# Model-Based Testing with TorXakis

## *The Mysteries of Dropbox Revisited*

**Jan Tretmans***
ESI (TNO), Eindhoven (NL)
Radboud University, Nijmegen (NL)
Halmstad University (S)
jan.tretmans@tno.nl

**Piërre van de Laar**
ESI (TNO)
Eindhoven
The Netherlands
pierre.vandelaar@tno.nl

**Abstract.** *Model-based testing is one of the promising technologies to increase the efficiency and effectiveness of software testing. This paper discusses model-based testing in general, it presents the model-based testing tool* TORXAKIS*, and it shows how* TORXAKIS *was applied to test a file synchronization system, Dropbox, revisiting an experiment presented in (Hughes, Pierce, Arts, & Norell, 2016).*

**Keywords.** software testing, model-based testing, modelling, test tool.

# 1 Introduction

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. This may seriously hamper the development and testing of future generations of software systems.

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. The main virtue of model-based testing is that it allows test automation that goes well beyond the mere automatic execution of manually crafted test cases. It 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.

In this paper we first discuss model-based testing in general. Secondly, we present our model-based testing tool TORXAKIS, while touching upon some of its underlying theories such as the **ioco**-testing theory for labelled transition systems (Tretmans, 2008). Thirdly, we show the application of TORXAKIS to model-based testing of the file-synchronization service Dropbox. The first two parts repeat large parts of (Tretmans, 2017); the latter part builds on (Hughes et al., 2016): "Mysteries of Dropbox – Property-Based Testing of a Distributed Synchronization Service", where model-based testing with the tool Quviq QuickCheck (Claessen & Hughes, 2000; Arts, Hughes, Johansson, & Wiger, 2006) is applied for testing Dropbox. It is our initial response to the challenge posed in that paper: "It would be interesting to try to reframe [Quviq QuickCheck's testing of Dropbox] in terms of **ioco** concepts." It is not the aim of this paper to present a completely formal treatment, nor to give a fully scientific account of model-based testing, nor to give definitions or algorithms – we refer to the literature for this – but we will show how a model of Dropbox can be developed for TORXAKIS, and how this model is used to test a distributed and non-deterministic system like Dropbox.

# 2 Model-Based Testing (MBT)

MBT is a form of black-box testing where a System Under Test (SUT) is tested against an abstract model of its required behaviour. The model specifies, in a formal way, what the system shall do, and it is the basis for the algorithmic generation of test cases and for the evaluation of test results. The model is *prescriptive*: it prescribes which inputs the SUT shall accept and which outputs the SUT shall produce in response to those inputs, as opposed to *descriptive* models as used in, e.g., simulation.

The main virtue of model-based testing is that it al-

*This work has been supported by NWO-TTW project 13859: SUMBAT – Supersizing Model-Based Testing.

lows test automation that goes well beyond the mere automatic execution of manually crafted test cases. It allows for the algorithmic generation of large amounts of test cases, including their expected results, completely automatically and correctly from the model of required behaviour.

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, testing can never be complete: testing can only show the presence of errors, not their absence (Dijkstra, 1969).

## 2.1 Benefits of Model-Based Testing

Model-based testing makes it possible to generate tests 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, test cases are valid by construction.

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., adapting test cases when the system evolves, is an important challenge of any testing process. In MBT, maintenance of a multitude of test cases is replaced by maintenance of one model. Finally, various notions of (model-)coverage can be automatically computed, expressing the level of completeness of testing, and allowing better selection of test cases.

## 2.2 Types 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 aspects 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.

In this contribution 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. 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. We do *specification-based*, *black-box* testing, since the externally observable behaviour of the system, seen as a *black-box*, is compared with what has been specified. 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*: there is a formal, well-defined theory underpinning models, SUTs, and correctness of SUTs with respect to models, which enables formal reasoning about *soundness* and *exhaustiveness* of generated test suites.

## 2.3 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, multi-disciplinary processes. In addition, systems continuously evolve and are composed into larger systems and systems-of-systems, whereas system components may come 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 abstractions, 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 and 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.

