Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 3–19
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

# Automated Software Testing of Asynchronous Systems

Percy Pari Salas [1,2] Padmanabhan Krishnan [3]

*Center for Software Assurance*
*School of Information Technology, Bond University*
*Gold Coast, QLD 4229 Australia*

**Abstract**

Test automation faces challenges when applied to the testing of asynchronous systems. Automated testing tools need to deal with local non-determinism and, contrarily to most theoretical work, imperfect communication channels. We use event structures as the formalism to reason about the testing process. We differentiate between controllable and observable events but rely only on the sequence of controllable events to generate the test case. Observable events are used mainly as test oracle and to update the system state. We take existing testing tools and enhance them with practical mechanisms that allow them to perform asynchronous testing. These extensions are based on sound theory and have shown practical in dealing with real systems.

*Keywords:* Asynchronous systems, Model-based Testing

## 1 Introduction

Advances in technology, such as networking and parallel processing, have enabled the development of distributed and concurrent systems. Most of these systems consist of subsystems, which can be independent or autonomous. In some cases, there is a need to allow a subsystem, who is sending a message, to continue with its tasks without waiting to determine what happened to the message. All these give the whole system an asynchronous characteristic.

Testing and validation of asynchronous systems is challenging. Many of these challenges, such as local non-determinism and communication delays, have been addressed theoretically [4,3]. However, from a practitioner's point of view, these solutions are not always available. In particular, automated testing tools need to deal with practical implementation challenges. For example, perfect communication

[1] Supported by a special scholarship from KJRoss and Associates
[2] Email: pparisal@staff.bond.edu.au
[3] Email: pkrishna@staff.bond.edu.au


doi:10.1016/j.entcs.2009.09.048

channels (without losses or delays) used in the theory are not present in real systems. Therefore, testing tools need to handle systems with imperfect channels. In the same way, if some subsystems rely on external choices, tools for testing those subsystems need to handle non-determinism.

As an example, consider a system with two agents $A$ and $B$ in which the communications protocol is described as
$A \rightarrow B : m_1$
$(B \rightarrow A : m_2 + B \rightarrow A : m_3) || A \rightarrow B : m_4$
where $m_1$, $m_2$, $m_3$ and $m_4$ are messages, $\rightarrow$ indicates the direction of the communication, $+$ represents choice and $||$ represents concurrent composition. Agent $A$ initiates the computation by sending $m_1$ to $B$. Then, $B$ can choose to answer with $m_2$ or $m_3$. In the meantime, $A$ doesn't have to wait for $B$ to respond to send another message, $m_4$. An automated testing tool can generate a test sequence $m_1\ m_2\ m_4$, however, $m_1\ m_3\ m_4$ can also be a valid sequence as it will be $m_1\ m_4\ m_2$. Certainly, the testing tool by itself cannot predict $B$'s response, and always waiting for $B$ to respond before continuing can lead it to produce a test sequence that probably will not match the behaviour of the real system. So, test sequence $m_1\ m_2\ m_4$ could fail because the system exhibits $m_1\ m_3\ m_4$, but that does not mean that the system is wrong.

Our practical approach aims to make testing theory accessible to practitioners. To address this goal we, first, bring the system under test (SUT) and the testing tool to work together in an on-line approach. Although this approach is present at some extent in available tools, we have extended capabilities already present in these tools to make this approach to work in *black-box* testing environments. This is, we deal mostly with available interfaces rather than internal workings of the system. Second, we extend existing tools to implement theoretical knowledge on how to handle asynchronous communications. We deal not only with communication delays but also communication losses which are less common in the literature. Finally, we rely on abstract models to drive the testing process.

In summary, the main contribution of this work is to allow for practical model-based testing applied to asynchronous systems.

The rest of this paper is organised as follows. In section 2 we describe a small system that serves as a running example to clarify the concepts we introduce through sections 3 and 4. Section 3 introduces notations and definitions for event structures used to model asynchronous distributed systems. In section 4 we describe the process that allows asynchronous testing of distributed systems, and in section 5 we present details of a particular implementation of this process by using existing testing tools. Then, section 6 shows an example of testing a more complex asynchronous distributed system. Finally, a brief description of related work and our conclusions are shown in sections 7 and 8.

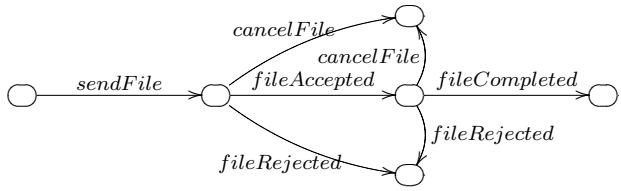


Fig. 1. Simplified model of a file exchanging system

## 2 Running example

Consider a system of file exchanging where a user (the sender) can connect to another one (the receiver) and send a file. There are, obviously, two distinguishable sides on this system. From the sender's point of view, he is responsible for executing only a subset of the actions in the system, he calls them, the subset of *controllable actions*. The other subset of actions is executed by the receiver. However, the sender needs to be notified when the receiver executes one of these actions. Thus, the sender calls those actions the subset of *observable actions*. We describe the actions that take place in this system mainly from the point of view of the sender.

A user can initiate the process by executing the action *sendFile*. The receiver can, then, accept the file or reject the file. From the sender's point of view, when the receiver accepts the file, action *fileAccepted* is observed. On the other hand, if the file is rejected, action *fileRejected* is observed. Even when the receiver has accepted the file, the transfer can be aborted at any time before it is completed. An aborted transfer results also in the occurrence of action *fileRejected*. The sender can also cancel the transfer at any time, which is represented as the execution of action *cancelFile*. Finally, if an accepted file is successfully transferred, action *fileCompleted* is observed. The graph in Figure 1 shows a pictorial description of the interactions of this system.

