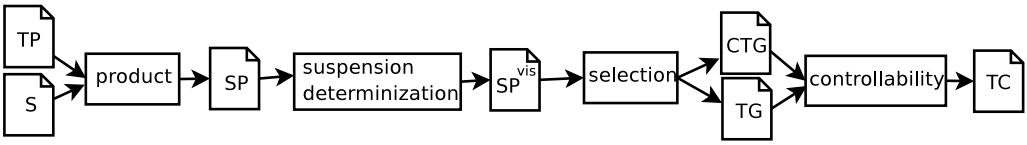

Fig. 1. TGV infrastructure [16].

From Figure 1, the first step is to perform a synchronous product between the specification S of the system and the test purpose TP, resulting in a third IOLTS SP. This solves the problem of identifying the behaviours of S accepted or refused by TP. As the ioco relation can be established based on observable behaviours, TGV defines another IOLTS which contains only observable actions of the system and provides a way to explicit define the quiescent states. This IOLTS is called $\mathsf{SP}^{\mathsf{vis}}$. After that, the test selection builds another IOLTS, called CTG, representing the complete test graph referred to the test purpose. The CTG is built through extraction of the accepted behaviours (identified from the *accept* states) and inversion of inputs and outputs, to provide the link between the IUT and the test cases.

Concluding the process, the so-called controllability conflicts are suppressed to obtain the test cases. These controllability conflicts are represented by the presence of choice between outputs or between inputs and outputs in some states. These conflicts are solved by pruning some transitions. In this case, either one output is kept and all other inputs and outputs are pruned, or all inputs are kept and outputs are pruned. Optionally, a test graph TG may be constructed during the selection phase by suppressing some conflicts on-the-fly.

The abstract test cases generated by TGV are sound and exhaustive with respect to the ioco relation. However, since the generated CTG may contain infinite test cases and the tool does not provide a mechanism to select the tests to be implemented, it is difficult to achieve soundness and exhaustiveness of test suites in practice. The tool does not provide mechanisms to implement and execute the test cases either. However, TGV is part of a wide project, called AGEDIS [13], which aims at the development of a methodology and tools for automated model driven test generation and execution for distributed systems.

3.2 Generation of MSC Based Test Purposes For Testing Distributed Systems

The generation of test purposes is treated in [14]. An algorithm for the generation of test purpose descriptions in the form of Message Sequence Charts (MSCs) from labelled event structures (LESs) is presented. The LES structure is aimed at representing the behaviour of a system of asynchronously communicating extended finite state machines, which is composed of a set of extended finite state machines (EFSM) and a set of FIFO queues connecting the EFSM's with each other and with the environment. The communication between the EFSM's through the queues is considered to be reliable, i.e. without loss or reordering of messages.

The EFSM model considered may contain variables that are used for representing message parameters, for buffering values, or calculations to be carried out during

the execution of transitions. However, it does not contain enabling conditions for the occurrence of events. The communication of the EFSM's with the environment is done at points of control and observation (PCO's), i.e. points in which the environment can interact with the system sending inputs and receiving outputs from it. The environment can only put a message into a queue if the associated EFSM is ready to consume it.

LES is the model used to describe the test purposes. It is defined over a set of events, to which actions are assigned. An action can occur various times in a system run, each time forming a new distinguishable event. The actions model inputs and outputs of the system, calculations in the context variables, and setting, resetting and expiration of timers of the EFSM's. Formally, a LES is a tuple $\langle E, \preceq, \#, l \rangle$, where: E is a finite set of events; $\preceq \subseteq E \times E$ is a partial order relation in E, called causality relation, such that for all $e \in E$ the set $\{e' \in E \mid e' \preceq e\}$ is finite; $\# \subseteq E \times E$ is an irreflexive and symmetric relation in E, called conflict relation; and $l : E \to A$ is a labelling function assigning an action to each event.

This approach generates only test purposes from complete prefixes of the LES of the communicating EFSM's. A prefix of the LES $\langle E, \preceq, \#, l \rangle$ is a LES $\langle E', \preceq', \#', l' \rangle$ that is induced by a causally closed subset of events $E' \preceq E$. A set of actions E' is causally closed if $\forall e' \in E' \; \forall e \in E \cdot (e \preceq e' \Rightarrow e \in E')$. A prefix of the LES of a system of asynchronously communicating EFSM's is complete if it contains a configuration for each reachable global state of the system. Thus, for each reachable global state of the system, consisting of the states of the EFSM's and the contents of the queues, there must exist a configuration C, such that, only events $e \in C$ have occurred, not any other. A set of events C is a configuration if it is causally closed and conflict free, i.e. $\forall e, e' \in C(\neg(e \# e'))$.