# 3 Model-Based Testing Theory

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* (Brinksma, Alderden, Langerak, Lagemaat, & Tretmans, 1990). Although an SUT is a black box, we can assume it could be modelled by some model instance in a domain of implementation models. This assumption is commonly referred to as the *testability hypothesis*, or *test assumption* (Gaudel, 1995). 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, i.e., do all correct SUTs pass, and exhaustiveness, i.e., do all incorrect SUTs fail, of test suites is 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 (Chow, 1978; Lee & Yannakakis, 1996; Petrenko, 2001). In this paper 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. Actions can be inputs, outputs, or internal steps of the system.

Conformance for LTS models is precisely defined with the **ioco**-conformance relation (**i**nput-**o**utput-**co**nformance) (Tretmans, 1996, 2008). The conformance relation **ioco** expresses that an SUT conforms to its specification LTS if the SUT never produces an output that cannot be produced by the specification in the same situation, i.e., after the same sequence of actions. A particular, virtual output is *quiescence*, actually expressing the *absence* of real outputs. Quiescence corresponds to observing that there is no output of the SUT, which is observed in practice as a time-out during which no output from the SUT is observed.

The **ioco**-conformance relation supports partial models, under-specification, abstraction, and non-determinism. An SUT is assumed to be modelled as an *input-enabled* LTS (testability hypothesis), that is, any input to the implementation is accepted in every state, whereas 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 form 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 outputs after a sequence of actions, of which an implementation only needs to implement a (non-empty) subset, thus supporting implementation freedom.

The **ioco**-testing theory for LTS provides a test generation algorithm that was proved to be *sound* and *exhaustive*, i.e., the (possibly infinitely many) test cases generated from an LTS model detect all and only **ioco**-incorrect implementations. The **ioco**-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, such as TorX (Belinfante et al., 1999), the Agedis Tool Set (Hartman & Nagin, 2004), TGV (Jard & Jéron, 2005), Uppaal-Tron (Hessel et al., 2008), JTorX (Belinfante, 2010), Axini Test Manager (ATM) (Axini, 2019), TESTOR (Marsso, Mateescu, & Serwe, 2018), and TorXakis (Sect. 4).

# 4 TorXakis Overview

TorXakis is an experimental tool for on-line model-based testing. This section gives a light-weight introduction to and overview of TorXakis. Sect. 5 illustrates TorXakis with the Dropbox example. TorXakis is freely available under a BSD3 license (TorXakis, 2019).

## 4.1 Test Generation

TorXakis implements the **ioco**-testing theory for labelled transition systems (Tretmans, 1996, 2008). More precisely, it implements test generation for symbolic transition systems. The **ioco**-testing theory 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 (Frantzen, Tretmans, & Willemse, 2006). 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. Test generation in TorXakis uses STS following the algorithm of (Frantzen, Tretmans, & Willemse, 2005).

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.

## 4.2 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 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 (Milner, 1989; Hoare, 1985; ISO, 1989). They are a good candidate to serve as a notation for LTS models.

TorXakis uses its own language to express models, which is strongly inspired by the process-algebraic language Lotos (Bolognesi & Brinksma, 1987; ISO, 1989), and which incorporates ideas from Extended Lotos (Brinksma, 1988) and mCRL2 (Groote & Mousavi, 2014). The semantics is based on STS, which in turn has a semantics in LTS. 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, strings, and regular expressions are provided.

Having its roots in process algebra, the language is compositional. It has several operators to combine processes: 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.

## 4.3 Implementation

TorXakis is based on the model-based testing tools TorX (Belinfante et al., 1999) and JTorX (Belinfante, 2010). 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.

In order to manipulate symbolic data and solve constraints for test-data generation, TorXakis uses SMT solvers (Satisfaction Modulo Theories) (De Moura & Bjørner, 2011). Currently, Z3 (De Moura & Bjørner, 2008) and CVC4 (Barrett et al., 2011) are used via the SMT-LIBv2.5 standard interface (C., Fontaine, & Tinelli, 2015). 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 (Haskell, 2019), while parts of TorXakis itself have been tested with the Haskell MBT tool QuickCheck (Claessen & Hughes, 2000).

