Côte de Resyste in Progress

René de Vries, Jan Tretmans,
Axel Belinfante, Jan Feenstra
University of Twente
Formal Methods and Tools Group
Faculty of Computer Science
P.O. Box 217, NL-7500 AE Enschede
{rdevries|tretmans| belinfan|feenstra}
@cs.utwente.nl

Loe Feijs, Sjouke Mauw, Nicolae Goga
Eindhoven University of Technology
Eindhoven Embedded Systems Institute
P.O. Box 513, NL-5600 MB Eindhoven
{feijs|sjouke|goga}@win.tue.nl

Lex Heerink
Philips Research Laboratories
Prof. Holstlaan 4
NL-5656 AA Eindhoven
lex.heerink@philips.com

Arjan de Heer
Lucent Technologies
R&D Centre Twente
Capitool 5, NL-7521 PL Enschede
arieheer@lucent.com

Abstract— **Traditional (manual) testing of software systems is a costly, laborious and error-prone activity. It gets even more complicated nowadays with complex reactive software like embedded system software and communication protocols. Such software is characterized by a high degree of interactivity and concurrency.** *Côte de Resyste* **aims at developing methods and an integrated tool environment to support and, whenever possible, automate the testing process of reactive systems. Contrary to other test tools, this tool environment builds on a sound and well-defined theoretical basis. First results are the implementation of the prototype test tool TorX and its successful application to the Philips' A/V Link protocol and to the academic** *Conference protocol* **case study. These experiments show that test automation, based on formal methods, is feasible and beneficial.**

Current work concentrates on improving TorX, on developing methods for effective selection of test sets and on generic test execution environments. A major part of the project is devoted to industrial case studies which are executed in close cooperation with Philips, Lucent Technologies and Interpay.

Keywords— **conformance testing, test automation, formal methods**

I. Introduction

Testing is an important activity for checking the correctness of software. Nowadays a lot of software is embedded in systems. Examples of such embedded systems are consumer electronics equipment and mission critical systems. Correctness of such *embedded systems* becomes increasingly important when the risks of malfunctioning, and the associated costs, increase. Since we demand more and more functionality of these systems, the amount of software and its complexity increases and as a result it is harder to assure correctness of the software inside. Several methods can be applied to guard correctness during the development phase of the system. *Testing* is one of these methods. Testing is performed by applying test experiments to an implementation under test (IUT) and by making observations during the execution of the tests. Subsequentially a verdict is assigned about the correct functioning of the implementation, based on its specification and a correctness notion.

Côte de Resyste (COnformance TEsting of REactive SYSTEms) [1] is an STW project with a 4 year scope that comprises 21 fte. It has been running for 2 years now. Participants in the project are Philips Research Laboratories Eindhoven, Eindhoven University of Technology, the University of Twente and Lucent Technologies Enschede. The goal of the project is to develop methods, techniques and tools for testing reactive software systems based on formal methods and to validate these by industrial and academic case studies.

The objective of this paper is to present an overview of the *Côte de Resyste* project and the results that have been achieved so far.

II. Conformance testing

A. Testing

Testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Tests are applied to the implementation under test in a controlled environment, and, based on observations made during the execution of the tests, a verdict about the correct functioning of the implementation is given. The specification is the basis for testing, since it prescribes how the system should behave.

There are many different kinds of testing, e.g. performance testing, stress testing and others. Within the *Côte de Resyste* project we consider *conformance testing* of *reactive systems*. Conformance testing is used to check whether an implementation of a system functionally behaves as specified by its functional specification. It is characterized as a black box testing method, i.e. it is assumed that no information is available about the internal structure of the implementation. Reactive systems are systems that respond to stimuli provided by the system environment. Many systems can be seen as reactive systems, e.g. communication protocols, embedded software systems and process control systems. Concurrency and interaction usually play

an important rôle in such systems.

Intuitively, conformance testing of reactive systems is carried out by offering stimuli to the implementation under test, i.e. give input, and observe the responses to these inputs, i.e. observe its output. When the responses are not as expected according to the behavioural specification (i.e. the implementation under test does not conform to its specification), it is concluded that the implementation under test is incorrect.

B. The conformance testing process

In the process of conformance testing there are two main phases: *test generation* and *test execution*. Test generation involves: analysing the specification and determining the functionalities to be tested, defining a strategy how to test these functionalities, and specifying and developing a set of test experiments (*test suite*). Test execution involves: developing a test environment in which the test suite can be executed, executing the test suite, analysing the execution results, and assigning a verdict about the correctness of the implementation based on the execution results.