The generation of the test purposes starts from a complete prefix of the LES constructed from a system of asynchronously communicating EFSM's (Figure 2). It is based on the identification of the significant behaviours of the system, which are represented by the maximal configurations of the complete prefix. A maximal configuration is a configuration to which no more events of the complete prefix of the LES can be added.

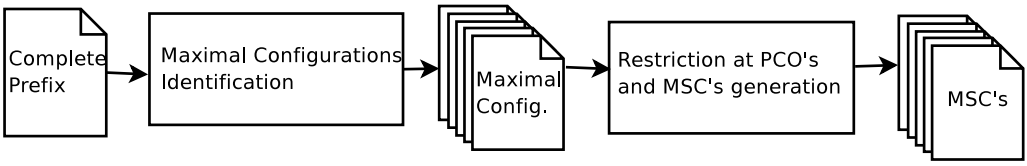

Fig. 2. MSC's Generation.

After the identification of all maximal configurations of the complete prefix, the next step consists of restricting the maximal configurations to events occurring at the PCO's. This is done by the identification and omission of the events for which the remote communication partner is not the environment. After the identification of the maximal configurations, a MSC description is generated to represent each one.

A prototype tool [19] implements the proposed algorithm. The MSC's can be

formatted to be applicable as input to the Autolink tool [27]. For each MSC a test case must be generated. A MSC aims at the generation of a test case for each maximal configuration of the complete prefix. Based on this approach, covering each maximal configuration of the complete prefix, the approach achieves all-nodes coverage of the complete prefix. However, this approach produces a small number of test cases, making difficult to reach exhaustiveness. Covering more than cut-off points of the complete prefix, producing larger test cases, could improve the soundness of the test cases, providing higher degree of confidence on the IUT. This soundness requirement is weaker than the presented in Section 2.1. It is based on the fact that larger executions produce higher test coverage. The test purposes generated do not provide any further verdict informations for the generation of test cases. The verdicts must be included in the test cases during its generation.

3.3 Property Oriented Test Case Generation

An approach to automatically produce test cases allowing to check the satisfiability of a linear property on a given implementation is discussed in [8]. The authors present formal definitions and a sketch of a test generation algorithm.

Linear properties are commonly modelled by Linear Temporal Logic (LTL) formulas [5] to formally verify concurrent reactive systems. As such kind of systems are characterised by no execution termination, various specification formalisms based on finite state automata have been proposed to recognise the infinite sequences of events, defined by an LTL formula. To reason about testing temporal properties, the authors propose the use of a deterministic automaton, called observer, defined by the Rabin automaton [24] formalism to express the properties and recognise execution sequences.

The temporal properties are usually classified into safety or liveness [18]. The liveness properties can usually be checked by considering infinite sequences, while safety properties can be checked by considering finite sequences. To test the satisfiability of a liveness property, the notion of bounded liveness properties is introduced. In a bounded property, the number of execution sequences to consider in the recognition process is limited to a specified length defined in number of interactions or in execution time. The automaton on infinite words can be parameterised to specify such properties.

IOLTS is the formalism used as the basis of the considered models. The specification needs not to be exhaustive with relation to the system behaviour, i.e. the specification can be a partial specification of system behaviour. The authors argue that the ioco based testing conformance theory [29] is limited, since any IUT unspecified behaviour by the specification would be rejected. The conformance relation is not stated in terms of the specification. This is done through a satisfiability relation defined between the implementation and the property. Therefore, this approach states that an IUT will satisfy a given linear property P if and only if all of its execution sequences belong to the model of P. Based on this, the test cases are expressed in terms of the parameterised Rabin automaton.

The test generation algorithm proposed is based on two steps: generation of a

test graph (TG) and test case selection from it (Figure 3). TG originates from a set of execution sequences computed from the specification and the observer by an asymmetric product. However the number of sequences matching this definition is quite large. This could be any sequence over inputs and outputs recognised by the observer. Most of the possible sequences may not conform to the actual IUT behaviour; thus, of no practical interest. This may happen because the asymmetric product produces any sequence of actions recognised by the observer. Therefore, an heuristic to restrict the TG to the most promising execution sequences is proposed.