This system is asynchronous in the sense that the sender does not need to wait for the receiver to accept or reject a file. The sender can send a file and then another one before he observes any action executed by the receiver. Such behaviour can also be represented into a graph by interleaving the actions related to each of the files being transferred.

## 3 Modelling asynchronous systems

State machine based formalisms such as labelled transition systems (LTS) have been commonly used to describe system's behaviour in general and therefore used in model-based testing. Particularly, LTS have also been used to describe concurrent and asynchronous system's behaviour not without relying on some extensions that capture particular characteristics of these systems [13,15,2]. However, reasoning about asynchronous testing in terms of LTS can turn itself into a complex process. It would require, for example, to define or extend the concepts of refinement and alternating simulation as in [15] or would require to introduce different equivalence relations as in [2]. We decided to take another approach in order to simplify the

presentation of the testing process. Therefore, here we work in the settings of event structures.

Event structures are, in general, unfoldings of other formalisms like labelled transition systems and PetriNets [7]. They differ from transition systems mainly in that event structures have the concept of concurrency naturally embedded while transition systems represent it by combining choice and sequencing. Nevertheless, the literature contains different approaches for deriving one formalism from the other [6,7].

Different kinds of event structures have been defined in the literature derived from the classical ones introduced in [10] and [16]. The particular event structure model that we consider was introduced by van Glabeek and Plotkin in [14] and has been adapted to suit our purposes. Before we define this event structure, some preliminary concepts need to be introduced.

*Events.* An event denotes a *unique* execution of an action. As an example, assume that Alice is using the system described in Section 2 for sending files to Bob and she realises the following behaviour

*sendFile* · *fileAccepted* · *sendFile* · *fileRejected* · *fileCompleted*

where action *sendFile* is allowed to perform twice. In an event structure, these two instances would be differentiated and modelled as distinct events. There are several mechanisms which can be used to differentiate between events, e.g. timestamps, sequential id's. These mechanisms depend on the implementation and are not discussed here.

*Causality.* This is a relationship between events, which characterises how events enable (or cause) other events. In our example, event *fileAccepted* can (possibly) occur if event *sendFile* has occurred before. In addition, more than one event can cause a particular event. For example, *fileCompleted* is only possible after both, *sendFile* and *fileAccepted* have occurred.

Considering the concepts introduced above, we define our event structure as follows.

**Definition 3.1** An event structure is a 4-tuple $M = (\Sigma, A, \mapsto, \ell)$ where

- $\Sigma$ is a set of events
- $A$ is a set of actions
- $\mapsto \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ is an enabling relation
- $\ell : \Sigma \times A$ is the action-labelling function

For representing asynchronous systems, the set of events $\Sigma$ is partitioned into two sets: $\Sigma_c$ the subset of *controllable events* and $\Sigma_o$ the subset of *observable events*. It should be obvious that this partitioning extends into the set $A$ of actions resulting in $A_c$ and $A_o$ disjoint subsets of controllable and observable actions, respectively. As we mentioned in the description of our example, we call *controllable actions* the set of actions that can be directly invoked by the testing environment. Contrarily, actions which are executed independently by the SUT (or its environment), usually in response to other actions, are called *observable actions*.

In our definition of event structure, the interpretation of $E \mapsto E'$ for two sets $E, E'$ of events is that all events $e'$ in $E'$ are enabled if and only if all events in $E$ have occurred. It does not place any restrictions on the occurrence of individual events in $E'$. However, from the definition of events, it is stated that only one event in $E'$ can occur at a time. As an example, consider the graph in Figure 1. The enabling relation that describes the system is

$$\begin{array}{lcl}
\{\} & \mapsto & \{\textit{sendFile}\} \\
\{\textit{sendFile}\} & \mapsto & \{\textit{cancelFile}, \textit{fileAccepted}, \textit{fileRejected}\} \\
\{\textit{sendFile}, \textit{fileAccepted}\} & \mapsto & \{\textit{cancelFile}, \textit{fileCompleted}, \textit{fileRejected}\} \\
\{\textit{sendFile}, \textit{fileRejected}\} & \mapsto & \{\} \\
\{\textit{sendFile}, \textit{cancelFile}\} & \mapsto & \{\}
\end{array}$$

A specific run of an event structure leads to the concept of an *event trace*. In order to define an event trace, for a finite sequence of events $w = e_1 \cdot e_2 \cdot \ldots \cdot e_n$, first we denote $\overline{w} = \{e_1, e_2, \ldots, e_n\}$ the set of all elements in $w$. The symbol $\xi$ denotes an empty sequence and $\overline{w} = \{\}$ for $w = \xi$. Finally, $w_i$ denotes the prefix of $w$ up to its (i-1)-th element, this is, $w_1 = \xi$ and $w_i = e_1 \ldots e_{i-1}$, for $1 < i \leq n + 1$.

**Definition 3.2** [Event trace]

A given sequence of events $w = e_1 \cdot \ldots \cdot e_n$ is an *event trace* of the event structure $M = (\Sigma, A, \mapsto, \ell)$ if for all $1 \leq i \leq n$,

- $e_i \notin \overline{w_i}$, and
- $\exists E : \overline{w_i} \mapsto E \wedge e_i \in E$

For testing purposes it is necessary to define when a trace of events enables the occurrence of other events. Thus, given an event trace $w$ and a single event $e$, we define the predicate $\textit{enabled}(w, e)$ that holds iff $e \notin \overline{w}$ and $\overline{w} \mapsto E \wedge e \in E$. Similarly, to improve readability we denote $\textit{observableSuccessors}(\overline{w})$ the set of potential observable successors of $w$, defined as $\textit{observableSuccessors}(\overline{w}) = \{e | e \in \Sigma_o \cap E \wedge \overline{w} \mapsto E\}$.