C. Problems of testing

Many problems in the testing process occur because specifications are unclear, imprecise, incomplete and ambiguous. Without a specification which clearly, precisely and unambiguously prescribes how a system implementation shall behave, any testing will be very difficult because it is unclear what to test for. Additionally, one should also have a notion of correctness, i.e. have a precise notion of what behaviour conforms to such a specification. Note that specifications may leave implementation freedom; many correct implementations can be associated to the same specification. A clear specification and correctness notion are the basic ingredients for derivation of valid test experiments. Test derivation based on an ambiguous specification and correctness notion is likely to result in an erroneous test suite that, after execution, can lead to an incorrect judgment concerning the correctness of the implementation under test.

The development of valid tests from a specification and a correctness notion is usually a complicated task. For complex systems that are characterized by a high degree of interactivity and concurrency this is even more difficult due to the huge amount of different states in which the system can be (this is known as the *state space explosion* problem).

Once a test suite is obtained, it should be executed on the implementation under test. For this we need a test execution environment. A test execution environment is the environment in which the tester interfaces with the IUT. The tester then executes the test suite (either manually or automatically). The interface between the tester and IUT is not always trivial, e.g., when the IUT cannot be accessed directly, but only via an intermediate environment (the *test context*). Building such an environment is often a difficult task.

Manual test derivation and execution usually is very laborious, costly and error-prone, so automation may help solving part of these problems. Automation of test derivation and test execution activities may help in making the testing process faster, in making it less susceptible to human error by automating routine or error-prone tasks, and in making it more reproducible by making it less dependent on human interpretation. Furthermore, new releases of the system can benefit from automation by reusing the developed tests and test environment (regression testing).

III. Côte de Resyste's approach

One of the *Côte de Resyste*'s solutions of the identified problems is to use *Formal Methods*. Formal methods are concerned with mathematical modelling of software and hardware systems. Due to their mathematical underpinning, formal methods allow to specify systems with more precision, more consistency and less ambiguity. Moreover, formal methods allow to simulate, validate and reason about system models, i.e. to prove with mathematical precision the presence or absence of particular properties in a design or specification. This makes it possible to detect deficiencies earlier in the development process. An important aspect is that specifications expressed in a formal language are processable by tools, hence allowing automation in the software development trajectory, e.g. the testing activity.

When using formal methods in conformance testing we use a specification as basis, i.e. a specification given in a formal specification language. This allows us to reason mathematically about testing. The notion of correctness is defined precisely, which is an *implementation relation*. Consequently, we try to derive a test suite algorithmically from a formal specification following a well-defined and precisely specified algorithm. Well-defined test derivation algorithms guarantee that tests are *valid*, i.e. that derived tests really test what they should test.

There are several theories of formal testing. *Côte de Resyste* adopts the so-called **ioco** testing theory [2]. The underlying formalisms are labelled transition systems, which model systems in terms of sequences of events (potential test actions). This theory enables us to discriminate among systems not only based on (erroneous) output, but also on absence of output, called quiescence.

The test suite derivation algorithm of this theory has been proven to generate a sound test suite. A test suite is called *sound* if the verdict never rejects a correct implementation. For practical application, a minimal requirement on a test suite is that it is sound.

There are two main phases in the testing process: *test derivation*, i.e. obtaining a test suite, and *test execution*, i.e. applying the test suite to the IUT. Both phases can be automated. This can be done in two ways: as two separate phases, which is called *batch testing*, or in an integrated manner, which is called *on-the-fly testing*.

*batch testing***.** In the *test derivation* phase a test suite is derived and stored in some representation, usually the test notation language TTCN. In the *test execution* phase this test suite is executed along with the IUT. Batch test

derivation is computationally expensive and suffers from the state space explosion problem. This complexity can be reduced by user guidance and on-the-fly derivation techniques [3].

on-the-fly testing. As opposed to batch testing, test derivation and test execution occur simultaneously. Seldom, all the information in a test case (one test scenario in a test suite) is needed during an execution. In fact just a minor part is needed. Instead of deriving a complete *test case* (one test scenario in a test suite), the test derivation process derives *test primitives* from the specification. Test primitives are basic building blocks for test cases that can be applied immediately to the IUT. Examples of test primitives are inputs (that can be used as stimuli) and outputs (that can be used to check observations). While executing a test case, only the necessary part of a test case is considered: the test case is derived *lazily*. Observations during the test execution allows us to reduce the effort needed to derive test information from the specification compared with batch derivation; see also [4].

Batch testing and on-the-fly testing have both their advantages and disadvantages. Firstly, the batch-wise approach is better suited for manual test suite preparation and for semi-automatic test suite preparation. Humans are good at test selection but they are not fast enough to do it at run-time which is required for on-the-fly testing.