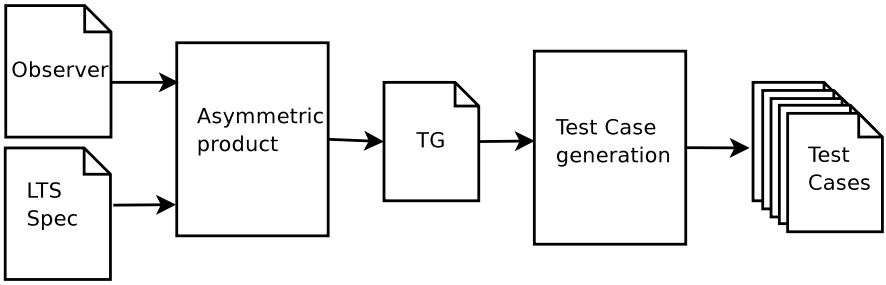

Fig. 3. Test case generation algorithm sketch.

The test case (TC) generation consists of a traversal search through the TG to extract subgraphs that are controllable (i.e. with inputs and outputs) and containing at least a sequence of actions recognised by the observer. The sketched algorithm (Figure 3) to generate the test cases is based on the computation of the asymmetric product between the specification and observer, traversal algorithms and identification of strong connected components. The generated test cases produce fail verdicts only if the IUT does not satisfy the specified property.

3.4 Test Purpose Generation from CTL Formulas

In [28], we present an approach to the automatic generation of test purposes for reactive distributed systems from temporal properties specified in CTL (Computation Tree Logic). This is based on an adaptation of the CTL model checking [5] technique, which provides efficient algorithms to analyse state spaces. The CTL formula is checked against the model, making possible analysis over the model to generate an abstract graph representing the test purpose equivalent to the formula.

Based on the extended conformance test framework in [6], presented in Section 2.2, we can conclude that if an IUT exhibits the behaviour defined by a test purpose e, it satisfies the property stated by e. In an analogous way, the model checking technique states about the satisfiability between a model of a reactive system [5] given as a kripke structure M and a given property P as a temporal logic formula f. This is a search problem over the set of states S of M. Formally: $\{s \in S \mid M, s \models f\}$. Since we can define both problems, i.e. model checking and property oriented testing, as satisfiability problems and both relate a model and a property, we can establish a relation between them. Thus, for $e \equiv f$ and $i_{IUT} \in MODS$: i_{IUT} **rev** $e \iff \exists s \in S : M, s \models f$.

Considering this assumption, we present an approach to generate test purposes from CTL formulas. The approach is based on an adaptation of the well known CTL model checking technique [5]. The approach consists of an adaptation of a model checker algorithm to extract model traces representing examples and counter-examples (if there are any) from the state space and later analysis over these model traces to generate an abstract graph representing the test purpose (Figure 4).

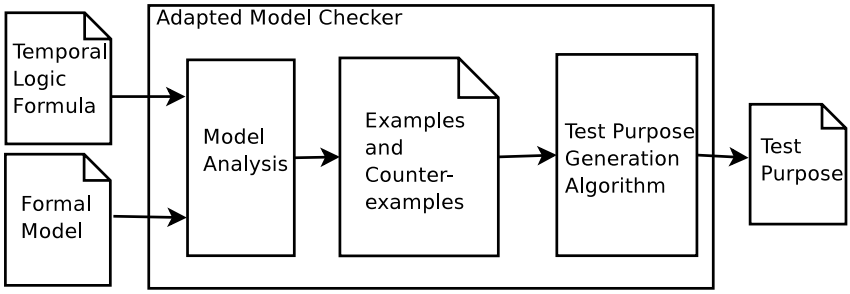

Fig. 4. Test purpose generation process.

The analysis over the examples and counter-examples are made to identify the relevant transitions w.r.t. the specified property. Such transitions compose the the test purpose, specified in the form of an LTS. Formally, a test purpose is a tuple $e = (Q, A, \rightarrow, q_0)$, where Q is a finite set of states, A the alphabet of actions, $\rightarrow \subseteq Q \times A \times Q$ the transition relation and $q_0 \in Q$ the initial state. The test purpose is equipped with two sets of special states *accept* and *refuse*.

