Model-Based Testing and TorXakis – A Tutorial

September 15, 2021

TorXakis is an experimental tool for on-line model-based testing, founded in the **ioco**-testing theory for labelled transition systems. TorXakis is open source software with a BSD3 license that is freely available [58]. This tutorial gives a light-weight introduction to TorXakis, with the following sections, where Sect. 1, 4, 5, 6, and 7 concentrate on TorXakis itself, and Sect. 2 and 3 provide background information:

- Section 1: how to get started with TorXakis, installation, and running a first model-based test;
- Section 2: a discussion of model-based testing in general and the positioning of TorXakis;
- Section 3: a glimpse of the underlying theory, with references for further reading;
- Section 4: the main features of TorXakis;
- a couple of elaborated examples:
  - Section 5: the obligate *Hello World!* example,
  - Section 6: a simple *Job Dispatcher* system, and
  - Section 7: the well-known Dropbox file-synchronization service.

1 TorXakis: Getting Started

TorXakis is a tool for Model-Based Testing (MBT). This section gives step-by-step instructions for installation, for executing your first model-based test, and for detecting your first bug with TorXakis. As an example, we use an *integer queue*.

For model-based testing you first need the tool TorXakis; second, a System under Test (sut) that is a Java program implementing the *queue*; third, a model specifying the behaviour of the *queue*; fourth, a connection between the test tool and the sut, see Fig. 1; and then you can start running model-based tests.

1.1 TorXakis Installation

You can download and install TorXakis for your favourite operation system from the Downloads-page on the TorXakis website: `https://torxakis.org`. After installation you can run TorXakis in a terminal window, with

```
$ torxakis
```

You should get the TorXakis prompt:

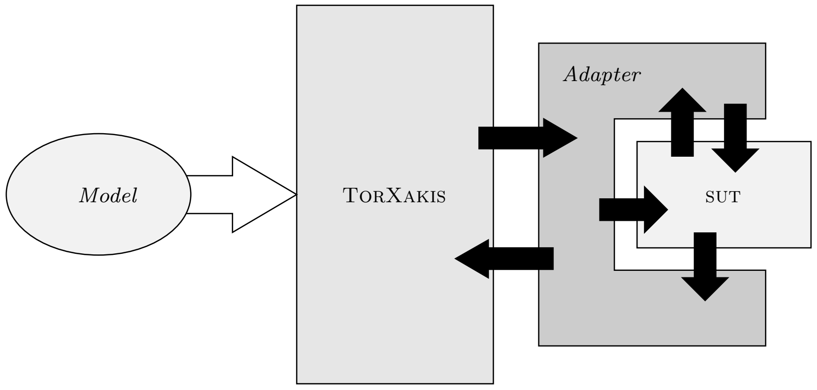

Figure 1: Test architecture.

```
TXS >>
```

after which you can use various TorXakis commands; try `h(elp)` to get an overview of possible commands, `q(uit)` to quit TorXakis, and try evaluating an expression, to check whether everything was successfully installed:

```
TXS >> eval 42-17
```

1.2 System Under Test

The second thing you need for MBT is an SUT. A couple of examples for TorXakis usage can be found on the Examples-page via the Documentation-tab on the TorXakis website `https://torxakis.org`. One of the examples is the *Queue*, which has various models and SUTs. There is a Java implementation of the *Queue* in `sut0/QueueServer0.java`, which will be our SUT in these instructions. The program is a *Queue*-implementation that offers its service via a plain old socket interface. It accepts input `Enq(x)` (Enqueue) to put integer value `x` in the queue, and it accepts input `Deq` (Dequeue) and then provides the first value from the queue as output; see Fig. 2.

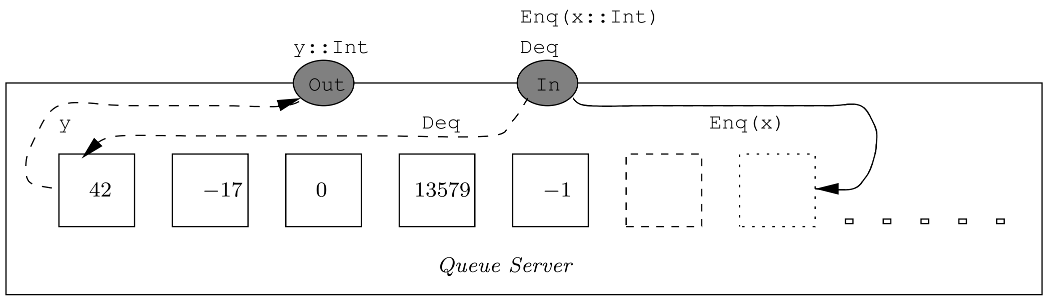

Figure 2: A Queue of integers.

To experiment with the SUT, compile it and run it, or use the precompiled version, and use the portnumber `7890`. To use the *Queue* we need to connect to it via the plain old socket interface using an application that can communicate via sockets, e.g., `telnet`, `nc` (`netcat`), or `putty`. Start the Java-*Queue* in one window and `telnet`, etc. in another, and connect them by choosing the same port number.

```
User Window                         SUT Window
$ telnet localhost 7890             $ java -jar QueueServer0.jar 7890
Trying 127.0.0.1...                 Waiting for tester
Connected to localhost.             Tester connected.
Escape character is '^]'.
                                    [  ]
Enq(42)                             [ 42 ]
Enq(-17)                            [ 42, -17 ]
Deq                                 [ -17 ]
42
```

1.3 Model

In the *Queue*-example, there are also a couple of TorXakis models, written in the TorXakis modelling language Txs. One of them is `Queue.txs`, You can view and edit the model in your favourite plain editor. The model specifies an unbounded, first-in-first-out *Queue* of integers. There are some comments in the file explaining the model; comments in Txs are either between {- and -}, or after `--` until end-of-line. The state-transition system of the queue model is graphically represented as a Txs state automaton, called `STAUTDEF`, in Fig. 3. In such a representation, the `STAUTDEF` declaration is textually described and the transitions are visualized as a graph.

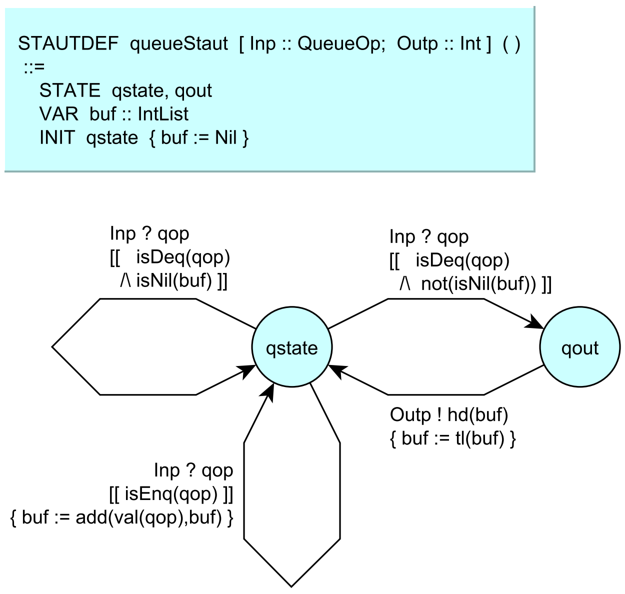
```
STAUTDEF  queueStaut  [ Inp :: QueueOp;  Outp :: Int ]  ( )
 ::=
    STATE  qstate, qout
    VAR  buf :: IntList
    INIT  qstate  { buf := Nil }
```

Figure 3: State automaton for the Queue.

You can copy the file `Queue.txs` to a new directory; also copy the file `.torxakis.yaml` that contains some TorXakis configuration information, to be modified later. Now you can use TorXakis to step through the model, i.e., use the `stepper`-command on the TorXakis-prompt, followed by the `step <n>` command to specify how many steps you wish to make through the transitions of the state-automaton. This will show a trace of possible of behaviour, i.e., a sequence of transitons, as it is described in the model. The result looks like below; 'looks like' because the sequence of actions and the integer values are randomly chosen, so your result might differ a bit.

```
$ torxakis Queue.txs

TXS >>  TorXakis :: Model-Based Testing

TXS >>  txsserver starting: "PC-21165.tsn.tno.nl" : 54888
TXS >>  Solver "z3" initialized : Z3 [4.8.5 - build hashcode b63a0e31d3e2]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["Queue0.txs"]
TXS >> stepper Queue
TXS >>  Stepper started
TXS >> step 7
TXS >>  .....1: NoDir: Act { { ( In, [ Enq(-1325) ] ) } }
TXS >>  .....2: NoDir: Act { { ( In, [ Enq(0) ] ) } }
TXS >>  .....3: NoDir: Act { { ( In, [ Enq(-1782) ] ) } }
TXS >>  .....4: NoDir: Act { { ( In, [ Enq(-90992) ] ) } }
TXS >>  .....5: NoDir: Act { { ( In, [ Enq(-75) ] ) } }
TXS >>  .....6: NoDir: Act { { ( In, [ Deq ] ) } }
TXS >>  .....7: NoDir: Act { { ( Out, [ -1325 ] ) } }
TXS >>  PASS
TXS >>
```

1.4 Model-Based Testing of the Queue

Now that we have a SUT – `QueueServer0.java` – and a model specifying the required behaviour of the SUT – `Queue.txs` –, we can start testing the SUT against its model. To test the Queue, run the SUT in one window and start TorXakis with the model as input, in another window. When TorXakis gives its prompt, start testing with `tester Queue Sut`, that is, the `tester`-command with `Queue` as model, i.e., the `MODELDEF` in the model file `Queue.txs`, and `Sut` as proxy to the SUT, i.e., the `CNECTDEF` in the model file. Upon `tester Queue Sut` TorXakis will connect directly to the SUT, so you do not need `telnet`, etc. Then the command `test 7` specifies how many test steps will be taken; you can easily try bigger numbers, e.g., `test 7777`. Now you have executed your first successful test with TorXakis!

```
$ torxakis Queue0.txs

TXS >>  TorXakis :: Model-Based Testing

TXS >>  txsserver starting: "PC-21165.tsn.tno.nl" : 54890
TXS >>  Solver "z3" initialized : Z3 [4.8.5 - build hashcode b63a0e31d3e2]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["Queue0.txs"]
TXS >> tester Queue Sut
TXS >>  Tester started
TXS >> test 7
TXS >>  .....1: In: Act { { ( In, [ Enq(-1953) ] ) } }
TXS >>  .....2: In: Act { { ( In, [ Deq ] ) } }
TXS >>  .....3: Out: Act { { ( Out, [ -1953 ] ) } }
TXS >>  .....4: In: Act { { ( In, [ Deq ] ) } }
TXS >>  .....5: In: Act { { ( In, [ Enq(-1) ] ) } }
TXS >>  .....6: In: Act { { ( In, [ Deq ] ) } }
TXS >>  .....7: Out: Act { { ( Out, [ -1 ] ) } }
TXS >>  PASS
TXS >>
```

1.5 A Queue Mutant

You have now tested the SUT `QueueServer0.java` against its model, but `QueueServer0.java` does not contain bugs (at least, as far as we know, but ... "testing can only show the presence of errors, never their absence" [24]). Detecting bugs is probably more rewarding for testers, so we added in the *Queue*-example three Queue *mutants*, small modifications in the Java program that may make the SUT buggy. These mutants are `sut1`, `sut2`, and `sut3`. You can test these SUT's with the same model to see whether you can detect (and explain?) the bugs.

1.6 Utilities

1.6.1 Notepad++ and Txs

Notepad++ is a free editor running in the Windows environment: `https://notepad-plus-plus.org`. *Syntax high-lighting* for Txs is available for *Notepad++*. Follow the installation instructions on: `https://github.com/TorXakis/SupportNotepadPlusPlus` to install the *Notepad++*-plugin for Txs.

1.6.2 yEd and Txs

Models represent state-transition systems, which can intuitively be visualized as graphs. *yEd* is a powerfull, freely available graph editor that can be used to edit and (automatically) layout graphs, and that runs on Windows, Unix/Linux, and macOS: `https://www.yworks.com/products/yed`. A translation from *yEd* to Txs is available. Follow the installation instructions on: `https://github.com/TorXakis/yed2stautdef` to install the application `yed2stautdef` that translates *yEd*-output to a state-automaton definition `STAUTDEF` in Txs.

For the *Queue*-example, a graph representing its state-transition system, is available in `Qstaut.graphml`; actually, it is the graph of Fig. 3. This graph has three nodes and four edges. The edges represent the transitions in the state-transition system. Two nodes represent states and one node gives the declaration of the `STAUTDEF`. The labels in the nodes representing states are the state names; the labels on the transitions specify actions in Txs-syntax. The declaration node gives the name of the state automaton, its channels with message types between `[` and `]`, and optionally some parameters between `(` and `)`. Moreover, there is the list of all states, the state variables with their types, and the initial state with initial values for the state variables. Nodes and edges can be formatted as wished (colour, shape, lining, shadow, ...); it does not matter for the transformation to Txs.