The language represented by an event structure $M$ is defined as $L(M) = \{w \in \Sigma^* | w$ is an event trace of M $\}$. Given a system represented by the language $L$ that conforms to the previously defined event structure, an automated testing process for that system is described in the next section.

## 4 An asynchronous testing process

In distributed systems we can easily distinguish different components of the system because of their location. As mentioned before, in general, we can assume there exist two parts in these systems. There is one part whose actions we can control and there is another part whose actions we can only observe. We have described before a behavioural model that describes events that occur in both parts of the system.

Before we can describe how to test an asynchronous system, we need to define what a test case is in our context. We represent a test case as a set of possible event sequences. In order to define a test case in a more formal way, given a word $w \in \Sigma^*$ and $S \subseteq \Sigma$ we need to introduce $w \downarrow_S$ as the sub-word obtained by erasing all the symbols not in $S$.

In addition, we introduce the concept of *spontaneous observable events* representing observable events that can occur at any time during an execution. Formally, the set of *spontaneous observable events* $\Omega$ is defined as $\Omega = \{e | e \in (E \cap \Sigma_o) \wedge \{\} \mapsto E\}$. To clarify this concept, we recall the file exchanging system in section 2. Assume that Alice is sending files to Bob, but there is a time when Bob wants also to send a file to Alice. From Alice's point of view, at the time Bob sends a file, she observes another action that was not included in the model. Thus, we add action *fileSent* to our model. This action is observable and no Alice's action can be seen as the cause for its execution. Then, we say the execution of this action is an *spontaneous event*. The reasons to introduce this concept become apparent later in this section.

Now, we are ready to formally define a test case for asynchronous testing.

**Definition 4.1** [Test case] Given a sequence of controllable events $E = e_1 \cdot e_2 \cdot \ldots \cdot e_n$, a test case is the set of event sequences represented by $TC(E) = \{w | w \in L \wedge w \downarrow_{\Sigma_c} = E \ \wedge \ \forall e_o \in (\Sigma_o \backslash \Omega), w \cdot e_o \notin L\}$.

The previous definition states some properties of a generated test case for asynchronous systems. These properties are used to determine when a test case passes or fails its execution. These properties are:

(i) for all event sequences in a given test case, controllable actions are always executed in the same order; and,

(ii) no observable action, excepting spontaneous actions, is expected after the test sequence has been executed.

The first property states what is common for testing theories based on abstract models, this is, a test case *passes* if the implementation can execute the same (sub)sequence of (controllable) actions as the model. The second property, assumes that the execution of most observable actions are triggered by other actions (either controllable or observable). Those who are not, are spontaneous. Then, suppose there is an event sequence in a test case such that after the sequence has been executed, an observable event (not spontaneous) is still expected to occur. The test sequence is executed successfully, therefore the test case passes. However, the pending event can still reveal a faulty implementation. The verdict will be wrong. In fact, this test case would be undecidable because it misses information about expected responses of the system. The second property, forbids such kind of test cases.

To deal with asynchronous testing we need to decouple (at some extent) controllable and observable actions. This is, we cannot rely on the fact that the response to an action we decide to execute will follow immediately. Therefore, there is a waiting time which can be used to request the execution of more actions. However,

we also cannot produce beforehand a list of actions to be executed because some actions will become enabled (or not) depending on the (non-deterministic) responses of the other part of the system. Thus, we define two algorithms, test generator and execution observer, which run in parallel during the testing process. Both execute independently but communicate (and synchronise if necessary) by accessing two global structures, a queue $Q_c$ of controllable events and a set $S_o$ of observable events.

Algorithms 1 and 2 show the generation and observation processes, respectively. In the algorithms, we denote $h \in \Sigma^*$ the sequence of all events that occurred previously and $\overline{h}$ the set of events in $h$. We also assume that there exists a *communication channel*, where all messages to the SUT requesting the execution of an action and all messages from the SUT notifying of the execution of an action are written to. The communication channel is represented as a queue of events $CC$. Additionally, the function *dequeue(Q)* removes the first element in queue $Q$ and returns it to be used in the algorithms.

**Algorithm 1** Test generator

$Q_c$ is **empty**
$h$ is **empty**
$S_o = \Omega$
**while** $\exists e_c \in \Sigma_c$ such that $enabled(h, e_c)$ holds OR $(S_o \backslash \Omega) \neq \emptyset$ **do**
  **pick** randomly an enabled $e_c$
  update the execution history, $h = he_c$
  append $e_c$ to $Q_c$
  $S_o = observableSuccessors(\overline{h}) \cup \Omega$
**end while**

**Algorithm 2** Execution observer

**while** $Q_c$ and $(S_o \backslash \Omega)$ are not empty **do**
  $m \leftarrow dequeue(CC)$
  **if** $m \in \Sigma_o$ **then**
    update the execution history, $h = hm$
    **if** $m \in S_o$ **then**
      $S_o = observableSuccessors(\overline{h}) \cup \Omega$
    **else**
      Test fails
    **end if**
  **else if** $m \in \Sigma_c$ **then**
    $e_c \leftarrow dequeue(Q_c)$
    **if** $m \neq e_c$ **then**
      Test fails
    **end if**
  **end if**
**end while**

Summarising the process, the test generator appends events to the queue $Q_c$, therefore it also appends observable events to the set $S_o$ which, in principle, plays the role of oracle of the test case. The execution observer removes the events from $Q_c$ or $S_o$ as the execution is performed by the implementation. However, as non-determinism is resolved at certain points by implementation choices, the oracle $S_o$ needs to be modified to reflect the choices. Then, the choice of an event $e$ will remove events from $S_o$ which will not happen after $e$. Naturally, $e$'s enabling relations will also append new events to $S_o$.