The algorithm developed to perform the extraction of examples and counter-examples from the kripke structure of the specification is based on the traversal algorithms of the model checkers (e.g. depth search traversal algorithm). The examples are used to provide information of the accepted behaviour defined by the test purpose. The relevant transitions are then taken to construct the accept traces of the test purpose. The irrelevant ones are abstracted, usually by "*-transitions". Such *-transitions replace any occurring transition, except the transitions leading to another states.

Since the model checking technique is defined over transitions and states in terms of the kripke model, the use of LTS would lead to a misrepresentation of the property. The abstraction by *-transitions may be higher than the necessary to make the LTS correspondent to the formula, making possible the generation of test cases with transition sequences that may lead to property violation. To solve this problem, counter-examples of the formula, containing such undesirable transitions, are used to restrict the LTS to be generated. These transitions compose the traces leading to the *refuse* states of the test purpose. These states are interesting to the non-determinism problem of reactive systems too. It provides more restriction power to the test case generation.

To simplify the analysis over the extracted traces of the state space (i.e. examples and counter-examples), we define a simplified representation of its states in an abstract way. Thus, we represent these traces by a basic finite state machine defined by the tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite non-empty set of states, δ a finite

set of alphabet symbols accepted by the machine, $\delta : Q \times \Sigma \longrightarrow Q$ a transition function, $q_0 \in Q$ a initial state and $F \subseteq Q$ a set of final states, called accept states. The states of each trace are classified into sets defined by the propositions of the respective CTL formula, based on the satisfiability of the states w.r.t. to the formula propositions. Figure 5(a) shows an example of a representation of the CTL formula $EU(p, q)$. The states of the example are classified into two sets of state types, p and q. The states satisfying the proposition p are called p-states and the state satisfying the proposition q is called q-state. For $EU(p, q)$ formulas, the q-state represents the accept state of the machine.

(a) Example (b) Counter-example

Fig. 5. Simplified representation of traces of the $EU(p, q)$ formula.

The analysis algorithm classifies the relevant and the irrelevant transitions of the traces w.r.t. to the property based on the detection of the state changes over the simplified representation of them. This is done by identification of the transition and/or sequence of transitions necessary to the state changes. This identification consists of classifying the transitions of each trace extracted into the two sets (the relevant and the irrelevant). To detect a sequence of transitions necessary to cause a state change, the algorithm performs an intersection operation over the two sets. Only sequences of transitions that can occur in alternate orders are detected. Therefore, two subsets of the relevant set are created, one for the transitions identified in the intersection operation and one for the others.

After the examples and counter-examples analysis and transitions classification steps, the next step performed is the test purpose generation. The transitions of the two subsets of the relevant ones are used to construct the test purpose graph: (*i*) leading to *accept* states in case of the transitions obtained from examples and (*ii*) leading to *refuse* states in case of the transitions obtained from counter-examples. Figure 6 shows a test purpose generated from the examples of Figures 5(a) and 5(b).

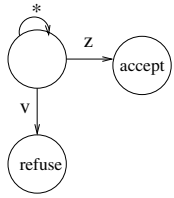

Fig. 6. Test purpose generated from the graphs of Figures 5(a) and 5(b).

This approach aims to solve the problem of generation of test purposes in an automatic way. It takes benefits of the efficient CTL model checking technique to generate the test purposes. It also is aimed at providing a direct link between the model checking and testing processes as a way to facilitate the investigation

of properties over model and implementation. A case study [28] was performed with the Mobile IP protocol [23]. The model checker adapted to generate the test purposes was the Veritas [25]. Test cases were generated with the TGV tool from the test purposes generated from the Mobile IP and some CTL formulas. Some analysis were performed comparing the CTG provided by TGV and the Mobile IP model, allowing us to conclude about the e-exhaustiveness [6] of the generated test suite.

4 Test Generation from Symbolic Transition Systems

Symbolic Transition Systems (STSs) extend LTSs by treating data symbolically, that is, by incorporating the explicit notion of data and guarded transitions using first-order logic, bringing the specification to a more abstract level [11]. The idea is to cope more effectively with the state explosion problem [5] and provide more understandable representations of test purposes and test cases. The semantics of STSs is given in terms of LTSs that can be even infinite. Therefore the theoretical testing framework for LTSs can be extended to STSs, even though further challenges need to be faced. Approaches based on STSs are presented in the next subsections.