The graph edited in *yEd* shall be saved in *Trivial Graph Format* TGF (`*.tgf`). The application `yed2stautdef` transforms a file in TGF-format to a Txs-file:

```
$ yed2stautdef QueueGraph.tgf
```

The result is a `STAUTDEF` – a *State Automaton Definition* in the language Txs:

```
STAUTDEF queueStaut [ Inp :: QueueOp; Outp :: Int ] ( )
```

```
    ::=
      STATE
        qstate, qout
      VAR
        buf :: IntList
      INIT
        qstate { buf := Nil }
      TRANS
        qstate -> Inp ? qop [[ isDeq(qop) /\ not(isNil(buf)) ]] -> qout
        qout -> Outp ! hd(buf) { buf := tl(buf) } -> qstate
        qstate -> Inp ? qop [[ isEnq(qop) ]] { buf := add(val(qop),buf) } -> qstate
        qstate -> Inp ? qop [[ isDeq(qop) /\ isNil(buf) ]] -> qstate
  ENDDEF
```

A `STAUTDEF` can be included in a `.txs`-file, or the file can be used as additional input file for TorXakis; TorXakis allows multiple `.txs` input files. In Txs, a `STAUTDEF` can be used anywhere where a *process*, defined in a `PROCDEF`, can be used. Note that the graph should *also* be saved in the standard GRAPHML format (`.graphml`), because the TGF-format, as the name suggests, is a very trivial format, which does not preserve graph layout and formatting. So, next time when you continue editing with *yEd* use the `.graphml`-file and not the `.tgf`-file.

The application `yed2stautdef` just transforms the `.tgf`-file and does not check any syntax or static semantics. Checking is only done on the `.txs`-file, where error messages might appear. Finding the corresponding error spot in the `.graphml`-file is, for the moment, left to the user.

2 Model-Based Testing

Software Testing Software quality is a matter of increasing importance and growing concern. Systematic testing plays an important role in the quest for improved quality and reliability of software systems. Software testing, however, is often an error-prone, expensive, and time-consuming process. Estimates are that testing consumes 30-50% of the total software development costs. The tendency is that the effort spent on testing is still increasing due to the continuing quest for better software quality, and the ever growing size and complexity of systems. The situation is aggravated by the fact that the complexity of testing tends to grow faster than the complexity of the systems being tested, in the worst case even exponentially. Whereas development and construction methods for software allow the building of ever larger and more complex systems, there is a real danger that testing methods cannot keep pace with these construction and development methods, so that these new systems cannot sufficiently fast and thoroughly be tested anymore. This may seriously hamper the development and testing of future generations of software systems.

Model-Based Testing. Model-Based Testing (MBT) is one of the technologies to meet the challenges imposed on software testing. With MBT a System Under Test (SUT) is tested against an abstract model of its required behaviour. This model serves as the system specification and is the starting point for testing. It prescribes what the SUT should, and what it should not do, that is, the behaviour of the SUT shall conform to the behaviour prescribed in the model. The model itself is assumed to be correct and valid; it is not the direct subject of testing or validation.

The main virtue of MBT is that the model is a perfect basis for the generation of test cases, allowing test automation that goes well beyond the mere automatic execution of manually crafted test cases. MBT allows for the algorithmic generation of large amounts of test cases, including test oracles for the expected results, completely automatically, from the model of required behaviour. Moreover, if this model is valid, i.e., expresses precisely what the system under test should do, all these generated tests are provably valid, too.

From an industrial perspective, model-based testing is a promising approach to detect more bugs faster and cheaper. The current state of practice is that test automation mainly concentrates on the automatic execution of tests, but that the problem of test generation is not addressed. Model-based testing aims at automatically generating high-quality test suites from models, thus complementing automatic test execution.

From an academic perspective, model-based testing is a formal-methods approach to testing that complements formal verification and model checking. Formal verification and model checking intend to show that a system has specified properties by proving that a model of that system satisfies these properties. Thus, any verification is only as good as the validity of the model on which it is based. Model-based testing, on the other hand, starts with a (verified) model, and then aims at showing that the real, physical implementation of the system behaves in compliance with this model. Due to the inherent limitations of testing, such as the limited number of tests that can be performed in a reasonable time, testing can never be complete: "testing can only show the presence of errors, not their absence" [24].

Benefits of model-based testing. Model-based testing makes it possible to generate test cases automatically, enabling the next step in test automation. It makes it possible to generate more, longer, and more diversified test cases with less effort, whereas, being based on sound algorithms, these test cases are provably valid. Since a model specifies both the possibe stimuli (inputs) to the SUT and the allowed responses (outputs), the generated test cases contain inputs to be provided to the SUT as well as outputs expected from the SUT, which leads to better test oracles and less misinterpretation of test results.

Creating models for MBT usually already leads to better understanding of system behaviour and requirements and to early detection of specification and design errors. Moreover, constructing models for MBT paves the way for other model-based methods, such as model-based analysis, model checking, and simulation, and it forms the natural connection to model-based system development that is becoming an important driving force in the software industry.

Test suite maintenance, i.e., continuously adapting test cases when systems are modified, is an important challenge of any testing process. In MBT, maintenance of a multitude of test cases is replaced by maintenance of a model. Also diagnosis, i.e., localizing the fault when a failure is detected, is facilated through model-based diagnostic analysis. Finally, various notions of (model-) coverage can be automatically computed, expressing the level of completeness of testing, and allowing better selection of test cases.

Altogether, MBT is a promising approach to detect more bugs faster and cheaper, and thus to improve the quality and reliability of the system under test.

Sorts of model-based testing. There are different kinds of testing, and thus of model-based testing, depending on the kind of models being used, the quality characteristics being tested, the level of formality involved, the degree of accessibility and observability of the system being tested, and the kind of system being tested. Here, we consider model-based testing as *formal*, *specification-based*, *active*, *black-box*, *functionality testing* of *reactive systems*.

It is *testing*, because it involves checking some properties of the SUT by systematically performing experiments on the real, running SUT. as opposed to, e.g., formal verification, where properties are checked on a model of the system. The kind of properties being checked are concerned with *functionality*, i.e., testing whether the system correctly does what it should do in terms of correct responses to given stimuli. as opposed to, e.g., performance, usability, reliability, or maintainability properties. Such classes of properties are often referred to as *quality characteristics* [41].

We do *specification-based*, *black-box* testing. The SUT is seen as a black box without internal detail, which can only be accessed and observed through its external interfaces, as opposed to white-box testing, where the internal structure of the SUT, i.e., the code, is the basis for testing. The externally observable behaviour of the system is compared with what has been specified in the model.

The testing is *active*, in the sense that the tester controls and observes the SUT in an active way by giving stimuli and triggers to the SUT, and observing its responses, as opposed to passive testing, or monitoring. Our SUTs are *dynamic, data-intensive, reactive systems.* Reactive systems react to external events (stimuli, triggers, inputs) with output events (responses, actions, outputs). In dynamic systems, outputs depend on inputs as well as on the system state. Data-intensive means that instances of complex data structures are communicated in inputs and outputs, and that state transitions may involve complex computations and constraints.

Finally, we deal with *formal testing*: the model, which serves as specification prescribing the desired behaviour is written in some formal language with precisely defined syntax and semantics. Moreover, there is a formal, well-defined theory underpinning these models, SUTs, tests, and their relations, in particular, the correctness (conformance) of SUTs with respect to models, and the validity of tests with respect to models. This enables formal reasoning about *soundness* and *exhaustiveness* of test generation algorithms and the generated test suites, i.e., that tests exactly test what they should test.

In another form of model-based testing, called *statistical model-based testing*, models do not prescribe required behaviour of the SUT, but they describe how users use a system. Such models are called *statistical usage profiles*, operational profiles, or customer profiles. The idea is that tests are selected based on the expected usage of the SUT, so that behaviours that are more often used, are more thoroughly tested [55]. Such models are derived from usage information such as usage logs. Statistical model-based testing enables the comprehensive field of statistics to be used with the goal of assessing the reliability of systems.

Theory for model-based testing. A theory for model-based testing must, naturally, first of all define the models that are considered. The modelling formalism determines the kind of properties that can be specified, and, consequently, the kind of properties for which test cases can be generated. Secondly, it must be precisely defined what it means for an SUT to conform to a model. Conformance can be expressed using an *implementation relation*, also called *conformance relation* [17]. Since the SUT is considered as a black box, its behaviour is unknown and we cannot construct a model that precisely describes the behaviour of the SUT, yet, we do assume that such a model, though unknown, exists in a domain of implementation models. This assumption is commonly referred to as the *testability hypothesis*, or *test assumption* [31]. The testability hypothesis allows reasoning about SUTs as if they were formal models, and it makes it possible to define the implementation relation as a formal relation between the domain of specification models and the domain of implementation models. Soundness of test suites, i.e., do all correct SUTs pass, and exhaustiveness, i.e., do all incorrect SUTs fail, are defined with respect to an implementation relation.

In the domain of testing reactive systems there are two prevailing 'schools' of formal model-based testing. The oldest one uses Mealy-machines, also called finite-state machines (FSM); see [18, 47, 51]. Here, we concentrate on the other one that uses *labelled transition systems* (LTS) for modelling. A labelled transition system is a structure consisting of states with transitions, labelled with actions, between them. The states model the system states; the labelled transitions model the actions that a system can perform. There is a rich and well-understood theory for MBT with LTS, which is elaborated in Sect. 3. Other approaches to MBT for non-reactive systems include abstract-data-type based testing [7] and *property-based testing*, of which the tool QUICKCHECK is the prime example [19]. Originally developed for HASKELL, property-based testing is now applied for many languages.

Labelled transition systems form a well-defined semantic basis for modelling and model-based testing, but they are not suitable for writing down models explicitly. Typically, realistic systems have more states than there are atoms on earth (which is approximately 10^{50}) so an explicit representation of states is impossible. What is needed is a language to represent large labelled transition systems. *Process algebras* have semantics in terms of labeled transition systems, they support different ways of composition such as choice, parallelism, sequencing, etc., and they were heavily investigated in the eighties [50, 39, 42]. They are a good candidate to serve as a notation for LTS models.

Model-based testing challenges. Software is anywhere, and ever more systems depend on software: software controls, connects, and monitors almost every aspect of systems, be it a car, an airplane, a pacemaker, or a refrigerator. Consequently, overall system quality and reliability are more and more determined by the quality of the embedded software. Typically, such software consists of several million lines of code, with complex behavioural control-flow as well as intricate data structures, with distribution and a lot of parallelism, having complex and heterogeneous interfaces, and controlling diverse, multidisciplinary processes. Moreover, this software often comes in many variants with different options and for different platforms. It is continuously evolving and being modified to adapt to different environments and new user requirements, while increasing in size, complexity, connectivity, and variability. Software is composed into larger systems and systems-of-systems, whereas system components increasingly originate from heterogeneous sources: there can be legacy, third-party, out-sourced, off-the-shelf, open source, or newly developed components.

For model-based testing, these trends lead to several challenges. First, the size of the systems implies that making complete models is often infeasible so that MBT must deal with partial and under-specified models and abstraction, and that partial knowledge and uncertainty cannot be avoided. Secondly, the combination of complicated state-behaviour and intricate input and output-data structures, and their dependencies, must be supported in modelling. Thirdly, distribution and parallelism imply that MBT must deal with concurrency in models, which introduces additional uncertainty and non-determinism. In the fourth place, since complex systems are built from sub-systems and components, and systems themselves are more and more combined into systems-of-systems, MBT must support compositionality, i.e., building complex models by combining simpler models. Lastly, since complexity leads to an astronomical number of potential test cases, test selection, i.e., how to select those tests from all potential test cases that can catch most, and most important failures, within constraints of testing time and budget, is a key issue in model-based testing.

In short, to be applicable to testing of modern software systems, MBT shall support partial models, under-specification, abstraction, uncertainty, state & data, concurrency, non-determinism, compositionality, and test selection. Though several academic and commercial MBT tools exist, there are not that many tools that support all of these aspects.

Model-based testing tools. Model-based testing activities are too laborious to be performed completely manually, so, for MBT to be effective and efficient, tool support is necessary. A large number of MBT tools exist, as a Web-search will immediately show. TorXakis is one of these MBT tools.

TorXakis is a proof-of-concept, research tool that is being developed by the Radboud University Nijmegen, the University of Twente, and ESI (TNO) in the Netherlands. It is an on-line (on-the-fly) MBT tool for formal, specification-based, active, black-box, functionality testing of reactive systems, rooted in the **ioco**-testing theory for labelled-transition systems [61, 62]. It implements the **ioco**-test generation algorithm for symbolic transition systems [30], and it uses a process-algebraic modelling language Txs inspired by the language LOTOS [11, 42], which is supplemented with an algebraic data-type specification formalism, for which rewriting and SMT solvers are used for calculation and manipulation [22]. Moreover, TorXakis deals with most of the challenges posed in the previous paragraph: it supports modelling of state-based control flow together with complex data, it deals with non-determinism, abstraction, partial models and under-specification, concurrency, and composition of complex models from simpler models. TorXakis supports state & data but no probabilities, real-time, or hybrid systems. Test selection is primarily random, but guidance can be provided using *test purposes*. TorXakis is an experimental MBT tool, used in (applied) research, education, and industrial case studies and experiments. TorXakis currently misses good usability, scalability does not always match the requirements of complex systems-of-systems, and more sophisticated test selection strategies are necessary but these are being investigated [14].

Future developments. Current MBT algorithms and tools can potentially generate many more tests from a model than can ever be executed. Consequently, *test selection* is one of the major research issues in model-based testing. Test selection concerns the problem of finding criteria for selecting from the astronomical number of potential test cases those tests that have the largest chance of detecting most, and the most important bugs, with the least effort. Random approaches, which are often used for small systems, do not suffice for large and complex systems: the probability of completely randomly selecting an important test case within the space of all possible behaviours converges to zero. At the other end from random there is the explicit specification of test purposes, i.e., a tester specifies explicitly what she wishes to test, but that requires a lot of manual effort, and, moreover, how should the tester know what to test. Different approaches have been identified for determining what the "most important behaviours" are, such as testing based on system requirements, code coverage, model coverage, risk analysis, error-impact analysis, or expected user behaviour (statistical usage profiles, or operational profiles).