## 4.4 Innovation

Compared to other model-based testing tools, TorXakis deals with most of the challenges posed in Sect. 2: it supports test generation from non-

deterministic models, it deals with abstraction, partial models and under-specification, it supports concurrency, parallelism, composition of complex models from simpler models, and the combination of constructive modelling in transition systems with property-oriented specification via data constraints.

### 4.5 System Under Test (SUT)

In order to use TorXakis we need a model specifying the allowed behaviour of the SUT. The TorXakis view of an SUT is a black-box that communicates with its environment via messages on its interfaces, i.e., on its input and output channels. 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. A 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.

### 4.6 Model

The model is written in the TorXakis modelling language. A (collection of) model file(s) contains all the definitions necessary for expressing the model: channel, data-type, function, constant, and process definitions, which are all combined in a model definition. In addition, the model file contains some testing specific aspects: connections and en/decodings. A connection definition defines how TorXakis is connected to the SUT by specifying the binding of abstract model channels to concrete sockets. En/decodings specify the mapping of abstract messages (ADTs) to strings and vice versa.

### 4.7 Adapter

Channels in TorXakis are implemented as plain old sockets where messages are line-based strings. However, a lot of real-life SUTs don't communicate via sockets. In those cases, the SUT must be connected via an adapter, wrapper, or test harness that interfaces the SUT to TorXakis, and that transforms the native communication of the SUT to the socket communication that TorXakis expects. Usually, such an adapter must be manually developed. Sometimes it is simple, e.g., transforming standard IO into socket communication using standard (Unix) tools like `netcat` or `socat`, as the Dropbox example will show. Sometimes, building an adapter can be quite cumbersome, e.g., when the SUT provides a GUI. In this case tools like Selenium (Selenium, 2019) or Sikuli (Sikuli, 2019) 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 in an adapter for MBT.

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.

### 4.8 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.

## 5 Testing Dropbox

We apply model-based testing with TorXakis to test Dropbox. Our work closely follows the work in (Hughes et al., 2016), 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.

### 5.1 Dropbox

Dropbox is a file-synchronization service (Dropbox, 2019), 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.

## 5.2 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 (Hughes et al., 2016), where a formal model for a synchronization service was developed and used for model-based testing of Dropbox with the tool Quviq QuickCheck (Arts et al., 2006). 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 (Hughes et al., 2016), we restrict testing to one file and three nodes, and we use actions (SUT inputs) $\text{READ}_N$, $\text{WRITE}_N$, 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 (Hughes et al., 2016)).

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, see Sect. 4, 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 (Hughes et al., 2016), we implement this by simply waiting for at least 30 seconds.

```
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 1:** Dropbox model - channels and their types.

## 5.3 Modelling

Our Dropbox model is a straightforward translation of (Hughes et al., 2016, Section IV: Formalizing the specification) into the modelling language of TorXakis. Parts of the model are shown in Figs. 1, 3, 4, 5, 6, 7, and 8.

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. 1. 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.

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

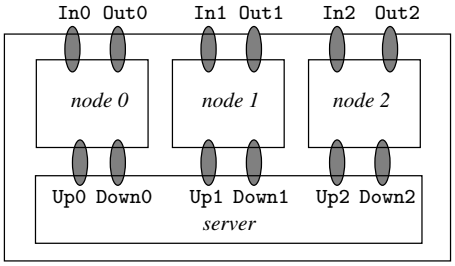


**Figure 2:** Dropbox structure.

`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. 2. The outer box is our SUT.

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. 3, 4, and 5. The state machine is defined as a recursive process `dropbox` with channel parameters `In0`, . . ., `Up2`, and with state variables exactly as in (Hughes et al., 2016):

- 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.

Consider file-reading for node 0 (Fig. 3). The first action is input `Read` on channel `In0`. Then the SUT will produce output `File(lookup(localVal,Node(0)))`, i.e., the

```
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 3:** Dropbox model - main process `dropbox` with transitions `Read` and `Write`.