4.1 STG - Symbolic Test Generation Tool

STG [4] is a tool for the generation of symbolic test cases for reactive systems. The underlying model to represent the system and test purpose specifications is a special kind of LTS, called Input-Output Symbolic Transition Systems (IOSTS) [26], which takes into account variables and parameters. The ioco theory [29] is adapted to cope with the IOSTS model. However, the quiescence is not considered in this new approach.

As input parameters, STG receives NTIF [6] [12] descriptions of the specification (of the system) and test purpose (Figure 7). These descriptions are automatically translated into IOSTS. The subsequent processes performed by the tool consist of the symbolic test case generation and later translation of them into a C++ implementation in order to make the execution possible.

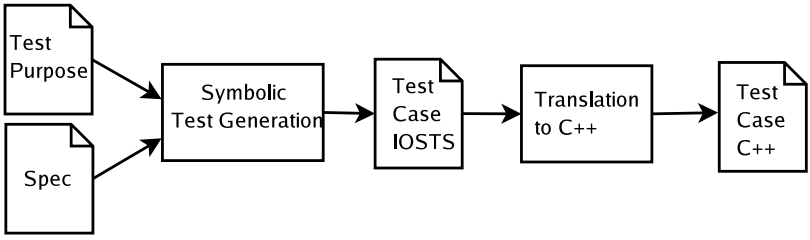

Fig. 7. Symbolic Test Generation Process.

The symbolic test generation produces symbolic test cases, which are reactive programs covering all the behaviours of the specification that are targeted by the

[6] NTIF is a LOTOS-like language.

test purpose. This process begins with a product between the specification and test purpose to select the traces corresponding to the accepting states defined by the test purpose. After that, STG determines the IOSTS generated from the product. Non-determinism must be eliminated prior to the test case execution. In the next step, verdicts are added to the appropriate states, according to the test purpose. A simplification is then performed to eliminate infeasible transitions, which may lead to unreachable parts in the test case. Finally, to obtain an executable test, the abstract symbolic test case is translated into a concrete test C++ program.

The STG tool was applied to simple case studies [3] for testing versions of the CEPS (Common Electronic Purse Specification) [2] and of the file system of the 3GPP (Third Generation Partnership Project) card. The main contribution of this technique is a set representation of states rather than an enumeration of the state space from the explicit representation, solving the state space explosion problem. Due to the symbolic representation of the generated test cases over variables and parameters of the specification, a single test can be applied to implementations based on different specification parameters values, which need to be instantiated only at the test execution time. The test cases are also more readable by humans.

4.2 Test Generation Based on Symbolic Specifications

In [11], the authors propose an algorithm to symbolic test generation based on the ioco theory [29]. The underlying model of LTS is extended with the concepts of location variables and data, allowing to specify data dependent behaviour. This extended model is called Symbolic Transition Systems (STS). The test generation algorithm for the STS model treats data symbolically, with data types represented by sets of values (algebras) and first order formulas specifying values or predicates.

The concept of gate is added to the STS definition, aiming at data communication. To cope with the observable behaviour at the interfaces, the gates are partitioned into two sets: input and output gates. The use of algebras and first order logic allows to combine STSs with any formalism for the specification and data manipulation. Test purposes are not used, and the test cases are generated based on the identification of the observable behaviours of the specification.

The proposed algorithm is sound and complete w.r.t. the ioco theory. It aims at combining the test generation from STSs with an on-the-fly execution of the test cases. Thus, only the part of state space corresponding to the observations made while testing is generated. A function to decide whether the currently executed trace is a visible trace of the specification is assumed to be implemented. Such function is aimed at guaranteeing the ioco relation and gives verdicts. There is no tool implemented based on this approach.

5 Comparing Approaches

Table 1 summarises the different LTS and STS based approaches presented in Sections 3 and 4. Note that they have different goals and focus either on test purpose or test case generation. In fact, they are dependent on other tools to be applied in

Approach	Goal	Algorithms	Tool Support	Output
[16] Section 3.1	Test case synthesis from IOLTS specifications and IOLTS test purposes	synchronous product, on-the-fly and traversal algorithms	TGV	Test cases represented by an IOLTS test graph
[14] Section 3.2	Test purpose generation from LES	Maximal configuration identification	Prototype	MSC test purposes
[8] Section 3.3	Test case generation based on LTL test purposes and partial specifications	asymmetric product and traversal algorithms	An extension of TGV is suggested	IOLTS test cases
[28] Section 3.4	Test purpose generation from CTL formulae	traversal algorithms and examples counter-examples extraction	Extension of the Veritas model checker	LTS test purposes
[4] Section 4.1	Symbolic test case generation from IOSTS specifications	non-determinism and infeasible transitions elimination	STG	Unparameter. C++ test cases
[11] Section 4.2	Test case generation from STSs based on the ioco relation	test generation and on-the-fly execution of test cases		STS test cases

Table 1
Summary of LTS and STS based approaches.

a test experiment, for not covering a complete test process or even a complete test selection process. For instance, the approach in [28] uses the TGV tool presented in [16] for test case generation.