Related to apriori test selection, is aposteriori coverage, quality, and confidence in the tested system. Since exhaustive testing is practically impossible, the question pops up what has been achieved after testing: can the coverage of testing, the quality of the tested SUT, or the confidence in correct functioning of the SUT, somehow be formalized and quantified? It is not to be expected that these fundamental research questions will soon be completely solved.

MBT is an interesting technique once a model of the SUT is available. Availability of behavioural models, however, is one of the issues that currently prohibits the widespread application of MBT. In the first place, there is the question of making and investing in models: there is reluctance against investing in making models, being considered as yet another software artifact. Secondly, mastering the art of behavioural modeling requires education and experience that is not always available. Thirdly, the information necessary to construct a model, in particular for legacy, third-party, or out-sourced systems or components, is not always (easily) available.

These issues lead to the question whether models can be generated automatically, e.g., for use in regression testing or testing systems after refactoring. Model generation from an SUT, a kind of black-box reverse engineering, (re)constructs a model by observing the behaviour of the SUT, either passively from system logs, or actively by performing special tests. This activity is called *model learning*, also known as test-based modeling, automata learning, or grammatical inference, and it is currently a popular research topic [65].

New software testing methodologies are needed if testing shall keep up with software development and meet the challenges imposed on it, otherwise we may not be able to test future generations of systems. systems. Model-based testing may be one of them.

3 Model-Based Testing with Labelled Transition Systems

Labelled transition systems. Labelled transition systems (LTS) and its variants constitute a powerful semantic model for describing and reasoning about dynamic, reactive systems. An LTS is a structure consisting of states with transitions, labelled with actions, between them. The states model the states of the system; the labelled transitions model the actions that a system can perform. Actions can be inputs, outputs, or internal steps of the system. LTS-based testing theory has developed over the years from a theory-oriented approach for defining LTS equivalences to a theory that forms a sound basis for real testing and industrially viable testing tools. In this section we first sketch the evolution of LTS-based testing theory and then describe to what extent TorXakis uses this theory.

Testing equivalences. Testing theory for LTS started with using testing to formalize the notion of behavioural equivalence for LTS. Two LTSs show equivalent behaviour if there is no test that can observe the difference between them. By defining appropriate formalizations for test and observation this led to the theory of testing equivalences and preorders for LTS [23]. Different equivalences can then be defined by choosing different formalizations of test and observation: more powerful testers lead to stronger equivalences, and the other way around. In the course of the years, many such variations were investigated, with testers that can observe the occurrence of actions, the refusal of actions, or the potentiality of doing actions, testers that can undo actions, that can make copies of the system state, or that can repeat tests indefinitely. Comparative concurrency semantics systematically compares these and other equivalences and preorders defined over LTS [1, 32, 33, 46, 53]. Crucial in these equivalences is the notion of *non-determinism*, i.e., that after doing an action in an LTS the subsequent state is not uniquely determined. For deterministic systems almost all equivalences coincide [28].

Test generation. Whereas the theory of testing equivalences and preorders is used to define semantic relations over LTS using all possible tests, actual testing turns this around: given an LTS s (the specification) and a relation **imp** over LTS (the implementation relation), determine a (minimal) set of tests $\Pi_{\mathbf{imp}}(s)$ that characterizes all implementations i with i **imp** s, i.e., i passes $\Pi_{\mathbf{imp}}(s)$ iff i **imp** s.

First steps towards systematically constructing such a test suite (sets of tests) from a specification LTS led to the *canonical tester* theory for the implementation relation **conf** [16]. The intuition of **conf** is that after traces, i.e., sequences of actions, that are explicitly specified in the specification LTS, the implementation LTS shall not unexpectedly refuse actions, i.e., the implementation may only refuse a set of actions if the specification can refuse this set, too. This introduces *under-specification*, in two ways. First, after traces that are not in the specification LTS, anything is allowed in the implementation. Second, the implementation may refuse less than the specification.

Inputs and outputs. The canonical tester theory and its variants make an important assumption about the communication between the SUT and the tester, viz. that this communication is synchronous and symmetric. Each communication event is seen as a joint action of the SUT and the tester, inspired by the parallel composition in process algebra. This also means that both the tester and the SUT can block the communication, and thus stop the other from progressing. In practice, however, it is different: actual communication between an SUT and a tester takes place via inputs and outputs. Inputs are initiated by the tester, they trigger the SUT, and they cannot be refused by the SUT. Outputs are produced by the SUT, and they are observed and cannot be refused by the tester.

A first approach to a testing theory with inputs and outputs was developed by interpreting each action as either input or output, and by modelling the communication medium between SUT and tester explicitly as a queue [63]. Later this was generalized by just assuming that inputs cannot be refused by the SUT – the SUT is assumed to be *input-enabled*, i.e., in each state there is a transition for all input actions – and outputs cannot be refused by the tester, akin to I/O-automata [48]. Adding these assumptions to the concepts of the canonical tester theory and **conf** – refusal sets of the implementation shall be refusal sets of the specification, but only for explicitly specified traces – leads to a new implementation relation that was coined **ioconf** [60]. The assumptions that the SUT cannot refuse inputs and the tester cannot refuse outputs makes that the only relevant refusal that remains is refusing all possible outputs by the SUT, which is called *quiescence* [64]. Intuitively, quiescence corresponds to observing that there is no output of the SUT, which is an important observation in testing theory as well as in practical testing.

Implementation relation ioco. In **ioconf** the test will stop after observing quiescence, i.e., during each test run quiescence occurs at most once, as the last observation. Phalippou noticed that in practical testing quiescence is observed as a time-out during which no output from the SUT is observed, and that after such a time-out testing continues with providing a next input to the SUT, so that quiescence can occur multiple times during a test run [52]. Inspired by this observation, *repetitive quiescence* was added to **ioconf**, leading to the implementation relation **ioco** (**i**nput-**o**utput-**co**nformance) [61, 62]. Theoretically, **ioco** is akin to failure-trace preorder with inputs and outputs [46]. Intuitively, **ioco** expresses that an SUT conforms to its specification if the SUT never produces an output that cannot be produced by the specification in the same situation, i.e., after the same trace. *Quiescence* is treated as a special, virtual output, actually expressing the absence of real outputs, which is observed in practice as a time-out during which no output from the SUT is observed. A small modification to **ioco** is the weaker implementation relation **uioco** [8], where not the outputs after all traces are considered, but only after those traces where inputs in the trace cannot be refused. The relation **uioco** was shown to enjoy much nicer mathematical properties and to deal more accurately with under-specification [43].

The **ioco** and **uioco**-implementation relations support partial models, under-specification, abstraction, and non-determinism. The testability hypothesis is that an SUT is assumed to be modelled as an *input-enabled* LTS, that is, any input to the implementation is accepted in every state. Specifications are not necessarily input-enabled. Inputs that are not accepted in a specification state are considered to be underspecified: no behaviour is specified for such inputs, implying that any behaviour is allowed in the SUT. Models that only specify behaviour for a small, selected set of inputs are partial models. Abstraction is supported by modelling actions or activities of systems as internal steps, without giving any details. Non-deterministic models may result from such internal steps, from having transitions from the same state labelled with the same action, or having states with multiple outputs (output non-determinism). Non-determinism leads to having a set of possible, expected outputs after a sequence of actions, and not just a single expected output. The SUT is required to implement at least one of these outputs, but not all of them, thus supporting implementation freedom.

For **ioco** and **uioco**-testing, there are test generation algorithms that are proved to be *sound* – all **ioco**/**uioco**-correct SUTs pass all generated tests – and *exhaustive* – all **ioco**/**uioco**-incorrect SUTs are eventually detected by some generated test. Consequently, the **ioco** and **uioco**-testing theory constitutes, on the one hand, a well-defined theory of model-based testing, whereas, on the other hand, it forms the basis for various practical MBT tools. In particular, the implementation relation **ioco** is the basis for a couple of MBT tools, such as TGV [44], the AGEDIS TOOL SET [35], TORX [6], JTORX [5], Uppaal-Tron [38], TESTOR [49], Axini Test Manager (ATM) [3, 9], and TORXAKIS.

A couple of variations have been proposed for **ioco** and **uioco**, such as **mioco** for multiple input and output channels [37], **wioco** that diminishes the requirements on input enabledness [66], various variants of timed-**ioco** [15, 38, 45], **qioco** for quantitative testing [10], and **sioco** for LTS with data [29, 30].

Data. The **ioco**/**uioco**-testing theory for labelled transition systems mainly deals with the dynamic aspects of system behaviour, i.e., with state-based control flow. The static aspects, such as data structures, their operations, and their constraints, which are part of almost any real system, are not covered. *Symbolic Transition Systems* (STS) add (infinite) data and data-dependent control flow, such as guarded transitions, to LTS, founded on first order logic [29, 30]. Symbolic **ioco** (**sioco**) lifts **ioco** to the symbolic level. The semantics of STS and **sioco** is given directly in terms of LTS; STS and **sioco** do not add expressiveness but they provide a way of representing and manipulating large and infinite transition systems symbolically.

TorXakis TORXAKIS is rooted in the **ioco**-testing theory for labelled transition systems. Its implementation relation is **ioco** and the testability hypothesis is that an SUT is assumed to be modelled as an input-enabled LTS. Test generation is sound for **ioco** and in the exhaustive, i.e., any non-conforming SUT will eventually, after unbounded time, be detected. TORXAKIS implements the **ioco**-test generation algorithm for symbolic transition systems, and it uses a process-algebraic modelling language TXS inspired by the language LOTOS [11, 42], which is supplemented with an algebraic data-type specification formalism.

4 TorXakis: A Model-Based Testing Tool

TorXakis is a tool for model-based testing. This section gives a high-level overview of TorXakis. The next sections will illustrate TorXakis with a couple of examples. TorXakis is open source software and is freely available under a BSD3 license [58].

4.1 Basics

Features. TorXakis implements the **ioco**-testing theory for labelled transition systems. More specifically, it implements test generation for symbolic transition systems (STS) following the on-the-fly **sioco**-test generation algorithm described in [29]. This means that conformance in TorXakis is precisely defined by the **ioco**-implementation relation, the testability hypothesis is that SUTs behave as input-enabled labelled transition systems, and test generation is sound and, in the limit, exhaustive. Being based on the **ioco**-theory, TorXakis supports uncertainty and abstraction through non-determinism, and partial and under-specification.

TorXakis emphasizes *formal*, *specification-based*, *active*, *black-box* model-based testing of *functionality* of *dynamic, data-intensive, reactive systems.* Reactive systems react to external events (stimuli, triggers, inputs) with output events (responses, actions, outputs) [54]. In dynamic systems, outputs depend on inputs as well as on the system state. Data-intensive means that instances of complex data structures are communicated in inputs and outputs, and that state transitions may involve complex computations and constraints.

TorXakis is an on-the-fly (on-line) MBT tool which means that it combines test generation and test execution: generated test steps are immediately executed on the SUT and responses from the SUT are immediately checked and used when calculating the next step in test generation.

Currently, only random test selection is supported, i.e., TorXakis chooses a random action among the possible inputs to the SUT in the current state. This involves choosing among the transitions of the STS and choosing a value from the (infinite, constrained) data items attached to the transition. The latter involves constraint solving. To direct and set goals for testing, the selection can be restricted using user-specified *test purposes* [67].

TorXakis is an experimental MBT tool, used in research, education, and some case studies and experiments in industry. TorXakis currently misses good usability, scalability does not always match the requirements of complex systems, and test selection is still mainly random, but more sophisticated selection strategies are being investigated [12, 13]. TorXakis does not support probabilities, real-time, or hybrid properties in system models.

Modelling. Labelled transition systems or symbolic transition systems form a well-defined semantic basis for modelling and model-based testing, but they are not directly suitable for writing down models explicitly. Typically, realistic systems have more states than there are atoms on earth (which is estimated to be approximately 10^{50}) so an explicit representation of states is impossible. What is needed is a language to represent large labelled transition systems. *Process algebras* have semantics in terms of labeled transition systems, they support different ways of composition, such as choice, parallelism, concurrency, sequencing, etc., and they were heavily investigated in the eighties [50, 39, 42]. They are a good candidate to serve as a notation for LTS models.

TorXakis uses its own process-algebraic language Txs (pronounced *tèxès*) to express models. The language is strongly inspired by the process-algebraic language Lotos [11, 42], and incorporates ideas from Extended Lotos [16] and mCRL2 [34], combined with plain state-transition systems. The semantics is based on STS, which in turn has semantics in LTS. Having its roots in process algebra, the language is compositional. It has several operators to combine transition systems: sequencing, choice, parallel composition with and without communication, interrupt, disable, and abstraction (hiding). Communication between processes can be multi-way, and actions can be built using multiple labels.

Since symbolic transition systems (STS) combine state-based control flow with possibly infinite, complex data structures [30], the process-algebraic part is complemented with a data specification language based on algebraic data types (ADT) and functions like in functional languages. In addition to user-defined ADTs, predefined data types such as booleans, unbounded integers, and strings are provided.

Implementation. TorXakis is based on the model-based testing tools TorX [6] and JTorX [5]. The main additions are data specification and manipulation with algebraic data types, and its own, well-defined modelling language. Like TorX and JTorX, TorXakis generates tests by first unfolding the process expressions from the model into a *behaviour tree*, on which primitives are defined for generating test cases. Unlike TorX and JTorX, TorXakis does not unfold data into all possible concrete data values, but it keeps data symbolically. Unfolding of process expressions is similar to the LOTOS simulators HIPPO [27, 59] and SMILE [26].