The execution of test cases need to produce a verdict. Our algorithms do provide the concept of a test case's *fail* verdict. From it we can derive the concept of a *pass* verdict. A test cases passes if its execution does not fail and the algorithms terminate. An *inconclusive* verdict should be allowed when the algorithms do not terminate. However, we can force the *fail* verdict in those cases by introducing proper *time-out* events.

## 5 Implementing the testing process

In order to implement the automated testing process described before, existing tools and some extensions we have developed are integrated with the SUT as it is shown in Figure 2. As one can see, the testing process is driven by the system's model and performed by the testing tool. The testing tool links itself to the SUT to verify its conformance to the model. In our implementation we use SmartMBT [4] as the testing tool. This tool takes as input the specification of a system described in terms of a labelled transition system. It can be used to generate test cases from the model by applying different approaches, e.g. Chinese Postman algorithm, random selection, probabilistic selection. It has also the ability to link itself to the SUT in such way that while a test sequence goes through the model it is executed in the SUT. Thus, the SUT execution can be validated against the model, "on-the-fly".

The "on-the-fly" approach of SmartMBT relies on the tool having a channel that communicates with the SUT. In the middle of this channel, there is a component, usually a script, that translates the action names from the model into actual procedures or functions in the SUT. Moreover, the tool relies on the assumption that this script can determine the state of SUT and compare it with the expected state in the model. From the point of view of the tool then, the channel only returns a "pass" or "fail" verdict.

Although testing of asynchronous systems is technically possible with SmartMBT, it cannot be fully automated. It will require tester assistance in some tasks, such as interpreting and executing responses from the SUT. Extensions we have developed into SmartMBT to allow fully automation are explained in the remaining of this section.

[4] SmartMBT is a model-based testing tool developed by K.J.Ross & Associates.

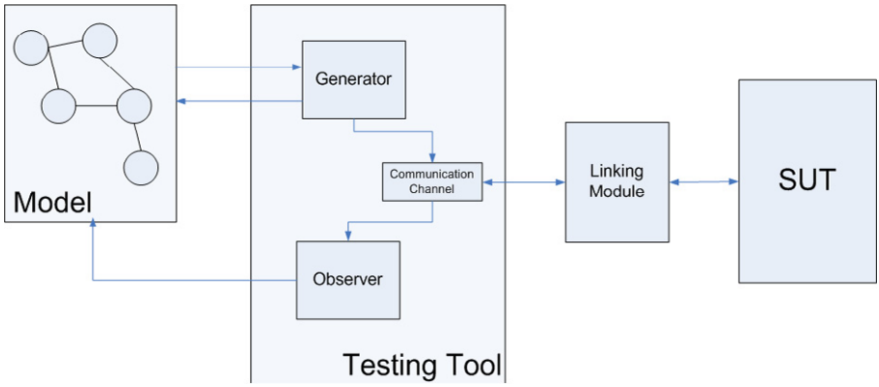


Fig. 2. Testing architecture for asynchronous systems

### *5.1 The model*

SmartMBT takes as input a labelled transition system that specifies the behaviour of a software system. The testing process is driven by the model. The model provides the testing tool with a set of enabled actions calculated from its current state. Usually, every action in a LTS model will be available for execution sometime. As we discussed before, that is not the case for asynchronous systems. There exist *observable* actions, which the SUT executes independently, usually in response to other actions. These actions should not be executed by the testing environment, therefore should not be included in any set of enabled actions. However, they should be part of the model because they actually update the state of the SUT (and of the model). We included a mechanism that allow the modeller to tag observable actions, and the tool to recognise them.

### *5.2 The generator module*

The generator module represents most of the normal workings of the SmartMBT tool. It gets from the model the set of enabled actions in the current state, picks one (or let the tester pick one) randomly and simulate its execution over the SUT updating the state of the system in the model. If linked to the SUT, this module also sends the chosen action through the communication channel firing its execution into the SUT. In our implementation, this module corresponds to the generator algorithm and has access only to controllable actions. This is, in a given state of the system it can choose randomly (or in a user driven way) among enabled actions tagged as controllable. In general, this component stops its execution if no enabled actions are provided by the model. In asynchronous systems, it can happen that on a defined state, no controllable actions are enabled but observable actions are still pending of execution. Our generator module remains in a "waiting" state when this case arises. The module only stops its execution if no actions, nor controllable neither observable, are enabled on a defined state.

### *5.3 The observer module*

This module is completely new in the testing tool. It has the ability to observe which (observable) actions have been executed in the SUT, usually in response of actions executed by the generator. This module implements our execution observer algorithm. It runs into a separate thread and is activated each time an action comes through the communication channel. Controllable actions are discarded because the generator retains the responsibility of updating the state of the model in this case. Observable actions are processed exactly as in the algorithm. This module relies in an extension of the *client module* which is explained next.