`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. 6), 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. 6 gives as examples the functions `lookup` and `update`; other functions are included in the full model (TorXakis Examples, 2019). After the output there is the recursive call of process `dropbox`, where state parameters are not modified in case of file-reading.

Writing a file for node 0 is analogous, but with two differences (Fig. 3). 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

```
  .....
##
  [[ 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 4:** Dropbox model - transitions `Down` and `Up` in the main process `dropbox`.

`write`-command must be a valid value, which means (see Fig. 1) 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.

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. 4) 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

```
  .....
##
  [[ 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 5:** Dropbox model - transition `Stabilize` and and process `fileAndConflicts`.

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 (Hughes et al., 2016) where it is very well explained, so for more details we refer there.

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. 5. 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 `EXIT`s, which is the trigger for the recursive call of `dropbox` after `fileAndConflicts`.

```
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 6:** Dropbox model - data types and functions.

The next step is to define the complete model in the MODELDEF; see Fig. 7. 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 $\tau$). 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 $\tau$ is no longer enabled in the state after the $\tau$-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 environment.

The last definition CNECTDEF specifies how the tester connects to the external world via sockets; see Fig. 8. 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 $\langle$txs0-pc, 7890$\rangle$ 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.

```
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 7:** Dropbox model - definition of the model that specifies the behaviour over the observable channels.

## 5.4 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. 9. 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 ob-

```
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 8:** Dropbox model - connection to the external world.

viously taken place, which can only happen using hidden `Up` and `Down` actions. Due to the distributed nature of Dropbox and its nondeterminism it 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 (Hughes et al., 2016)) can make that computation of the next action takes a minute.

## 5.5 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