In order to manipulate symbolic data and solve constraints for test-data generation, TorXakis uses SMT solvers (Satisfaction Modulo Theories) [22]. Currently, Z3 and CVC4 are used via the SMT-LIBv2.5 standard interface [21, 4, 20]. Term rewriting is used to evaluate data expressions and functions.

The well-defined process-algebraic basis with **ioco** semantics makes it possible to perform optimizations and reductions based on equational reasoning with testing equivalence, which implies **ioco**-semantics.

The core of TorXakis is implemented in the functional language Haskell [36], while parts of TorXakis itself have been tested with the Haskell MBT tool QuickCheck [19].

Innovation. Compared to other model-based testing tools TorXakis deals with some of the important challenges posed in Sect. 2: it offers support for test generation from non-deterministic models, it deals with abstraction, partial models and under-specification, it supports concurrency and parallelism, it enables composition of complex models from simpler models, and it combines constructive modelling in transition systems with property-oriented specification via data constraints.

4.2 Usage

In order to use TorXakis, we need a *System Under Test* (SUT), a *model* specifying the required and allowed behaviour of the SUT, and an *adapter* (also called test harness, wrapper, testing glue, or test scaffolding) to connect the actual SUT to the test tool TorXakis; see Fig. 4.

System under test. The SUT is the actual program, compoenent, or system that we wish to test. The TorXakis view of an SUT is a black-box communicating with messages on its interfaces. Interfaces can be distinguished as either an input interface, where the environment takes the initiative and the system always accepts the action (input-enabledness; black arrows going into the SUT in Fig. 4), or an output interface, where the system takes the initiative and the environment always accepts (input-enabledness of the environment for the output actions of the system; black arrows going out of the SUT in Fig. 4). Interfaces are modelled as *channels* in Txs. So, an input is a message sent by the tester to the SUT on an input channel; an output is the observation by the tester of a message from the SUT on an output channel.

An instance of behaviour of the SUT is a possible sequence of input and output actions. The goal of testing is to compare the actual behaviour that the SUT exhibits with the behaviour specified in the model.

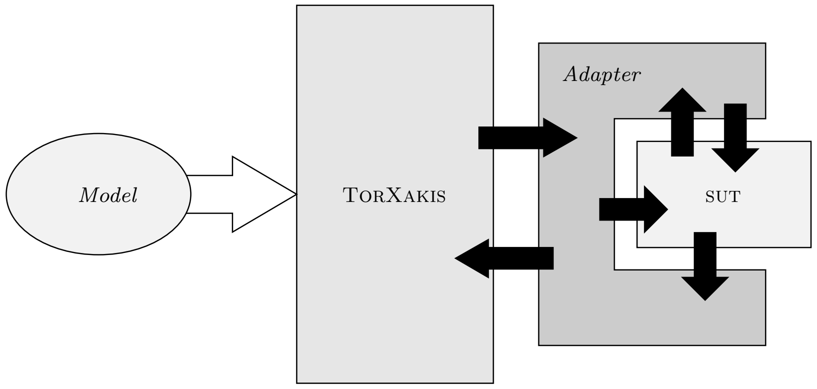

Figure 4: Test architecture.

Technically, channels are implemented as plain old sockets where messages are line-based strings, or string-encodings of some typed data. So, technically, the TorXakis view of an SUT is a black-box communicating with strings on a couple of sockets.

Model. The model is written in the TorXakis modelling language Txs. A model consists of a collection of definitions. There are channel, data-type, function, constant, process, and state-automaton definitions, which are contained in one or multiple files. In addition, there are some testing-specific definition: connections and en/decodings. A connection definition defines how TorXakis will connect to the SUT for test execution. It can been as a proxy for the SUT: it specifies the binding of abstract channels in the model to concrete sockets. En/decodings specify the mapping of abstract messages (ADTs) to strings and vice versa. The next sections will explain the details of modelling using some examples.

The model shall specify the allowed behaviour of the SUT, i.e., the allowed sequences of input and output actions exchanged on its channels. The basic structure to describe the allowed sequences is a state-transition system with data, called *state-automaton* in Txs. These state-transition systems can be composed using combinators (process-algebraic operators), so that complex state-transition systems can be constructed from simple ones. Combinators include sequencing of transition systems, choice, guards, parallelism, synchronization, communication, interrupt, disable, and abstraction (hiding of actions).

The data items used in these state-transition systems are either of standard data types such as integer, boolean, or string, or they are user-defined algebraic data-type definitions. Also functions and constants over data can be defined.

Adapter. TorXakis communicates with the SUT via sockets, so either the SUT must offer a socket interface – which a lot of real-life SUTs don't do – or the SUT must be connected via an adapter, wrapper, test harness, or glueing software, that interfaces the SUT to TorXakis, and that transforms the native communication of the SUT to the socket communication that TorXakis requires. Usually, such an adapter must be manually developed. Sometimes it is simple, e.g., transforming standard I/O into socket communication using standard (Unix) tools like `netcat` or `socat`. Sometimes, building an adapter can be quite cumbersome, e.g., when the SUT provides a GUI. In this case tools like Selenium [56] or Sikuli [57] may be used to adapt a GUI or a web interface to socket communication. An adapter is not specific for MBT but is required for any form of automated test execution. If traditional test automation is in place then this infrastructure can quite often be reused as adapter for MBT.

Even when a SUT communicates over sockets, there is still a caveat: sockets have asynchronous communication whereas models and test generation assume synchronous communication. This may lead to race conditions if a model offers the choice between an input and an output. If this occurs the asynchronous communication of the sockets must be explicitly modelled, e.g., as queues in the model.

Testing. Once we have an SUT, a model, and an adapter, we can use TorXakis to run tests. The tool performs on-the-fly testing of the SUT by automatically generating test steps from the model and immediately executing these test steps on the SUT, while observing and checking the responses from the SUT. A test case may consist of thousands of such test steps, which makes it also suitable for reliability testing, and it will eventually lead to a verdict for the test case.

Other features. Other functionality of TorXakis includes calculation of data values, constraint solving for data variables, exploration of a model without connecting to an SUT (closed simulation), and simulation of a model in an environment, i.e., simulation while connected to the outside world (open simulation).

5 Hello World!

Traditionally, the first program, in this case the first model, made in a new language is the famous *Hello World!* program. Since the original *Hello World!* program is rather easy to test, we take a slight variation: our *Hello World!* initially prints `Hello World!`, and then continues with waiting for a ⟨name⟩ as input, after which it outputs "Hello ⟨name⟩!", and then these two last actions are repeated.

In order to use TorXakis to test *Hello World!* we need a *System Under Test* (SUT), i.e., an executable program showing the above described *Hello World!* behaviour, a *model* in the Txs-language specifying the allowed behaviour of the *Hello World!* system, and a test *adapter* to connect the SUT to TorXakis; see Fig. 4. We can then use TorXakis to test whether the behaviour of the SUT complies with the behaviour specified in the model.

Hello World SUT Our SUT is an executable program that is claimed to behave according the *Hello World!* description given above. Our task is to test whether this SUT indeed behaves as prescribed. The SUT can be implemented in any language: we consider black-box testing, which means that we only consider its input-output behaviour. Our *Hello World!* SUT is a C-program with a simple line-oriented user interface that communicates via standard input/output; see the *Hello World!* example on the Examples-page via the Documentation-tab on the TorXakis website `https://torxakis.org`.

```
$ gcc -o HelloWorld HelloWorld.c
$ ./HelloWorld
Hello World!
Jan
Hello Jan!
Pierre
Hello Pierre!
...
```

Hello World Adapter We now consider a test adapter for *Hello World!*. Since the SUT commmunicates via standard input/output and TorXakis communicates via plain old sockets, this means that we have to convert standard input/output communication to socket communication. In a Linux-like environment this can be done using standard utilities like `netcat nc` or `socat`:

```
$ socat TCP4-LISTEN:7890 EXEC:"./HelloWorld"
```

Using `socat`, a socket-server connection is opened on port 7890. Data on this connection is forwarded to/from standard input/output for executable `HelloWorld`. So, `socat` constitutes the test adapter for *Hello World!*. The TorXakis view of the *Hello World!* program is a black box receiving names, i.e, strings of characters, on socket with port number 7890, and sending responses, being also strings, on the same socket.

Hello World Model Now it is time to construct a model in the TorXakis modelling language Txs. The description above says that, after an initial `Hello World!`, the system shall repeatedly receive names and then output `Hello` with that name. The behaviour is represented in the state automaton `STAUTDEF helloWorld` in Fig. 5; the complete model including all additional definitions is shown in Fig. 6.

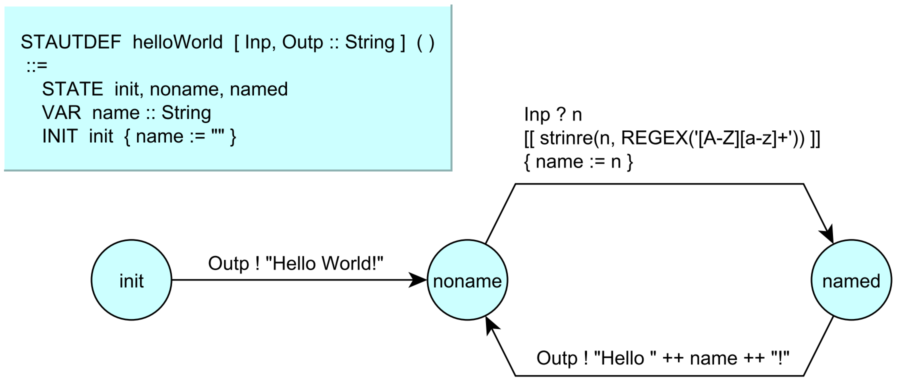
STAUTDEF helloWorld [Inp, Outp :: String] ( )
::=
STATE init, noname, named
VAR name :: String
INIT init { name := "" }
Inp ? n
[[strinre(n, REGEX('[A-Z][a-z]+'))]]
{ name := n }
init
Outp ! "Hello World!"
noname
named
Outp ! "Hello " ++ name ++ "!"

Figure 5: State automaton for *Hello World!*

In Fig. 5, there are three states: the initial state `init`, the state `noname` when no name has been entered yet, and state `named` after a name has been entered. The transition from *init* to *noname* specifies the welcoming message: on channel `Outp` the message `"Hello World!"` is produced. The exclamation mark `"!"` indicates that a fixed value, i.e., the string `"Hello World!"`, shall be ouput. The channel `Outp` is an ouput channel from the point of view of the *Hello World!* system. When testing, it will be an input for TorXakis, i.e., TorXakis will observe this message.

Once in state `noname`, the next possible action is an input message on channel `Inp` leading to state `named`. The part `? n :: String` indicates that the input message shall be of type `String` and that the actual input is bound to the local variable `n`. Moreover, the input message must satisfy the constraint given between `[[` and `]]`. This is a regular-expression constraint, it is expressed with the standard function `strinre` (*str*ing *in* *r*egular *e*xpression), and it requires that `n` starts with a capital letter followed by one or more small letters. The final part of this input transition, `name := n` , assigns the actual message that is communicated, i.e., the value of `n`, to the state variable `name`, so that it can be later used in other transitions. The variable `n` is local to the transiton, whereas the state variable `name` is global in the whole state automaton.

Once in state *named* the next possible transition is going back to *noname* while emitting on output channel `Outp` the message, that is, the string concatenation `"Hello " ++ name ++ "!"`, where `name` refers to the state variable containing the value obtained in the preceding transition.

An additional node in Fig. 5, which is not a state, gives the declaration of the state automaton as a `STAUTDEF`. It declares the name of the state automaton, its channels with message types, parameters, states, state variables, initial state, and the initial values of the state variables,

```
STAUTDEF  helloWorld [ Inp, Outp :: String ] ( )
 ::=
    STATE
        init, noname, named
    VAR
        name :: String
    INIT
        init { name := "" }
    TRANS
        init    ->  Outp ! "Hello World!"                                  ->  noname
        noname  ->  Inp ? n [[ strinre(n, REGEX('[A-Z][a-z]+')) ]] { name := n }  ->  named
        named   ->  Outp ! "Hello " ++ name ++ "!"                         ->  noname
ENDDEF

CHANDEF  Chans
 ::=
      Input  :: String ;
      Output :: String
ENDDEF

MODELDEF  Hello
 ::=
      CHAN IN   Input
      CHAN OUT  Output

      BEHAVIOUR
                helloWorld [Input, Output] ()
ENDDEF

CNECTDEF  Sut
 ::=
     CLIENTSOCK

     CHAN OUT  Input   HOST "localhost" PORT 7890
     ENCODE    Input   ? s  ->  ! s

     CHAN IN   Output  HOST "localhost" PORT 7890
     DECODE    Output  ! s  <-  ? s
ENDDEF
```

Figure 6: Txs model of *Hello World!*

In Txs, the input language for TorXakis, this model is given in Fig. 6. The model contains 4 definitions. The first one, `STAUTDEF`, defines a *state automaton*, and is directly generated from Fig. 5. The second definition `CHANDEF` defines two channels with messages of type `String`. Thirdly, the overall model is defined in `MODELDEF Hello`. It specifies which channels are inputs, which are outputs, and it specifies the behaviour of the model instantiating the state automaton `helloWorld` with appropriate channels and parameters.

Lastly, the `CNECTDEF` specifies that the tester connects as client to the SUT (the server) via sockets. It binds the channel `Input`, which is an input of the model and of the SUT, thus an *output* of TorXakis, to the socket $\langle \text{localhost}, 7890 \rangle$. Moreover, an encoding of actions to strings on the socket can be defined, but in this case, the encoding is trivial. Analogously, outputs from the SUT, i.e., inputs to TorXakis, are read from socket $\langle \text{localhost}, 7890 \rangle$ and decoded.