Approaches based on LTS are usually independent of a particular specification notation. They take as advantage the fact that a reasonable number of notations for reactive systems have their semantics given in terms of LTS and a number of tools are available for generating an LTS view of the model [13]. Also, LTS models are simple and suitable for the implementation of classical algorithms on test case selection. In addition, LTS models often represent abstract views of real programs that can be reasonably modelled by observable behaviour such as inputs and outputs between a program and its environment, providing a good abstraction for analysis and verdict on testing execution results.

As a disadvantage, LTS models can be too big or even infinite due to the state space explosion problem [5]. For this, algorithms are usually based on on-the-fly techniques such as the ones presented in Sections 3.1 and 3.4. Also, only a partial specification is required as in Section 3.3. Moreover, it is usually not trivial to exploit the syntactical and/or semantical information on states once an LTS model is generated. This is also addressed in 3.4 by classifying between relevant and irrelevant transitions so that a high level property can be abstracted. Furthermore, it may be difficult or even impractical to specify test purposes as LTS models. Therefore, attempts have been made to generate test purposes from abstract specification of properties, such as presented in Sections 3.3 and 3.4, where Rabin automaton and CTL are considered for test purpose specification, respectively.

As mentioned before, whenever data types with infinite domains are involved, the state explosion problem becomes worse due to the fact that an explicit state representation for each single value is required in LTSs. Therefore, STSs are being considered instead of plain LTSs, since they tend to be more concise and abstract representations that consider both state and data information. Approaches based on STSs are presented in Section 4. The drawbacks of these approaches are related to the decidability and computability of logic that is used to specify data types. A common solution in these cases is to restrict the use of logic to feasible predicates as in [20]. Moreover, appropriate data generation for complex data types is a challenge that may impair test case generation and effective selection.

6 Concluding Remarks

An overview of approaches to test purpose and test case generation and current research towards a discipline of property oriented testing for reactive systems is presented, focusing on LTS and STS. In both cases, theoretical background and tools have been developed along with case studies. LTS approaches may demand simpler algorithms for test case generation, but they have to deal with the state explosion problem due to the explicit representation of data values when complex data structures are involved. On the other hand, STS approaches focus on more concise and abstract representations of the system, but decidability and computability of data specifications are of concern when generating test cases. In both cases, test case selection is still a challenge. Appropriate techniques for selecting effective test cases/data leading to feasible test suites among the mass of generated ones are object of current investigation. As the problem is hard, it is likely that solutions will be connected with specific kinds of properties.

References

[1] Bernot, G., *Testing against formal specifications: a theoretical view*, in: *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Advances in distributed computing (ADC) and colloquium on combining paradigms for software development (CCPSD): Vol. 2* (1991), pp. 99–119.

[2] CEPSCO, *Common electronic purse specification, technical specification*, Technical report, http://www.cepsco.org (2000).

[3] Clarke, D., T. Jéron, V. Rusu and E. Zinovieva, *Automated test and oracle generation for smart-card applications*, in: *E-SMART '01: Proceedings of the International Conference on Research in Smart Cards* (2001), pp. 58–70.

[4] Clarke, D., T. Jéron, V. Rusu and E. Zinovieva, *Stg: A symbolic test generation tool*, in: *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2002), pp. 470–475.

[5] Clarke, E., O. Grumbert and D. Peled, "Model Checking," MIT Press, 1999.

[6] de Vries, R. G. and J. Tretmans, *Towards formal test purposes*, in: *Proceedings of 1st International Workshop on Formal Approaches to Testing of Software (FATES)*, Aalborg, Denmark, 2001, pp. 61–76.

[7] El-Far, I. K. and J. A. Whittaker, *Model-based software testing*, Encyclopedia on Software Engineering (2001).