```
$ 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 <<
```

**Figure 9:** TORXAKIS test run of *Dropbox*.

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 (Hughes et al., 2016).

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.

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?

# 6 Concluding Remarks

We discussed model-based testing, its principles, benefits, and some theory. We showed how the model-based testing tool TORXAKIS can be used to test a file synchronization service. TORXAKIS implements **ioco**-test generation for symbolic transition systems, and it supports state-based control flow together with complex data structures, on-the-fly testing, partial and under-specification, non-determinism, abstraction, random test selection, concurrency, and model compositionality. TORXAKIS is an experimental MBT tool, used in applied research, education, and case studies in the (embedded systems) industry. TORXAKIS currently misses good usability, scalability does not always match the requirements of complex systems, and test selection is still only random, but more sophisticated selection strategies are being investigated (Bos, Janssen, & Moerman, 2019; Bos & Tretmans, 2019).

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 software systems. Model-based testing may be one of them.

# References

Arts, T., Hughes, J., Johansson, J., & Wiger, U. (2006). Testing Telecoms Software with Quviq Quickcheck. In *ACM SIGPLAN Workshop on Erlang* (pp. 2–10). ACM.

*Axini.* (2019). `http://www.axini.com`.

Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., ... Tinelli, C. (2011). CVC4. In *CAV* (pp. 171–177). LNCS 6806. Springer.

Belinfante, A. (2010). JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution . In J. Esparza et al. (Eds.), *TACAS* (pp. 266–270). LNCS 6015. Springer.

Belinfante, A., Feenstra, J., Vries, R. d., Tretmans, J., Goga, N., Feijs, L., ... Heerink, L. (1999). Formal Test Automation: A Simple Experiment. In G. Csopaki et al. (Eds.), *Testing of Communicating Systems* (pp. 179–196). Kluwer.

Bolognesi, T., & Brinksma, E. (1987). Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, *14*, 25–59.

Bos, P. v. d., Janssen, R., & Moerman, J. (2019). $n$-Complete Test Suites for IOCO. *Software Quality Journal*, *27*(2), 563–588.

Bos, P. v. d., & Tretmans, J. (2019). Coverage-Based Testing with Symbolic Transition Sys-

tems. In D. Beyer et al. (Eds.), *TAP*. LNCS 11823. Springer.

Brinksma, E. (1988). *On the Design of Extended LOTOS* (Unpublished doctoral dissertation). University of Twente, Enschede (NL).

Brinksma, E., Alderden, R., Langerak, R., Lagemaat, J. v. d., & Tretmans, J. (1990). A Formal Approach to Conformance Testing. In J. de Meer et al. (Eds.), *Protocol Test Systems* (pp. 349–363). North-Holland.

C., B., Fontaine, P., & Tinelli, C. (2015). *The SMT-LIB Standard: Version 2.5* (Tech. Rep.). Dept. of Comp. Sc., Uni. of Iowa. `www.SMT-LIB.org`.

Chow, T. (1978). Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. on Soft. Eng.*, *4*(3), 178–187.

Claessen, K., & Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ACM SIGPLAN Functional Programming* (pp. 268–279). ACM.

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In C. Ramakrishnan et al. (Eds.), *TACAS* (pp. 337–340). LNCS 4963. Springer.

De Moura, L., & Bjørner, N. (2011). Satisfiability Modulo Theories: Introduction and Applications. *Comm. ACM*, *54*(9), 69–77.

Dijkstra, E. (1969). *Notes On Structured Programming – EWD249* (Report No. 70-WSK-03). Eindhoven (NL): T.H. Eindhoven.

*Dropbox.* (2019). `https://www.dropbox.com`.

Frantzen, L., Tretmans, J., & Willemse, T. (2005). Test Generation Based on Symbolic Specifications. In J. Grabowski et al. (Eds.), *FATES* (pp. 1–15). LNCS 3395. Springer.

Frantzen, L., Tretmans, J., & Willemse, T. (2006). A Symbolic Framework for Model-Based Testing. In K. Havelund et al. (Eds.), *FATES/RV* (pp. 40–54). LNCS 4262. Springer.

Gaudel, M.-C. (1995). Testing can be Formal, too. In P. Mosses et al. (Eds.), *TAPSOFT* (pp. 82–96). LNCS 915. Springer.

Groote, J., & Mousavi, M. (2014). *Modeling and Analysis of Communicating Systems*. MIT Press.

Hartman, A., & Nagin, K. (2004). The AGEDIS Tools for Model Based Testing. In *ISSTA* (pp. 129–132). ACM.

*Haskell.* (2019). `https://www.haskell.org`.

Hessel, A., Larsen, K., Mikucionis, M., Nielsen, B., Pettersson, P., & Skou, A. (2008). Testing Real-Time Systems Using UPPAAL. In R. Hierons et al. (Eds.), *Formal Methods and Testing* (pp. 77–117). LNCS 4949. Springer.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Hughes, J., Pierce, B., Arts, T., & Norell, U. (2016). Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *IEEE ICST* (pp. 135–145). IEEE.

ISO. (1989). *Inf. Proc. Syst., OSI, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. Int. Standard IS-8807.

Jard, C., & Jéron, T. (2005). TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. *Software Tools for Tech. Trans.*, *7*(4), 297–315.

Lee, D., & Yannakakis, M. (1996). Principles and Methods for Testing Finite State Machines – A Survey. *Procs. of IEEE*, *84*(8), 1090–1123.

Marsso, L., Mateescu, R., & Serwe, W. (2018). TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation. In D. Beyer et al. (Eds.), *TACAS* (pp. 211–228). LNCS 10806. Springer.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall.

Petrenko, A. (2001). Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography. In F. Cassez et al. (Eds.), *MOVEP* (pp. 196–205). LNCS 2067. Springer.

*Selenium.* (2019). `http://www.seleniumhq.org`.

*Sikuli.* (2019). `http://www.sikuli.org`.

*TorXakis.* (2019). `https://github.com/TorXakis/TorXakis`.

*TorXakis Examples.* (2019). `https://github.com/TorXakis/TorXakis/tree/develop/examps`.

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

Tretmans, J. (2008). Model Based Testing with Labelled Transition Systems. In R. Hierons et al. (Eds.), *Formal Methods and Testing* (pp. 1–38). LNCS 4949. Springer.

Tretmans, J. (2017). On the Existence of Practical Testers. In J.-P. Katoen et al. (Eds.), *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday* (pp. 87–106). LNCS 10500. Springer.