Hello World Testing Now we are ready to perform a test, by running the SUT with its adapter and TorXakis as two separate processes in two different windows. For the SUT we run:

```
$ socat TCP4-LISTEN:7890 EXEC:"./HelloWorld"
```

For TorXakis we have:

```
$ torxakis HelloWorld.txs

TXS >>  TorXakis :: Model-Based Testing

TXS >>  txsserver starting: "kubernetes.docker.internal" : 41873
TXS >>  Solver "z3" initialized : Z3 [4.8.5]
TXS >>  TxsCore initialized
TXS >>  LPEOps version 2019.07.05.02
TXS >>  input files parsed:
TXS >>  ["HelloWorld.txs"]
TXS >> tester Hello Sut
TXS >>  Tester started
TXS >> test 7
TXS >>  .....1: OUT: Act { { ( Output, [ "Hello World!" ] ) } }
TXS >>  .....2: IN: Act { { ( Input, [ "Pu" ] ) } }
TXS >>  .....3: OUT: Act { { ( Output, [ "Hello Pu!" ] ) } }
TXS >>  .....4: IN: Act { { ( Input, [ "Busvccc" ] ) } }
TXS >>  .....5: OUT: Act { { ( Output, [ "Hello Busvccc!" ] ) } }
TXS >>  .....6: IN: Act { { ( Input, [ "Pust" ] ) } }
TXS >>  .....7: OUT: Act { { ( Output, [ "Hello Pust!" ] ) } }
TXS >>  PASS
TXS >>
```

After having started TorXakis, we start the tester with `tester Hello Sut`, expressing that we wish to test with model `Hello` and SUT connection `Sut`, shown in the model file in Fig. 6. Then we can test 7 test steps with `test 7` and, indeed, after 7 test steps it stops with verdict `PASS`. A test run of 7 steps is rather small; we could have run for 100, 000 steps or more. TorXakis generates inputs to the SUT, such as `( Input, [ ""Busvccc" ] )`, with names satisfying the regular expression constraint. These input names are generated from the constraint by the SMT solver. Some extra functionality has been added in TorXakis in order to generate quasi-random inputs, which is not normally provided by an SMT solver. Moreover, it is checked that the outputs, such as `( Output, [ "Hello Busvccc!" ] )`, are correct.

6 A Job Dispatcher

The *Job-Dispatcher* is a system that distributes jobs over available processors. Jobs arrive in a dispatcher, are queued, and when a processor is available, the job is forwarded to that processor. When job processing is finished, the job is delivered. An overview of the system is given in Fig. 7; the TorXakis model is given in Figs. 8 and 9.

We explain some aspects of the model. The *Job-Dispatcher* has two typed channels for communication with the outside world. The types are user-defined algebraic data types. A value of type `JobOut` is either a `JobOut` with 3 fields, or an `Error` (which is not further used in this example). Type `JobList` defines a list of `JobData` in the standard recursive way. It is used in process `dispatcher` to queue the `Job` requests.

Functions can be defined in a standard, functional style, with recursion. Functions and expressions are strongly typed and overloading is allowed. The function `isValidJob` defines a constraint on `JobData` which is used when messages of type `JobData` are communicated to restrict the domain of valid messages.

Process `dispatcher` uses the choice process operator `##` to specify the choice between receiving a new job request, which is then added to the queue, or dispatching the first element of the queue to one of the processors if the queue is not empty. A `processor` processes a job by calculating the greatest common divisor using the function `gcd`. Process `processors` starts `pnum` `processor`-instances by ‘forking’ them. This is achieved by the parallel operator `|||`: a `processor` is started in parallel with more `processors` with decreased `pnum` until `pnum == 1`.

Finally, the behaviour of model `Disp` is defined as a `dispatcher` with empty initial queue in parallel with 4 `processors`. These communicate via channel `Job2Proc`: the `dispatcher` sends the jobs via this channel non-deterministically to one of the processors, but the `dispatcher` cannot influence which processor will take

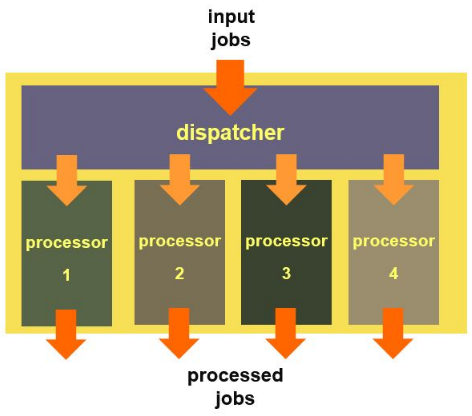

Figure 7: Overview Job-Dispatcher

the job. By using the `HIDE` constructor, the channel `Job2Proc` is hidden for the outside world, i.e., actions on this channel are abstracted away. From the outside, it can only be observed that after some time the job appears on channel `Finish`.

In the SUT connection the predefined functions `toString` and `fromString` are used to convert between the abstract data types on the model channels `Job` and `Finish` and a standard `String` representation on the socket. The user can also define her own converting functions or use the standard `to`/ `fromXML` functions.

Below the first 16 steps of a TorXakis test run are shown with the *Job-Dispatcher* on an SUT not shown here. The first parameter of both `JobData` and `JobOut` gives the `JobId`. It is clear that the jobs are not processed in order of arrival, and that sometimes there is no job being processed (e.g., after step 2 and after step 16), and sometimes there are 4 (e.g., after step 10).

```
$ torxakis JobDispatcher.txs

TXS >>  TorXakis :: Model-Based Testing
TXS >>  input files parsed: JobDispatcher.txs
TXS <<  tester Disp Sut
TXS >>  tester started
TXS >>  ....1: IN:  Act { { ( Job, [ JobData(56,"MX7",13,42) ] ) } }
TXS >>  ....2: OUT: Act { { ( Finish, [ JobOut(56,2,1) ] ) } }
TXS >>  ....3: IN:  Act { { ( Job, [ JobData(50,"KH",35,46) ] ) } }
TXS >>  ....4: IN:  Act { { ( Job, [ JobData(38,"KC00000",55,14) ] ) } }
TXS >>  ....5: OUT: Act { { ( Finish, [ JobOut(50,4,1) ] ) } }
TXS >>  ....6: IN:  Act { { ( Job, [ JobData(66,"KK0804",22,36) ] ) } }
TXS >>  ....7: IN:  Act { { ( Job, [ JobData(72,"PP839",57,41) ] ) } }
TXS >>  ....8: IN:  Act { { ( Job, [ JobData(36,"GC280",10,85) ] ) } }
TXS >>  ....9: OUT: Act { { ( Finish, [ JobOut(38,3,1) ] ) } }
TXS >>  ...10: IN:  Act { { ( Job, [ JobData(22,"AD36",87,2) ] ) } }
TXS >>  ...11: OUT: Act { { ( Finish, [ JobOut(22,4,1) ] ) } }
TXS >>  ...12: OUT: Act { { ( Finish, [ JobOut(36,3,5) ] ) } }
TXS >>  ...13: OUT: Act { { ( Finish, [ JobOut(72,1,1) ] ) } }
TXS >>  ...14: IN:  Act { { ( Job, [ JobData(90,"PK00",54,70) ] ) } }
TXS >>  ...15: OUT: Act { { ( Finish, [ JobOut(66,2,2) ] ) } }
TXS >>  ...16: OUT: Act { { ( Finish, [ JobOut(90,3,2) ] ) } }
```

```
CHANDEF  Channels
 ::=
      Job    :: JobData ;
      Finish :: JobOut
ENDDEF

TYPEDEF  JobData
 ::=
      JobData { jobId    :: Int
              ; jobDescr :: String
              ; x, y     :: Int
              }
ENDDEF

FUNCDEF  isValidJob ( jobdata :: JobData ) :: Bool
 ::=
          ( jobId(jobdata) > 0 )
      /\  ( (jobId(jobdata) % 2) == 0 )
      /\  strinre(jobDescr(jobdata), REGEX('[A-Z]{2}[0-9]*'))
      /\  ( x(jobdata) > 0 )
      /\  ( y(jobdata) > 0 )
ENDDEF

TYPEDEF  JobOut
 ::=
      JobOut  { jobId     :: Int
              ; processor :: Int
              ; gcd       :: Int
              }
    | Error   { reason    :: String
              }
ENDDEF

TYPEDEF  JobList
 ::=
      Nil
    | Cons    { hd    :: JobData
              ; tl    :: JobList
              }
ENDDEF

FUNCDEF  ++ (ll :: JobList; job :: JobData) :: JobList
 ::=
      IF   isNil(ll) THEN Cons(job, Nil)
                     ELSE Cons(hd(ll), tl(ll) ++ job)
      FI
ENDDEF

FUNCDEF  gcd (a, b :: Int) :: Int
 ::=
      IF a == b THEN a
                ELSE IF a > b THEN gcd (a - b, b)
                              ELSE gcd (a, b - a)
                     FI
      FI
ENDDEF
```

Figure 8: The *Job-Dispatcher* model: channels, types, and functions.

```
PROCDEF  dispatcher [Job :: JobData; Dispatch :: JobData] (queue :: JobList)
 ::=
               Job ? job :: JobData [[isValidJob(job)]]
          >->  dispatcher [Job, Dispatch] (queue ++ job)
      ##
          [[ not(isNil(queue)) ]]
          =>>  Dispatch ! hd(queue)
          >->  dispatcher [Job, Dispatch] (tl(queue))
ENDDEF

PROCDEF  processor [Start :: JobData; Finish :: JobOut] (pid :: Int)
 ::=
           Start ? job :: JobData
      >->  Finish ! JobOut(jobId(job), pid, gcd(x(job),y(job)))
      >->  processor [Start, Finish] (pid)
ENDDEF

PROCDEF  processors [Start :: JobData; Finish :: JobOut] (pnum :: Int)
 ::=
           processor [Start, Finish] (pnum)
      |||
           [[ pnum > 1 ]] =>>  processors [Start, Finish] (pnum - 1)
ENDDEF

MODELDEF Disp
 ::=
      CHAN IN    Job
      CHAN OUT   Finish

      BEHAVIOUR  HIDE [Job2Proc :: JobData]
                 IN
                          dispatcher [Job, Job2Proc]  (Nil)
                     |[ Job2Proc ]|
                          processors [Job2Proc, Finish] (4)
                 NI
ENDDEF

CNECTDEF  Sut
 ::=
      CLIENTSOCK

      CHAN OUT  Job                          HOST "localhost" PORT 7890
      ENCODE    Job    ? jd             ->  ! toString(jd)

      CHAN IN   Finish                       HOST "localhost" PORT 7890
      DECODE    Finish ! fromString(s)  <-  ? s
ENDDEF
```

Figure 9: The *Job-Dispatcher* model: processes, model, and SUT connection.

7 Testing Dropbox

We apply model-based testing with TorXakis to test Dropbox. Our work closely follows the work in [40], where Dropbox was tested with the model-based testing tool Quviq QuickCheck. We first briefly introduce Dropbox, we then discuss some aspects of the testing approach, we present a model in the TorXakis modelling language, we run some tests, and end with discussion.

7.1 Dropbox

Dropbox is a file-synchronization service [25], like Google-Drive and Microsoft-OneDrive. A file-synchronization service maintains consistency among multiple copies of files or a directory structure over different devices. A user can create, delete, read, or write a file on one device and Dropbox synchronizes this file with the other devices. One copy of the files on a device is called a *node*. A *conflict* arises when different nodes write to the same file: the content of the file cannot be uniquely determined anymore. Dropbox deals with conflicts by having the content that was written by one node in the original file, and adding an additional file, with a new name, with the conflicting content. Also this additional file will eventually be synchronized.

Synchronization is performed by uploading and downloading files to a Dropbox server, i.e., a Dropbox system with n nodes conceptually consists of $n + 1$ components. How synchronization is performed, i.e., when and which (partial) files are up- and downloaded by the Dropbox clients and how this is administered is part of the Dropbox implementation, i.e., the Dropbox protocol. Since we concentrate on testing the delivered synchronization service, we abstract from the precise protocol implementation.

A file synchronizer like Dropbox is a distributed, concurrent, and nondeterministic system. It has state (the synchronization status of files) and data (file contents), its modelling requires abstraction, leading to nondeterminism, because the precise protocol is not documented and the complete internal state is not observable, and partial modelling is needed because of its size. Altogether, file synchronizers are interesting and challenging systems to be tested, traditionally as well as model-based.

7.2 Testing Approach

We test the Dropbox synchronization service, that is, the SUT is the Dropbox behaviour as observed by users of the synchronization service, as a black-box. We closely follow [40], where a formal model for a synchronization service was developed and used for model-based testing of Dropbox with the tool Quviq QuickCheck [2], a descendant of Haskell QuickCheck [19]. This means that we use the same test setup, make the same assumptions, and transform their model for Quviq QuickCheck to the TorXakis modelling language. It also means that we will not repeat the detailed discussion of Dropbox intricacies and model refinements leading to their final model, despite that their model rules out implementations that calculate `clean` and handle a reverting write action without any communication with the server.

Like in [40], we restrict testing to one file and three nodes, and we use actions (SUT inputs) Read$_N$, Write$_N$, and Stabilize, which read the file at node N, (over-)write the file at node N, and read all files including conflict files when the system is stable, i.e., fully synchronized, respectively. Initially, and after deletion, the file is represented by the special content value "\$" ($\perp$ in [40]).