[8] Fernandez, J., L. Mounier and C. Pachon, *Property oriented test case generation*, in: *Formal Approaches to Software Testing, Proceedings of FATES 2003*, Lecture Notes in Computer Science **2931** (2004), pp. 147–163.

[9] Fernandez, J.-C., C. Jard, T. Jron and G. Viho, *An experiment in automatic generation of conformance test suites for protocols with verification technology*, Science of Computer Programming **29** (1997), pp. 123–146.

[10] Fink, G. and M. Bishop, *Property-based testing: a new approach to testing for assurance*, SIGSOFT Softw. Eng. Notes **22** (1997), pp. 74–80.

[11] Frantzen, L., J. Tretmans and T. A. C. Willemse, *Test generation based on symbolic specifications*, in: *Formal Approaches to Software Testing, Proceedings of FATES'04*, Lecture Notes in Computer Science **3395** (2005), pp. 1–15.

[12] Garavel, H. and F. Lang, *Ntif: A general symbolic model for communicating sequential processes with data*, in: *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems* (2002), pp. 276–291.

[13] Hartman, A. and K. Nagin, *The agedis tools for model based testing*, in: *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (2004), pp. 129–132.

[14] Henniger, O., M. Lu and H. Ural, *Automatic generation of test purposes for testing distributed systems*, in: *Formal Approaches to Software Testing, Proceedings of FATES'03*, Lecture Notes in Computer Science **2931** (2003), pp. 178–191.

[15] ISO/IEC JTC1/SC21 WG7, I.-T. S. Q., *Information retrieval, transfer and management for osi; framework: Formal methods in conformance testing*, Technical Report Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500, ISO - ITU-T, Geneve (1996).

[16] Jard, C. and T. Jéron, *Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems*, Software Tools for Technology Transfer (STTT) **6** (2004).

[17] Jorgensen, P. C., "Software Testing - A Craftsman's Approach," CRC Press, 2002.

[18] Lamport, L., "Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[19] Lu, M., "Generation of tests from labeled event structures," Master's thesis, University of Ottawa, Ottawa, Canada (2003).

[20] Machado, P. D. L., *On oracles for interpreting test results against algebraic specifications*, in: A. M. Haeberer, editor, *Algebraic Methodology and Software Technology, AMAST'98*, LNCS **1548** (1999), pp. 502–518.

[21] Machado, P. D. L., E. A. S. Oliveira, P. E. S. Barbosa and C. L. Rodrigues, *Testing from structured algebraic specifications: The veritas case study*, Electronic Notes in Theoretical Computer Science **130** (2005), pp. 235–261, proceedings of Brazilian Symposium on Formal Methods, SBMF'2004.

[22] McGregor, J. D. and D. A. Sykes, "A Practical Guide to Testing Object-Oriented Software," Object Technology Series, Addison-Wesley, 2001.

[23] Perkins, C., "Rfc 3344:IP mobility support for IPv4," (2002), status: Proposed Standard.

[24] Rabin, M. O., "Automata on Infinite Objects and Church's Problem," American Mathematical Society, Boston, MA, USA, 1972.

[25] Rodrigues, C. L., F. V. Guerra, J. C. A. de Figueiredo, D. D. S. Guerrero and T. S. Morais, *Modeling and verification of mobility issues using object-oriented petri nets*, in: *Proceeding of 3rd Int. Information and Telecommunication Technologies Symposium (I2TS2004)*, 2004.

[26] Rusu, V., L. du Bousquet and T. Jron, *An approach to symbolic test generation*, in: *International Conference on Integrating Formal Methods (IFM'00)*, Lecture Notes in Computer Science **1945** (2000), pp. 338–357.

[27] Schmitt, M., A. Ek, B. Koch, J. Grabowski and D. Hogrefe, *Autolink - putting sdl-based test generation into practice*, in: *IWTCS: Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems* (1998), pp. 227–244.

[28] Silva, D. A. and P. D. L. Machado, *Towards test purpose generation from ctl properties for reactive systems*, Electronic Notes in Theoretical Computer Science **164** (2006), pp. 29–40, proceedings of the Second Workshop on Model Based Testing (MBT 2006), Second Workshop on Model Based Testing 2006.

[29] Tretmans, G. J., *Test generation with inputs, outputs and repetitive quiescence*, Software—Concepts and Tools **3** (1996), pp. 103–120.

[30] Tretmans, J., *Testing concurrent systems: A formal approach*, in: J. Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, Lecture Notes in Computer Science **1664** (1999), pp. 46–65.