### *5.4 The linking module and communication channel*

We have extended SmartMBT allowing it to observe the execution of actions in the SUT rather than only receiving a *pass* or *fail* verdict. On this way, the responsibility of generating the verdict "returns" to the testing tool itself. Moreover, we do not rely on the capabilities of the channel to look into the current state of the SUT. We determine the state of the SUT based on the responses we observe in the channel. Additionally, to decouple the actual interface with the SUT instead of a script we include a module (the *linking module*) in charge of translating action names into messages to the SUT, and vice versa. This module also writes every in/out communication with the SUT into a queue representing the *communication channel*.

Figures 5 and 6 presented in the example section show screenshots of the actual interfaces of the extended tool and a SUT we connect to. In Figure 5 the upper window shows the list of executed actions and the verdict, activates the link with the SUT among other functions. The window below is an interface that allows the generator to let the tester choose the next action to be executed from the set of enabled actions (actions in the left hand side). Observable actions expected to be executed from the SUT are also shown (actions in the right hand side). This scenario shows how the execution of seven actions left the system in a state where five other were available for execution and six observable actions can also occur. However, the occurrence of an unexpected event in the SUT lead the testing tool to report an error.

In Figure 6 we show the SUT working together with SmartMBT. The upper part of the figure shows interaction with a server, incoming and outgoing messages. The window in the middle right is an interface that allows us to control the linking module developed in Java. Actions executed in the SUT can be moderated by the tester or passed directly to the observer. The window down shows a client that communicates with the server in the SUT. This client was modified to include the linking module.

In the next section we report via an example the use of this implementation in the testing of an asynchronous distributed application.

| |
|---|
| **buy(1,100)**, buyAck(1), **cancel(1)**, cancelAck(1), **buy(2,80)**, buyAck(2), fullFill(2,80) |
| **buy(1,100)**, **cancel(1)**, buyAck(1), **buy(2,80)**, partialFill(1,80), cancelAck(1), buyAck(2) |
| **buy(1,100)**, **cancel(1)**, **buy(2,80)**, buyAck(2), partialFill(1,80), cancelAck(1), buyAck(1) |
| **buy(1,100)**, **cancel(1)**, buyAck(1), **buy(2,80)**, fullFill(2,80), buyAck(2), cancelAck(1) |

Fig. 3. Valid sequences of actions in the FIX protocol

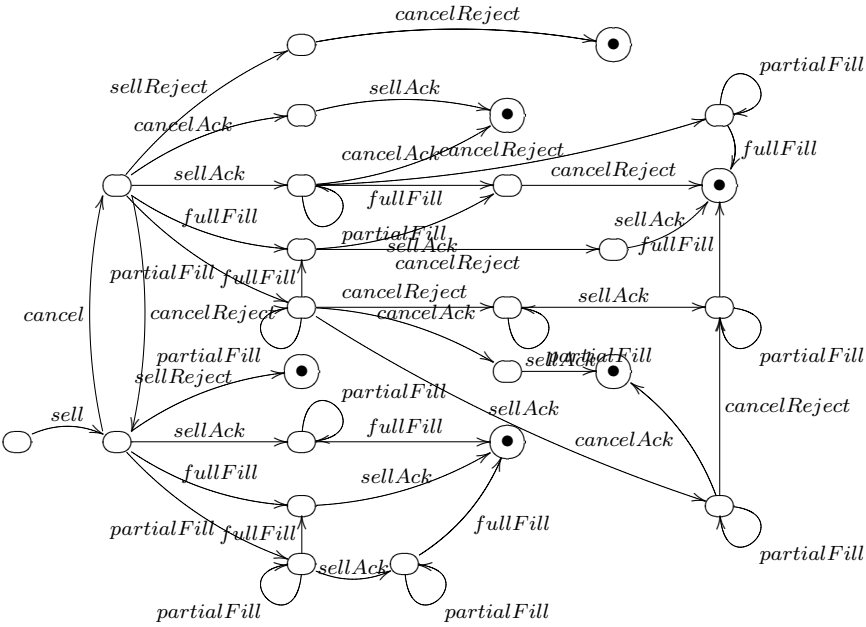


Fig. 4. Simplified model of the FIX protocol

# 6 Example: The FIX Protocol

Stock monitoring and financial data processing applications are classical examples for asynchronous distributed systems. Here we refer to a trading system that implements the Financial Exchange (FIX) protocol[5]. The FIX protocol define a series of messaging specifications for the electronic communication of trade-related messages. A trade operation has always buyer and a seller, represented by a client (that initiates a transaction by buying or selling some security) and a server (which plays the complementary role). Any system that implements the FIX protocol is asynchronous in the sense that a buyer client can send a message to the server requesting to buy something. Then the buyer can wait for the server to acknowledge his request, but in the meantime it can send another message to buy a different instrument, or maybe change his mind and request to cancel the previous buying order or alter the amount or price. On the other hand, the server processes the orders and can acknowledge the buying request, and actually fill the buying order before receiving the cancelling order. An example of valid sequences of actions that this trading system can execute is shown in Figure 3. The syntax and meaning of each action are described later in this section.

[5] http://www.fixprotocol.org

In Figure 3, actions requested (or controlled) by the user are in **bold**. Other actions represent responses from the server. The first sequence is what could be considered a normal case, each user action is linked to a proper response and they interleave. However, this does not happen always in real systems. Sequences in the second part of the table represent alternative sequences which are also valid but derive on a different result. The first alternative sequence, for example, shows delays in the responses from the server. Users continue doing requests before the server responds, so it happens that we ask to cancel a transaction that has not been acknowledged yet. It also happens that this transaction is partially filled before the cancellation takes place. In the second sequence, the acknowledgement *buyAck(1)* has probably been lost and resent. Thus, it arrives after the partial filling has been received. Nevertheless, it is still a valid sequence. Same losses happen to *buyAck(2)* and *cancelAck(1)* in the third sequence. However, the cancel request was processed on time to avoid the partial filling. In all the cases, the execution of a *sell(3,80)* action is assumed but, in principle, we have no means of controlling when it will happen.