Our test setup consists of three Linux-virtual machines with Dropbox clients implementing the three nodes, numbered 0, 1, and 2. The file manipulation on the nodes is performed by plain Linux shell commands. These commands are sent by TorXakis, which runs on the host computer, via sockets; see Sect. 4. The adapters connecting TorXakis to the SUT consist of a one-line shell script connecting the sockets to the shell interpreter via the standard Linux utility `socat`.

For Stabilize we assume that the system has stabilized, i.e., all file synchronizations have taken place including distribution to all nodes of all conflict files. Like in [40], we implement this by simply waiting for at least 30 seconds. Since the TorXakis modelling language itself does not support realtime, TorXakis sends a command to the adapter to wait for 30 seconds.

7.3 Modelling

Our Dropbox model is a straightforward translation of [40, Section IV: Formalizing the specification] into the modelling language of TorXakis. Parts of the model are shown in Figs. 10, 12, 13, 14, 15, 16, and 17.

```
CHANDEF MyChans ::=
    In0,  In1,  In2   :: Cmd ;
    Out0, Out1, Out2  :: Rsp
ENDDEF

TYPEDEF Cmd ::=
      Read
    | Write      { value :: Value  }
    | Stabilize
ENDDEF

TYPEDEF Rsp ::=
      Ack
    | NAck       { error :: String }
    | File       { value :: Value  }
ENDDEF

TYPEDEF Value ::=
    Value { value :: String }
ENDDEF

FUNCDEF isValidValue ( val :: Value ) :: Bool ::=
    strinre( value(val), REGEX('[A-Z]{1,3}') )
ENDDEF
```

Figure 10: Dropbox model - channels and their types.

A TorXakis model is a collection of different kinds of definitions; see Sect. 4. The first one, `CHANDEF`, defines the channels with their typed messages; see Fig. 10. TorXakis assumes that an SUT communicates by receiving and sending typed messages. A message received by the SUT is an input, and thus an action initiated by the tester. A message sent by the SUT is an SUT output, and is observed and checked by the tester. For Dropbox there are three input channels: `In0`, `In1`, and `In2`, where commands of type `Cmd` are sent to the SUT, for each node, respectively. There are also three output channels `Out0`, `Out1`, and `Out2`, where responses of type `Rsp` are received from the SUT. The commands (SUT inputs) with their corresponding responses (SUT outputs) are:

`Read` reads the file on the local node, which leads to a response consisting of the current file content `value`;

`Write(value)` writes the new value `value` to the file while the response gives the old value;

`Stabilize` reads all file values, i.e., the original file and all conflict files, after stabilization, i.e., after all file synchronizations have taken place.

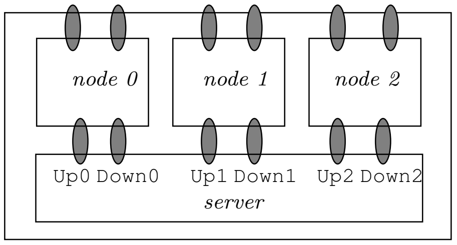

Figure 11: Dropbox structure.

In addition to these visible actions, there are hidden actions. If a user modifies a file, Dropbox will upload it to the Dropbox server, and then later download this file to the other nodes. But a Dropbox user, and thus also the (black-box) tester cannot observe these actions, and consequently, they do not occur in the CHANDEF definition. Yet, these actions do occur and they do change the state of the Dropbox system. We use six channels Down0, Down1, Down2, Up0, Up1, and Up2 to model these actions, and later it will be shown how we can explicitly *hide* these channels. The conceptual structure of Dropbox with nodes, server, and channels is given in Fig. 11. The outer box is our SUT.

```
PROCDEF dropBox [ In0,  In1,  In2     :: Cmd
                ; Out0, Out1, Out2    :: Rsp
                ; Down0, Down1, Down2
                ; Up0, Up1, Up2
                ]
                ( serverVal :: Value
                ; conflicts :: ValueList
                ; localVal  :: ValueList
                ; fresh     :: BoolList
                ; clean     :: BoolList
                ) ::=
       In0     !Read
   >-> Out0    !File(lookup(localVal,Node(0)))
   >-> dropBox [ In0,In1,In2,Out0,Out1,Out2
               , Down0,Down1,Down2,Up0,Up1,Up2
               ]
               ( serverVal
               , conflicts
               , localVal
               , fresh
               , clean
               )
 ##
       In0     ?cmd [[ IF   isWrite(cmd)
                       THEN isValidValue(
                                  value(cmd))
                       ELSE False
                       FI ]]
   >-> Out0    !File(lookup(localVal,Node(0)))
   >-> dropBox [ In0,In1,In2,Out0,Out1,Out2
               , Down0,Down1,Down2,Up0,Up1,Up2
               ]
               ( serverVal
               , conflicts
               , update(localVal
                       ,Node(0)
                       ,value(cmd))
               , fresh
               , update(clean,Node(0),False)
               )
 ##
   .....
```

Figure 12: Dropbox model - main process dropbox with transitions Read and Write.

The next step is to define the processes that model state behaviour. The main process is PROCDEF dropbox which models the behaviour of Dropbox, combining the commands (SUT inputs), responses (SUT outputs), and the checks on them in one state machine; see Figs. 12, 13, and 14. The state machine is defined as a recursive process dropbox with channel parameters In0, ..., Up2, and with state variables exactly as in [40]:

- a global stable value serverVal represents the file value currently held on the server;
- a global set conflicts holds the conflicting file values, represented as a ValueList;
- for each node N, there is a local file value $\texttt{localVal}_N$, where all local file values together are represented as a list of values with three elements, the first element representing $\texttt{localVal}_0$, etc.;
- for each node N, there is a freshness value $\texttt{fresh}_N$, indicating whether node N has downloaded the latest value of serverVal; all freshness values together are represented as a list of Booleans with three elements,

the second element representing $\texttt{fresh}_1$, etc.;

- for each node N, there is a cleanliness value $\texttt{clean}_N$, indicating whether the latest local modification has been uploaded; together they are represented as a list of Booleans with three elements, the third element representing $\texttt{clean}_2$, etc.

The recursive process `dropbox` defines for each node transitions for reading, writing, uploading, and downloading the file, and one transition for `Stabilize`. The different transitions are separated by `'##'`, the TorXakis *choice* operator. The transitions for reading and writing consist of two steps: first a command (SUT input) followed by an SUT output. "Followed by" is expressed by the TorXakis *action-prefix* operator `'>->'`. After the response, `dropbox` is recursively called with updated state variables.

```
  .....
##
  [[ not(lookup(fresh,Node(0)))
     /\  lookup(clean,Node(0)) ]]
  =>> Down0
  >-> dropBox [ In0,In1,In2,Out0,Out1,Out2
              , Down0,Down1,Down2,Up0,Up1,Up2
              ]
              ( serverVal
              , conflicts
              , update(localVal
                      ,Node(0)
                      ,serverVal)
              , update(fresh,Node(0),True)
              , clean
              )
##
  [[ not(lookup(clean,Node(0))) ]]
  =>> Up0
  >-> dropBox [ In0,In1,In2,Out0,Out1,Out2
              , Down0,Down1,Down2,Up0,Up1,Up2
              ]
              ( IF   lookup(fresh,Node(0))
                   /\ (lookup(localVal,Node(0))
                        <> serverVal)
                THEN lookup(localVal,Node(0))
                ELSE serverVal
                FI
              , IF   not(lookup(fresh,Node(0)))
                   /\ (lookup(localVal,Node(0))
                        <> serverVal)
                   /\ (lookup(localVal,Node(0))
                        <> Value("$"))
                THEN Values(lookup(localVal
                                  ,Node(0))
                                  ,conflicts)
                ELSE conflicts
                FI
              , localVal
              , IF   lookup(fresh,Node(0))
                   /\ (lookup(localVal,Node(0))
                        <> serverVal)
                THEN othersUpdate(fresh
                                 ,Node(0)
                                 ,False)
                ELSE fresh
                FI
              , update(clean,Node(0),True)
              )
##
    .....
```

Figure 13: Dropbox model - transitions `Down` and `Up` in the main process `dropbox`.

Consider file-reading for node 0 (Fig. 12). The first action is input `Read` on channel `In0`. Then the SUT will produce output `File(lookup(localVal,Node(0)))`, i.e., the `File` made by looking up the `localVal` value of `Node(0)`. This is an expression in the data specification language of TorXakis, which is based on algebraic data types (ADT) and functions like in functional languages. This data language is very powerful, but also very rudimentary. Data types such as `ValueList` have to be defined explicitly as recursive types (Fig. 15), consisting of either an empty list `NoValues`, or a non-empty list `Values` with as fields a head value `hd` and a tail `tl`, which is again a `ValueList`. Functions like `lookup` have to be defined explicitly, too, in a functional (recursive) style. Fig. 15 gives as examples the functions `lookup` and `update`; other functions are included in the full model [58]. After the output there is the recursive call of process `dropbox`, where state parameters are not modified in case of file-reading.

```
   .....
 ##
   [[ allTrue(fresh) /\ allTrue(clean) ]]
   =>>
     (      In0  !Stabilize
        >-> fileAndConflicts [Out0]
                   (Values(serverVal,conflicts))
     >>> dropBox [ In0,In1,In2,Out0,Out1,Out2
                 , Down0,Down1,Down2,Up0,Up1,Up2
                 ]
                 ( serverVal
                 , conflicts
                 , localVal
                 , fresh
                 , clean
                 )
     )
ENDDEF -- dropbox

PROCDEF fileAndConflicts [ Out :: Rsp ]
                         ( values :: ValueList )
                         EXIT ::=
       Out ?rsp [[ IF   isFile(rsp)
                   THEN isValueInList(
                            values,value(rsp))
                   ELSE False
                   FI ]]
   >-> fileAndConflicts [Out]
           (removeListValue(values,value(rsp)))
 ##
   [[ isNoValues(values) ]]
   =>> Out !Ack
   >-> EXIT
ENDDEF -- fileAndConflicts
```

Figure 14: Dropbox model - transition `Stabilize` and and process `fileAndConflicts`.

Writing a file for node 0 is analogous, but with two differences (Fig. 12). First, the action of writing is not a unique action, but it is parameterized with the new file value. This is expressed by `?cmd`, stating that on channel `In0` any value, represented by variable `cmd`, can be communicated, which satisfies the constraint between `'[['` and `']]'`. This constraint expresses that `cmd` must be a `Write` command, referring to the constructor `Write` in type `Cmd`. Moreover, the `value` of the `write`-command must be a valid value, which means (see Fig. 10) that it shall be a string contained in the regular expression `REGEX('[A-Z]{1,3}')`, i.e., a string of one to three capital letters. Using this constraint, TorXakis will automatically generate valid input values, using an SMT solver.

The second difference concerns the updates to the state parameters in the recursive call of `dropbox`. We see that `localVal` for `node(0)` is updated with the new file value that was used as input in the communication on channel `In0`. Moreover, `node(0)` is not `clean` anymore.

The transitions for uploading and downloading will be hidden, so they do not have communication with the SUT. They just deal with constraints and updates on the state. Downloading to node 0 (Fig. 13) can occur if node 0 is not `fresh` yet `clean`, as is modelled in the guard (precondition) between `'[['` and `']] =>>'`, before execution of action `Down0`. The effect of the action is an update of the `localVal` of `Node(0)` with `serverVal`, and re-established freshness.

Uploading can occur if a node is not `clean`. The state update is rather intricate, which has to do with conflicts that can occur when uploading, and with special cases if the upload is actually a delete (represented by file value "$") and if the upload is equal to serverVal. The state update has been directly copied from [40] where it is very well explained, so for more details we refer there.

```
TYPEDEF Node ::=
      Node { node :: Int }
ENDDEF

TYPEDEF ValueList ::=
      NoValues
    | Values   { hd :: Value; tl :: ValueList }
ENDDEF

FUNCDEF lookup ( vals :: ValueList
               ; n :: Node ) :: Value ::=
    IF   isNoValues(vals)
    THEN Value("$")
    ELSE IF   node(n) == 0
         THEN hd(vals)
         ELSE lookup(tl(vals),Node(node(n)-1))
         FI
    FI
ENDDEF

FUNCDEF update ( vals :: ValueList
               ; n :: Node
               ; v :: Value ) :: ValueList ::=
    IF   isNoValues(vals)
    THEN NoValues
    ELSE IF   node(n) == 0
         THEN Values(v,tl(vals))
         ELSE Values(hd(vals)
                   ,update(tl(vals),
                       Node(node(n)-1),v))
         FI
    FI
ENDDEF
```

Figure 15: Dropbox model - data types and functions.

We have discussed reading, writing, uploading, and downloading for node 0. Similar transitions are defined for nodes 1 and 2. Of course, in the final model, parameterized transitions are defined for node N, which can then be instantiated. Since this parameterization is not completely trivial because of passing of state variable values, we do not discuss it here.

The last action is Stabilize, which can occur if all nodes are fresh and clean; see Fig. 14. Since all nodes are assumed to have synchronized it does not matter which node we use; we choose node 0. Stabilize produces all file content values that are currently available including the conflict files. These content values are produced one by one, in arbitrary order, as responses on channel Out0 with an acknowledge Ack after the last one. Process fileAndConflicts models that all these content values indeed occur once in the list of serverVal and conflicts. It removes from Values (which is of type ValueList) each content value value(rsp) that has been observed on Out0, until the list is empty, i.e., isNoValues(values) holds. Then the acknowledge Ack is sent, and the process EXITs, which is the trigger for the recursive call of dropbox after fileAndConflicts.