For purposes of exemplifying the process of generating and executing test sequences from a system's model we use here a simplified description of the FIX protocol . We consider controllable actions *buy(id,amt)*, *sell(id,amt)*, *cancel(id)* and *alter(id)*. Actions *buy(id,amt)* and *sell(id,amt)* initiate a request for buying or selling an amount *amt* of some security. The buy (or sell) request is assigned with a unique identifier *id*. Action *cancel(id)* places a request for cancelling the order identified by *id*, and *alter(id)* request a modification on the order identified by *id*.

The server can respond by acknowledging the buy/sell request with actions *buyAck(id,amt), sellAck(id,amt)*; or it can reject the request with actions *buyReject(id), sellReject(id)*. The server can also fill a request, partially or completely, with actions *partialFill(id,amt)* or *fullFill(id,amt)* if it receives matching requests (a buy and a sell requests). Actions *cancelAck(id), cancelReject(id), alterAck(id), alterReject(id)* are suitable responses for the *cancel* and *alter* actions respectively. For illustration purposes only we show in Figure 4 part of the LTS that describes our model of the FIX implementation.

*Testing process*

Our algorithm requires the set of possible events to be finite. Otherwise, it will run indefinitely. We restrict the size of the set of possible events by restricting the number of available *id*'s to three. We have also modified the code in the client module of *QuickFixJ* [6] , an open implementation of the FIX protocol, to send generated action messages to the *server* and to receive messages from the server and translate them to the testing tool. Figures 5 and 6 show part of the execution of this example. In the real implementation actions have parameters other than the transaction *id*. For simplicity, we do not include these parameters in our discussions. In Figure 5 we can see actions that have been executed and observed. They are presented as

[6] http://www.quickfixj.org

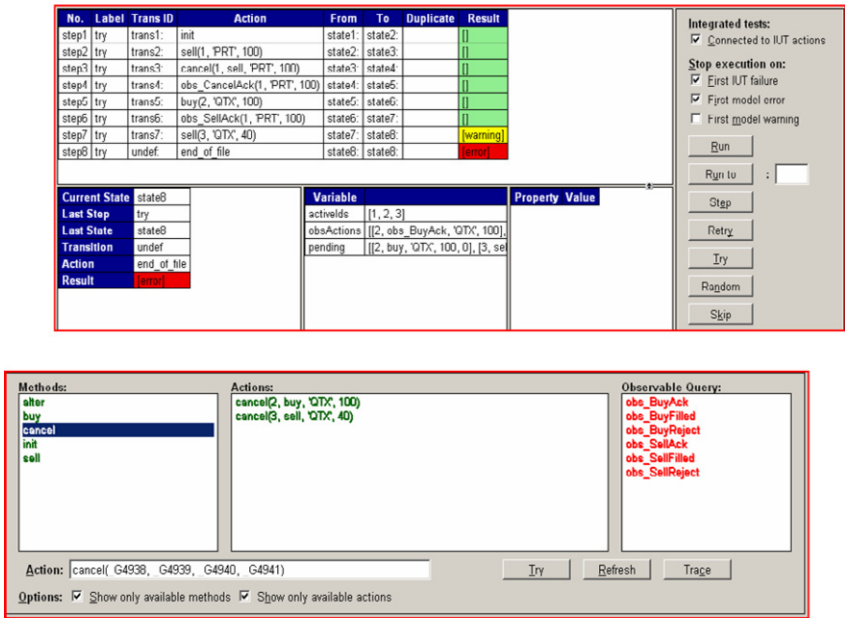


Fig. 5. The extended SmartMBT

sequential steps of the test. Each one can *pass* if it was expected to happen, or *fail* otherwise. We can see that actions $sell(1)$ and $cancel(1)$ have been requested in steps 2 and 3, and acknowledged in steps 4 and 6. We forced a fail in the last step by sending an unexpected action to the observer. We can also see that after step 7 ($sell(3)$), enabled actions are *alter*, *buy*, *cancel*, *init* and *sell*. However, enabled *cancel* events are only $cancel(2)$ and $cancel(3)$ ($cancel(1)$ has been already executed). In the same way, as no more *id*'s are available *sell* and *buy* events will not be available. Figure 6 shows the state of the FIX server which does not need major discussion.

In Table 1 we summarise the workings of the implemented testing process. We describe it as a sequence of events that interleaves execution requests by the testing tool and observation of executions performed by the SUT. The first column in the table shows events requested (and executed) by the test generator. Events observed in the communication channel are shown in the second column. Finally, the third column shows the evolution of the set of expected observable events. This process was executed in a closed environment, this is, no other components than a single instance of the testing tool communicates with the server.

We can observe in the first row of the table that the generation algorithm chooses to initiate the test case with event $sell(1, 100)$. Then, the set of expected observable events is updated (see column $S_o$). This event enables events $cancel(1), alter(1)$ as well as $buy(2), sell(2)$. In the second row we see that event $cancel(1)$ is executed. This updates the set of expected observable events. Note that in column $S_o$, the set in the second row has additional elements $cancelAck(1)$ and $cancelReject(1)$ which correspond to the expected responses for $cancel(1)$.