The next step is to define the complete model in the MODELDEF; see Fig. 16. The MODELDEF specifies which channels are inputs, which are outputs, and what the BEHAVIOUR of the model is using the previously defined processes. In our case it is a call of the dropbox process with appropriate instantiation of the state variables serverVal, conflicts, localVal, fresh, and clean. Moreover, this is the place where the channels Down0, . . ., Up2 are hidden with the construct HIDE [*channels*] IN . . . NI. Actions that occur on hidden channels are *internal actions* (in process-algebra usually denoted by τ). They are not visible to the system environment, but they do lead to state changes of which the consequences can be visible, e.g., when a transition that is enabled before the occurrence of τ is no longer enabled in the state after the τ-occurrence. Visible actions, that is inputs and outputs, are visible to the system environment. They lead to state changes both in the system and in its

```
MODELDEF DropboxModel ::=
   CHAN IN   In0,  In1,  In2
   CHAN OUT  Out0, Out1, Out2
   BEHAVIOUR
     HIDE [Down0,Down1,Down2,Up0,Up1,Up2] IN
       dropBox [ In0,In1,In2,Out0,Out1,Out2
               , Down0,Down1,Down2,Up0,Up1,Up2
               ]
               ( Value("$")
               , NoValues
               , Values(Value("$"),
                  Values(Value("$"),
                   Values(Value("$"),NoValues)))
               , Bools(True,Bools(True,
                   Bools(True,NoBools)))
               , Bools(True,Bools(True,
                   Bools(True,NoBools)))
               )
     NI
ENDDEF
```

Figure 16: Dropbox model - definition of the model that specifies the behaviour over the observable channels.

environment.

The last definition CNECTDEF specifies how the tester connects to the external world via sockets; see Fig. 17. In the Dropbox case, TorXakis connects as socket client, CLIENTSOCK, to the SUT, that shall act as the socket server. The CNECTDEF binds the abstract model channel In0, which is an input of the model and of the SUT, thus an *output* of TorXakis, to the socket on host txs0-pc, one of the virtual machines running Dropbox, and port number 7890. Moreover, the encoding of abstract messages of type Cmd on channel In0 to strings on the socket is elaborated with function encodeCmd: a command is encoded as a string of one or more Linux commands, which can then be sent to and executed by the appropriate virtual machine. Analogously, outputs from the SUT, i.e., inputs to TorXakis, are read from socket ⟨txs0-pc, 7890⟩ and decoded to responses of type Rsp on channel Out0 using function decodeRsp. Analogous bindings of abstract channels to real-world socksets are specified for In1, Out1, In2, and Out2.

7.4 Model-Based Testing

Now that we have an SUT and a model, we can start generating tests and executing them. First, we start the SUT, that is, the virtual machines, start the Dropbox client on these machines, and start the adapter scripts. Then we can start TorXakis and run a test; see Fig. 18. User inputs to TorXakis are marked TXS << ; responses from TorXakis are marked TXS >> .
We start the tester with tester DropboxModel DropboxSut, expressing that we wish to test with MODELDEF DropboxModel and CNECTDEF DropboxSut. Then we test for 100 test steps with test 100, and indeed, after 100 test steps it stops with verdict PASS.

TorXakis generates inputs to the SUT, such as on line 7: In0, [Write(Value("SHK"))]) , indicating that on channel In0 an input action Write with file value "SHK" has occurred. The input file value is generated by TorXakis from the isValidValue constraint, using the SMT solver. This action is followed, on line 8, by an output from the SUT on channel Out0, which is the old file value of Node 0, which is "$", representing the empty file. TorXakis checks that this is indeed the correct response.

Only visible input and output actions are shown in this trace. Hidden actions are not shown, but they do occur internally, as can be seen, for example, from line 24: the old file value on Node 2 was "X", but this value was only written to node 0 (line 11), so node 0 and node 2 must have synchronized the value "X" via internal Up and Down actions. Also just before Stabilize, lines 67–74, synchronization has obviously taken place, which can only happen using hidden Up and Down actions. Due to the distributed nature of Dropbox and its nondeterminism it

```
CNECTDEF DropboxSut ::=
     CLIENTSOCK
     CHAN OUT  In0   HOST "txs0-pc" PORT 7890
     ENCODE    In0   ?cmd  ->  !encodeCmd(cmd)
     CHAN IN   Out0  HOST "txs0-pc" PORT 7890
     DECODE    Out0  !decodeRsp(s)  <-  ?s
     CHAN OUT  In1   HOST "txs1-pc" PORT 7891
     ENCODE    In1   ?cmd  ->  !encodeCmd(cmd)
     CHAN IN   Out1  HOST "txs1-pc" PORT 7891
     DECODE    Out1  !decodeRsp(s)  <-  ?s
     CHAN OUT  In2   HOST "txs2-pc" PORT 7892
     ENCODE    In2   ?cmd  ->  !encodeCmd(cmd)
     CHAN IN   Out2  HOST "txs2-pc" PORT 7892
     DECODE    Out2  !decodeRsp(s)  <-  ?s
ENDDEF

FUNCDEF encodeCmd ( cmd :: Cmd ) :: String ::=
    IF   isRead(cmd)
    THEN "cat testfile"
    ELSE IF isWrite(cmd)
    THEN "cat testfile ; " ++ "echo \"" ++
         value(value(cmd)) ++ "\" > testfile"
    ELSE IF isStabilize(cmd)
    THEN "sleep 30 ; cat * ; echo "
    ELSE "" FI FI FI
ENDDEF

FUNCDEF decodeRsp ( s :: String ) :: Rsp ::=
    IF   s == ""
    THEN Ack
    ELSE IF s == "$"
    THEN File(Value("$"))
    ELSE IF strinre(s,REGEX('[A-Z]{1,3}'))
    THEN File(Value(s))
    ELSE NAck(s) FI FI FI
ENDDEF
```

Figure 17: Dropbox model - connection to the external world.

```
$ torxakis Dropbox.txs
TXS >>  TorXakis :: Model-Based Testing
TXS >>  txsserver starting: "PC-31093.tsn.tno.nl" : 60275
TXS >>  Solver "z3" initialized : Z3 [4.6.0]
TXS >>  TxsCore initialized
TXS >>  input files parsed:
TXS >>  ["Dropbox.txs"]
TXS <<  tester DropboxModel DropboxSut
TXS >>  tester started
TXS <<  test 100
TXS >>  .....1: IN:  Act { { ( In1, [ Read ] ) } }
TXS >>  .....2: OUT: Act { { ( Out1, [ File(Value("$")) ] ) } }
TXS >>  .....3: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  .....4: OUT: Act { { ( Out2, [ File(Value("$")) ] ) } }
TXS >>  .....5: IN:  Act { { ( In1, [ Write(Value("P")) ] ) } }
TXS >>  .....6: OUT: Act { { ( Out1, [ File(Value("$")) ] ) } }
TXS >>  .....7: IN:  Act { { ( In0, [ Write(Value("SHK")) ] ) } }
TXS >>  .....8: OUT: Act { { ( Out0, [ File(Value("$")) ] ) } }
TXS >>  .....9: IN:  Act { { ( In1, [ Read ] ) } }
TXS >>  ....10: OUT: Act { { ( Out1, [ File(Value("P")) ] ) } }
TXS >>  ....11: IN:  Act { { ( In0, [ Write(Value("X")) ] ) } }
TXS >>  ....12: OUT: Act { { ( Out0, [ File(Value("SHK")) ] ) } }
TXS >>  ....13: IN:  Act { { ( In2, [ Write(Value("A")) ] ) } }
TXS >>  ....14: OUT: Act { { ( Out2, [ File(Value("$")) ] ) } }
TXS >>  ....15: IN:  Act { { ( In2, [ Write(Value("SP")) ] ) } }
TXS >>  ....16: OUT: Act { { ( Out2, [ File(Value("A")) ] ) } }
TXS >>  ....17: IN:  Act { { ( In1, [ Write(Value("BH")) ] ) } }
TXS >>  ....18: OUT: Act { { ( Out1, [ File(Value("P")) ] ) } }
TXS >>  ....19: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  ....20: OUT: Act { { ( Out2, [ File(Value("SP")) ] ) } }
TXS >>  ....21: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ....22: OUT: Act { { ( Out0, [ File(Value("X")) ] ) } }
TXS >>  ....23: IN:  Act { { ( In2, [ Write(Value("PXH")) ] ) } }
TXS >>  ....24: OUT: Act { { ( Out2, [ File(Value("X")) ] ) } }
TXS >>  ....25: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  ....26: OUT: Act { { ( Out2, [ File(Value("PXH")) ] ) } }
TXS >>  ....27: IN:  Act { { ( In0, [ Write(Value("AX")) ] ) } }
TXS >>  ....28: OUT: Act { { ( Out0, [ File(Value("PXH")) ] ) } }
TXS >>  ....29: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  ....30: OUT: Act { { ( Out2, [ File(Value("AX")) ] ) } }
TXS >>  ....31: IN:  Act { { ( In1, [ Read ] ) } }
TXS >>  ....32: OUT: Act { { ( Out1, [ File(Value("AX")) ] ) } }
TXS >>  ....33: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ....34: OUT: Act { { ( Out0, [ File(Value("AX")) ] ) } }
TXS >>  ....35: IN:  Act { { ( In2, [ Write(Value("TPH")) ] ) } }
TXS >>  ....36: OUT: Act { { ( Out2, [ File(Value("AX")) ] ) } }
TXS >>  ....37: IN:  Act { { ( In0, [ Write(Value("X")) ] ) } }
TXS >>  ....38: OUT: Act { { ( Out0, [ File(Value("AX")) ] ) } }
TXS >>  ....39: IN:  Act { { ( In2, [ Write(Value("CPH")) ] ) } }
TXS >>  ....40: OUT: Act { { ( Out2, [ File(Value("TPH")) ] ) } }
TXS >>  ....41: IN:  Act { { ( In1, [ Write(Value("HX")) ] ) } }
TXS >>  ....42: OUT: Act { { ( Out1, [ File(Value("CPH")) ] ) } }
TXS >>  ....43: IN:  Act { { ( In1, [ Read ] ) } }
TXS >>  ....44: OUT: Act { { ( Out1, [ File(Value("HX")) ] ) } }
TXS >>  ....45: IN:  Act { { ( In1, [ Read ] ) } }
TXS >>  ....46: OUT: Act { { ( Out1, [ File(Value("HX")) ] ) } }
TXS >>  ....47: IN:  Act { { ( In2, [ Write(Value("Q")) ] ) } }
TXS >>  ....48: OUT: Act { { ( Out2, [ File(Value("HX")) ] ) } }
TXS >>  ....49: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ....50: OUT: Act { { ( Out0, [ File(Value("Q")) ] ) } }
TXS >>  ....51: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ....52: OUT: Act { { ( Out0, [ File(Value("Q")) ] ) } }
TXS >>  ....53: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  ....54: OUT: Act { { ( Out2, [ File(Value("Q")) ] ) } }
TXS >>  ....55: IN:  Act { { ( In0, [ Write(Value("K")) ] ) } }
TXS >>  ....56: OUT: Act { { ( Out0, [ File(Value("Q")) ] ) } }
TXS >>  ....57: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  ....58: OUT: Act { { ( Out2, [ File(Value("K")) ] ) } }
TXS >>  ....59: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ....60: OUT: Act { { ( Out0, [ File(Value("K")) ] ) } }
TXS >>  ....61: IN:  Act { { ( In2, [ Write(Value("ABL")) ] ) } }
TXS >>  ....62: OUT: Act { { ( Out2, [ File(Value("K")) ] ) } }
TXS >>  ....63: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  ....64: OUT: Act { { ( Out2, [ File(Value("ABL")) ] ) } }
TXS >>  ....65: IN:  Act { { ( In2, [ Write(Value("P")) ] ) } }
TXS >>  ....66: OUT: Act { { ( Out2, [ File(Value("ABL")) ] ) } }
TXS >>  ....67: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ....68: OUT: Act { { ( Out0, [ File(Value("P")) ] ) } }
TXS >>  ....69: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  ....70: OUT: Act { { ( Out2, [ File(Value("P")) ] ) } }
TXS >>  ....71: IN:  Act { { ( In1, [ Read ] ) } }
TXS >>  ....72: OUT: Act { { ( Out1, [ File(Value("P")) ] ) } }
TXS >>  ....73: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ....74: OUT: Act { { ( Out0, [ File(Value("P")) ] ) } }
TXS >>  ....75: IN:  Act { { ( In0, [ Stabilize ] ) } }
TXS >>  ....76: OUT: Act { { ( Out0, [ File(Value("P")) ] ) } }
TXS >>  ....77: OUT: Act { { ( Out0, [ File(Value("X")) ] ) } }
TXS >>  ....78: OUT: Act { { ( Out0, [ File(Value("BH")) ] ) } }
TXS >>  ....79: OUT: Act { { ( Out0, [ File(Value("SP")) ] ) } }
TXS >>  ....80: OUT: Act { { ( Out0, [ Ack ] ) } }
TXS >>  ....81: IN:  Act { { ( In1, [ Write(Value("AB")) ] ) } }
TXS >>  ....82: OUT: Act { { ( Out1, [ File(Value("P")) ] ) } }
TXS >>  ....83: IN:  Act { { ( In1, [ Write(Value("X")) ] ) } }
TXS >>  ....84: OUT: Act { { ( Out1, [ File(Value("AB")) ] ) } }
TXS >>  ....85: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ....86: OUT: Act { { ( Out0, [ File(Value("P")) ] ) } }
TXS >>  ....87: IN:  Act { { ( In2, [ Write(Value("PNB")) ] ) } }
TXS >>  ....88: OUT: Act { { ( Out2, [ File(Value("P")) ] ) } }
TXS >>  ....89: IN:  Act { { ( In1, [ Write(Value("D")) ] ) } }
TXS >>  ....90: OUT: Act { { ( Out1, [ File(Value("X")) ] ) } }
TXS >>  ....91: IN:  Act { { ( In1, [ Write(Value("L")) ] ) } }
TXS >>  ....92: OUT: Act { { ( Out1, [ File(Value("D")) ] ) } }
TXS >>  ....93: IN:  Act { { ( In2, [ Read ] ) } }
TXS >>  ....94: OUT: Act { { ( Out2, [ File(Value("PNB")) ] ) } }
TXS >>  ....95: IN:  Act { { ( In1, [ Write(Value("KK")) ] ) } }
TXS >>  ....96: OUT: Act { { ( Out1, [ File(Value("PNB")) ] ) } }
TXS >>  ....97: IN:  Act { { ( In2, [ Write(Value("P")) ] ) } }
TXS >>  ....98: OUT: Act { { ( Out2, [ File(Value("PNB")) ] ) } }
TXS >>  ....99: IN:  Act { { ( In0, [ Read ] ) } }
TXS >>  ...100: OUT: Act { { ( Out0, [ File(Value("KK")) ] ) } }
TXS >>  PASS
TXS <<
```

is not so easy to check the response of the `Stabilize` command on line 75. It is left to the reader to check that the outputs on lines 76–80 are indeed all conflict-file contents together with the server file, and that TorXakis correctly assigned the verdict `PASS`.