In the third row of our table, the observer reports events $sell(1, 100)$ and

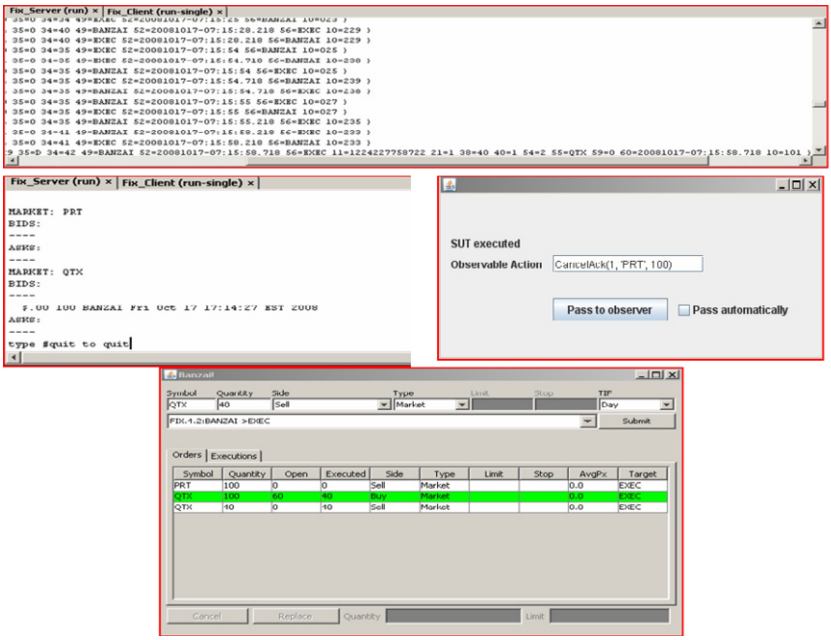


Fig. 6. Linking to the SUT

Table 1
Evolution of the state of an asynchronous testing process

| | Testing Tool | Execution Observer | $S_o$ |
|---|---|---|---|
| 1 | sell(1,100) | | {sellAck(1), sellReject(1), fullFill(1,100), partialFill(1,amt¡100) } |
| 2 | cancel(1) | | {sellAck(1), sellReject(1), fullFill(1,100), partialFill(1,amt¡100), cancelAck(1), cancelReject(1) } |
| 3 | | sell(1,100) cancel(1) | {sellAck(1), sellReject(1), fullFill(1,100), partialFill(1,amt¡100), cancelAck(1), cancelReject(1) } |
| 4 | | cancelAck(1) | {sellAck(1) } |
| 5 | buy(2,100) | | {sellAck(1), buyAck(2), buyReject(2), fullFill(2,100), partialFill(2,amt¡100) } |
| 6 | | sellAck(1) buy(2,100) | {buyAck(2), buyReject(2), fullFill(2,100), partialFill(2,amt¡100) } |
| 7 | sell(3,40) | | { buyAck(2), buyReject(2), sellAck(3), sellReject(3), fullFill(2,100), partialFill(2,amt¡100), fullFill(3,40), partialFill(3,amt¡40) } |
| 8 | | buyAck(2) sell(3,40) sellAck(3) | { fullFill(2,100), partialFill(2,amt¡100), fullFill(3,40), partialFill(3,amt¡40) } |
| 9 | | partialFill(2,40) fullFill(3,40) | {fullFill(2,60), partialFill(2,amt¡60) } |
| 10 | cancel(2) | | {fullFill(2,60), partialFill(2,amt¡60), cancelAck(2), cancelReject(2) } |
| 11 | | cancel(2) cancelAck(2) | { } |

$cancel(1)$ in the communication channel. Internally it does check that those actions are the ones the generator requested and discards them. Then, as shown in the fourth row, it observes also event $cancelAck(1)$ in the communication channel. This event triggers the update $S_o$ resulting in $S_o = \{sellAck(1)\}$. We can compare it with the previous set of expected observable actions and notice that event $cancelAck(1)$ has been removed from $S_o$ as it has already been executed. Additionally, events like $sellReject(1)$ have been removed as they cannot happen any more.

A similar scheme repeats until row 9 in which $fullFill(3, 40)$ and $partialFill(2, 40)$ are executed by the SUT and observer in the communication channel. This updates the set of expected observable events by removing any *fill* events with $id = 3$. The amount that can be filled for transaction with $id = 2$ is also modified to 60.

Finally, in the last two rows a cancellation for transaction with $id = 2$ is requested, observed and acknowledged, leaving $S_o$ empty and enabling the termination condition of the algorithms.

The execution history of this example can be read from the column of the Execution Observer module. This sequence of action executions represent a test case. This test case execution produces a *pass* verdict since it did not fail and the algorithms terminate.

## 7 Related work

Bhateja et al. [2] they review and analise some recent work in the area of testing for asynchronous system. They work on the settings of labelled transition systems, inspired by the earlier work of Tretmans [13]. They define, as we do, input and output actions (which we call observable and controllable, respectively). They use a queue semantics to "postpone" input actions and define the equivalence of two systems if one can be transformed into the other by means of postponing input actions. Finally they do a theoretical analysis of different testing equivalences. Our work, in contrast, uses a set semantics in order not only to postpone observable actions but also to deal with imperfect channels where communication delays can occur and where the order of messages is not guaranteed. In addition, we apply the theoretical concepts into an automated testing framework rather than perform theoretical analysis.

Different existing tools for automated testing use different formalisms to describe the systems under test. Our work has not focussed on the transformation of these models into the general event structures we use. Algorithms for generating an equivalent event structure from different formalisms have been developed elsewhere. Henigger [6] generates an equivalent prime event structure from a system of asynchronously communicating state machines. Nakata et al. [9], they link context-free processes specifications to the concept of symbolic event structures. They use the properties of event structures to derive the protocol specification of a distributed system. Herbreteau et al. [7] present an algorithm to generate labelled event struc-

tures from a well-structured LTS specification. In principle, the well-structured condition is necessary to deal with infinite states systems.

Several testing tools, among whom we can cite TGV [8] and TorX [1], have been used in experiments with distributed and synchronous systems. As the SmartMBT tool used in our work, TorX uses an on-the-fly approach. In [1] TorX was used to test a *Conference Protocol*. Contrarily to our work this experiment assumes the communication is reliable and message exchange can be modelled as (FIFO) queues. TGV, on the other hand, requires test cases to be serialised first. Consequently, losses concurrency and introduces unnecessary synchronisations between distributed testers.

Finally, our work is based on very well defined conformance testing theories and conformance relations such as IOCO [12]. However, the systems we are interested in are not always *input enabled*. This is, some inputs are only enabled by causality relations with other inputs or outputs. Event structures provide a natural way of expressing such relations. Other formalisms such as *IOLTS*[12], *trace automatas* and *concurrent transition systems* [11] have been used to model concurrent systems. There are no differences between these formalisms and event structures, for the purposes of this work. However, event structures provide the concept of a conflict relationship which, although not used in this work, we believe can be addressed in future work to increase the expression power of the input models.

## 8 Conclusion

In this paper we have shown our extensions to existing testing tools to allow automated testing for asynchronous systems. This extensions are based on sound theory available in the literature. We have shown via an example that these extensions can handle real systems such as an implementation of the FIX protocol. We believe that our choice of event structures as the underlying model jointly with an on-the-fly approach for test case generation and execution is a step forward preserving concurrency in asynchronous testing.

This work can be extended in several ways. On test sequence generation, for example, our algorithm proposes a random walk over the graph that represents the system. This approach does not guarantee any coverage for the generated sequences, so the question of when to stop testing is not answered. However, our approach can be extended to use some of the techniques for generating transition tours over the graph, such as Chinese Postman and others reviewed in [5].

## References

[1] Belinfante, A., J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw and L. Heerink, *Formal test automation: A simple experiment*, in: G. Csopaki, S. Dibuz and K. Tarnay, editors, $12^{th}$ *Int. Workshop on Testing of Communicating Systems* (1999), pp. 179–196.

[2] Bhateja, P., P. Gastin and M. Mukund, *A fresh look at testing for asynchronous communication*, in: *Automated Technology for Verification and Analysis*, number 4218 in Lecture Notes in Computer Science (2006), pp. 369 – 383.

[3] Boreale, M., R. D. Nicola and R. Pugliese, *Trace and testing equivalence on asynchronous processes*, Inf. Comput. **172** (2002), pp. 139–164.

[4] Castellani, I. and M. Hennessy, *Testing theories for asynchronous languages*, in: *In the Proc. of FSTTCS'98*, number 1530 in Lecture Notes in Computer Science (1998), pp. 90–101.

[5] Gargantini, A., *Conformance testing*, in: M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker and A. Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, Lecture Notes in Computer Science **3472**, Springer, 2005 pp. 87–111.

[6] Henniger, O., *On test case generation from asynchronously communicating state machines*, in: *Proceedings of the 10th International Workshop on Testing of Communicating Systems*, Cheju Island, South Korea, 1997.

[7] Herbreteau, F., G. Sutre and T. Q. Tran, *Unfolding concurrent well-structured transition systems*, in: O. Grumberg and M. Huth, editors, *Proc. of the 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS07)*, Lecture Notes in Computer Science **4424**, ETAPS (2007), pp. 706–720.
URL http://www.labri.fr/publications/mef/2007/HST07

[8] 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*, Int. J. Softw. Tools Technol. Transf. **7** (2005), pp. 297–315.

[9] Nakata, A., T. Higashino and K. Taniguchi, *Protocol synthesis from context-free processes using event structures*, in: *Proc. of the 5th International Conference on Real-Time Computing Systems and Applications*, Hiroshima, Japan, 1998, pp. 173–180.

[10] Nielsen, M., G. D. Plotkin and G. Winskel, *Petri nets, event structures and domains, part I*, Theoretical Computer Science **13** (1981), pp. 85–108.

[11] Stark, E. W., *Connections between a concrete and an abstract model of concurrent systems*, in: *In Fifth Conference on the Mathematical Foundations of Programming Semantics, Springer-Verlag. Lecture Notes in Computer Science* (1989), pp. 53–79.

[12] Tretmans, J., *Model based testing with labelled transition systems.*

[13] Tretmans, J., "A Formal Approach to Conformance Testing," Ph.D. thesis, University of Twente, Enschede, The Netherlands (1992).

[14] van Glabbeek, R. and G. Plotkin, *Event structures for resolvable conflicts*, in: *Proc. 29 th Int. Symp. Mathematical Foundation of Computer Science (MFCS04). Volume 3153 of Lect. Notes Comp. Sci* (2004), p. 550.

[15] Veanes, M., C. Campbell, W. Schulte and P. Kohli, *On-the-fly testing of reactive systems*, Technical Report MSR-TR-2005-05, Microsoft Research (2005).

[16] Winskel, G., *Event structure semantics for CCS and related languages*, in: *Proceedings of the 9th Colloquium on Automata, Languages and Programming* (1982), pp. 561–576.