Many more test cases can be generated and executed on-the-fly. TorXakis generates random test cases, so each time another test case is generated, and appropriate responses are checked on-the-fly. It should be noted that TorXakis is not very fast. Constraint solving, nondeterminism, and dealing with internal (hidden) actions (exploring all possible 'explanations' in terms of [40]) can make that computation of the next action takes a minute.

7.5 Discussion and Comparison

We showed that model-based testing of a file synchronizer that is distributed, concurrent, and nondeterministic, that combines state and data, and that has internal state transitions that cannot be observed by the tester, is possible with TorXakis, just as with Quviq QuickCheck. The model used for TorXakis is a direct translation of the QuickCheck model.

As opposed to the work with Quviq QuickCheck, we did not yet try to reproduce the detected 'surprises', i.e., probably erroneous behaviours of Dropbox. More testing and analysis is needed, probably with steering the test generation into specific corners of behaviour. Moreover, some of these 'surprises' require explicit deletion of files, which we currently do not do. For steering, TorXakis has a feature called *test purposes*, and future work will include using test purposes to reproduce particular behaviours. But it might be that these Dropbox 'surprises' have been repaired in the mean time, as was announced in [40].

A difference between the Quviq QuickCheck approach and TorXakis is the treatment of hidden actions. Whereas Quviq QuickCheck needs explicit reasoning about possible 'explanations' on top of the state machine model using a specifically developed technique, the process-algebraic language of TorXakis has `HIDE` as an abstraction operator built into the language, which allows to turn any action into an internal action. Together with the **ioco**-conformance relation, which takes such internal actions into consideration, it makes the construction of 'explanations' completely automatic and an integral part of test generation and observation analysis. Although no real speed comparisons were made, it looks like the general solution of dealing with hidden actions and nondeterminism in TorXakis has a price to be paid in terms of computation speed.

TorXakis has its own modelling language based on process algebra and algebraic data types and with symbolic transition system semantics. This allows to precisely define what a conforming SUT is using the **ioco**-conformance relation, in a formal testing framework which enables to define soundness and exhaustiveness of generated test cases. Quviq QuickCheck is embedded in the Erlang programming language, that is, specifications are just Erlang programs that call libraries supplied by QuickCheck and the generated tests invoke the SUT directly via Erlang function calls. A formal notion of 'conformance' of a SUT is missing.

A powerful feature of Quviq QuickCheck for analysis and diagnosis is shrinking. It automatically reduces the length of a test after failure detection, which eases analysis. Currently, TorXakis has no such feature.

Several extensions of the presented work are possible. One of them is applying the same model to test other file synchronizers. Another is adding additional Dropbox behaviour to the model, such as working with multiple, named files and folders. This would complicate the model, but not necessarily fundamentally change the model: instead of keeping a single file value we would have to keep a (nested) map of file names to file values, and read and write would be parameterized with file or folder names.

Another, more fundamental question concerns the quality of the generated test cases. How good are the test suites in detecting bugs, what is their coverage, and to what extent can we be confident that an SUT that passes the tests is indeed correct? Can we compare different (model-based) test generation strategies, e.g., the one of Quviq QuickCheck with the one of TorXakis, and assign a measure of quality or coverage to the generated test suites, and thus, indirectly, a measure to the quality of the tested SUT?

References

[1] S. Abramsky. Observational Equivalence as a Testing Equivalence. *Theoretical Computer Science*, 53(3):225–241, 1987.

[2] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq Quickcheck. In *ACM SIGPLAN Workshop on Erlang*, ERLANG'06, pages 2–10. ACM, NY, USA, 2006.

[3] Axini. Testautomatisering. `http://www.axini.com`.

[4] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, and C. Reynolds, A.and Tinelli. CVC4. In *Computer Aided Verification – CAV 2011*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer-Verlag, 2011.

[5] A. Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution . In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[6] A. Belinfante, J. Feenstra, R.G. 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, *Int. Workshop on Testing of Communicating Systems 12*, pages 179–196. Kluwer Academic Publishers, 1999.

[7] G. Bernot, M. G. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 1991(November):387–405, 1991.

[8] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional Testing with IOCO. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing – FATES 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 2004.

[9] Bijl, M. van der and Beek, H. van. Model-Based Testing in Safety-Critical Scaled Agile. *Bits & Chips*, August 2021.

[10] H.C. Bohnenkamp and M.I.A. Stoelinga. Quantitative Testing. In *ACM & IEEE Int. Conf. on Embedded Software – EMSOFT'08*, pages 227–236. ACM, 2008.

[11] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[12] P. van den Bos, R. Janssen, and J. Moerman. n-Complete Test Suites for IOCO. *Software Quality Journal*, 27(2):563–588, 2019.

[13] P. van den Bos and J. Tretmans. Coverage-Based Testing with Symbolic Transition Systems. In D. Beyer and C. Keller, editors, *Tests and Proofs – TAP 2019*, volume 11823 of *Lecture Notes in Computer Science*, pages 64–82. Springer Int. Publishing, 2019.

[14] Bos, P. van den. *Coverage and Games in Model-Based Testing.* PhD thesis, Radboud University, Nijmegen, The Netherlands, 2020.

[15] L. Brandán Briones and E. Brinksma. A Test Generation Framework for *quiescent* Real-Time Systems. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, volume 3395 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, 2005.

[16] E. Brinksma. A Theory for the Derivation of Tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 63–74. North-Holland, 1988.

[17] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A Formal Approach to Conformance Testing. In J. de Meer, L. Mackert, and W. Effelsberg, editors, *Second Int. Workshop on Protocol Test Systems*, pages 349–363. North-Holland, 1990.

[18] T.S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[19] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ACM SIGPLAN Int. Conf. on Functional Programming 2000*, ICFP'00, pages 268–279. ACM, NY, USA, 2000.

[20] D.R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial.* GrammaTech, Inc., 2011.

[21] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C.R. Ramakrishnan and J. Rehof, editors, *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[22] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, September 2011.

[23] R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[24] E.W. Dijkstra. Notes On Structured Programming – EWD249. T.H. Report 70-WSK-03, Technische Hogeschool Eindhoven, Eindhoven, The Netherlands, 1969.

[25] Dropbox. `https://www.dropbox.com`.

[26] H. Eertink. Executing LOTOS specifications: The SMILE tool. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 221–234. Kluwer Academic Publishers, 1995.

[27] P.H.J. van Eijk. *Software Tools for the Specification Language LOTOS*. PhD thesis, University of Twente, Enschede, The Netherlands, 1988.

[28] J. Engelfriet. Determinacy $\rightarrow$ (Observation Equivalence = Trace Equivalence). *Theoretical Computer Science*, 36(1):21–25, 1985.

[29] L. Frantzen, J. Tretmans, and T. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2005.

[30] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Roşu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV'06*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer-Verlag, 2006.

[31] M.-C. Gaudel. Testing can be Formal, too. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1995.

[32] R.J. van Glabbeek. The Linear Time – Branching Time Spectrum. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90*, number 458 in Lecture Notes in Computer Science, pages 278–297. Springer-Verlag, 1990.

[33] R.J. van Glabbeek. The Linear Time – Branching Time Spectrum II (The Semantics of Sequential Systems with Silent Moves). In E. Best, editor, *CONCUR'93*, number 715 in Lecture Notes in Computer Science, pages 66–81. Springer-Verlag, 1993.

[34] J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.

[35] A. Hartman and K. Nagin. The AGEDIS Tools for Model Based Testing. In *Int. Symposium on Software Testing and Analysis – ISSTA 2004*, pages 129–132, New York, USA, 2004. ACM Press.

[36] Haskell: An Advanced, Purely Functional Programming Language. `https://www.haskell.org`.

[37] L. Heerink. *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.

[38] A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing Real-Time Systems Using UPPAAL. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer-Verlag, 2008.

[39] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[40] J. Hughes, B.C. Pierce, T. Arts, and U. Norell. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *IEEE Int. Conf. on Software Testing, Verification and Validation – ICST*, pages 135–145. IEEE, 2016.

[41] International Organization for Standardization. *ISO/IEC 25010:2011*. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. ISO, Geneva, 2011.

[42] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard IS-8807. ISO, Geneve, 1989.

[43] R. Janssen and J. Tretmans. Matching Implementations to Specifications: The Corner Cases of *ioco*. In *ACM/SIGAPP Symp. on Applied Computing – Software Verification and Testing Track*, SAC'19, pages 2196–2205, New York, NY, USA, 2019. ACM.

[44] C. Jard 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*, 7(4):297–315, 2005.

[45] M. Krichen and S. Tripakis. Black-Box Conformance Testing for Real-Time Systems. In *11th Int. SPIN Workshop on Model Checking of Software – SPIN'04*, volume 2989 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[46] R. Langerak. A Testing Theory for LOTOS using Deadlock Detection. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 87–98. North-Holland, 1990.

[47] D. Lee and M. Yannakakis. Principles and Methods for Testing Finite State Machines – A Survey. *The Proceedings of the IEEE*, 84(8):1090–1123, August 1996.

[48] N.A. Lynch and M.R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989. Also: Technical Report MIT/LCS/TM-373 (TM-351 revised), Massachusetts Institute of Technology, Cambridge, U.S.A., 1988.

[49] L. Marsso, R. Mateescu, and W. Serwe. TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2018*, volume 10806 of *Lecture Notes in Computer Science*, pages 211–228. Springer Int. Publishing, 2018.

[50] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[51] A. Petrenko. Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *Modeling and Verification of Parallel Processes – 4th Summer School MOVEP 2000*, volume 2067 of *Lecture Notes in Computer Science*, pages 196–205. Springer-Verlag, 2001.

[52] M. Phalippou. *Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties.* PhD thesis, L'Université de Bordeaux I, France, 1994.

[53] I. Phillips. Refusal Testing. *Theoretical Computer Science*, 50(2):241–284, 1987.

[54] A. Pnueli. Specification and development of reactive systems. In H.J. Kugler, editor, *Information Processing 86*, pages 845–858. North-Holland, 1986.

[55] J.H. Poore, L. Lan, R. Eschbach, and T. Bauer. Automated Statistical Testing for Embedded Systems. In J. Zander, I. Schieferdecker, and P.J. Mosterman, editors, *Model-Based Testing for Embedded Systems*, pages 111–146. CRC Press, 2012.

[56] Selenium – Browser Automation. `http://www.seleniumhq.org`.

[57] Sikuli Script. `http://www.sikuli.org`.

[58] TorXakis – A Tool for Model-Based Testing. `https://torxakis.org`.

[59] J. Tretmans. HIPPO: A LOTOS Simulator. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 391–396. North-Holland, 1989.

[60] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 127–146. Lecture Notes in Computer Science 1055, Springer-Verlag, 1996.

[61] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[62] J. Tretmans. Model Based Testing with Labelled Transition Systems. In R.M. Hierons, J.P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer-Verlag, 2008.

[63] J. Tretmans and L. Verhaard. A Queue Model relating Synchronous and Asynchronous Communication. In R.J. Linn and M.Ü. Uyar, editors, *Protocol Specification, Testing, and Verification XII*, number C-8 in IFIP Transactions, pages 131–145. North-Holland, 1992.

[64] F. Vaandrager. On the Relationship between Process Algebra and Input/Output Automata. In *Logic in Computer Science*, pages 387–398. Sixth Annual IEEE Symposium, IEEE Computer Society Press, 1991.

[65] Vaandrager, F. Model Learning. *Commun. ACM*, 60(2):86–95, January 2017.

[66] M. Volpato and J. Tretmans. Towards Quality of Model-Based Testing in the IOCO Framework. In *Int. Workshop on Joining AcadeMiA and Industry Contributions to testing Automation – JAMAICA'13*, pages 41–46, New York, NY, USA, 2013. ACM.

[67] R.G. de Vries and J. Tretmans. Towards Formal Test Purposes. In E. Brinksma and J. Tretmans, editors, *Formal Approaches to Testing of Software – FATES'01*, number NS-01-4 in BRICS Notes Series, pages 61–76, University of Aarhus, Denmark, 2001. BRICS.