Secondly, test implementation is easier in the batch-wise approach. Translation of test actions is necessary to interface the tester with the IUT. For batch-wise test derivation it is possible to compile the abstract test cases into concrete test cases which have all the mapping details encoded. For the on-the-fly approach the translation has to be done at run-time. Such a run-time translator has to meet timing constraints and should be generic enough to allow reuse.

Thirdly, in case of on-the-fly testing all computations have to be done at run-time, whereas batch-wise testing allows some of the work to be moved to compile-time. So, the batch-wise approach has an advantage which makes it easier to satisfy the IUT's real-time requirements. But the price to pay is that many test-steps which will not happen at run-time are pre-computed. This leads to test-suites of an enormous size, and the amount of pre-computation work and the storage demands involved may well undo the advantage. On-the-fly testing is here of help to fight this state space explosion.

To reduce the effort for engineering a test environment, *Côte de Resyste* develops the TorX architecture and implements a prototype according to this architecture, also called TorX. TorX is discussed in the next section.

Côte de Resyste validates the developed theory, methods, techniques and tools for practical applicability. This is done by executing industrial and academic case studies. The results obtained by these studies are steering the development of new theory and improved tools.

IV. TorX

TorX is a test tool architecture which can be instantiated in two ways: to support on-the-fly testing and to support batch-wise testing. First, we discuss the architecture, next we discuss the existing configurations in which this architecture has been instantiated.

A. TorX *Architecture*

The main characteristics of TorX are its flexibility and openness. Flexibility is obtained by requiring a modular architecture with well-defined interfaces between the components – this allows easy replacement of a component by an improved or modified version (e.g. one that supports another specification language or test generation algorithm). Openness is achieved by using, when possible, industrial standard interfaces to link the components in our tool environment – this enables integration of 'third party' components that implement these interfaces. When no standard interface is available we connect components by pipes over which textual commands and responses are exchanged – these textual interfaces make it simple to debug and test individual components, to experiment using (Unix style) filters to massage the information exchanged, and even to split the tool over several machines.

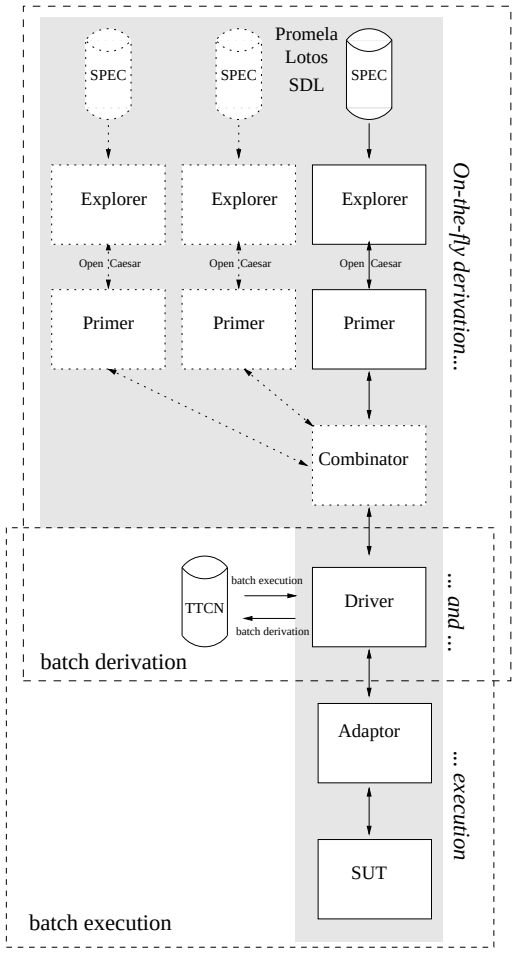

Fig. 1. TorX tool architecture

The TorX architecture consists of the following components: Explorer, Primer, Combinator, Driver, Adaptor, and TTCN storage. Figure 1 depicts how

these components are linked for batch derivation (with EXPLORER, PRIMER COMBINATOR, DRIVER and TTCN storage), batch execution (with TTCN storage, DRIVER, ADAPTOR), and on-the-fly derivation and execution (involving all components without storage of TTCN). The SUT is the system under test, i.e. it is the IUT together with a surrounding environment, the so called test context. We now discuss each component of the TORX architecture and its interfaces in specific.

Explorer. The EXPLORER is a specification language-specific component that offers functions (to the PRIMER) to explore the transition-graph of a specification and to provide, for a given state, the set of transitions (actions). Currently, we have several interfaces between EXPLORER and PRIMER. One of them is the Open/Caesar interface [5], which is a C API that provides exactly such state and transition functions for labeled transition systems. This implies that we can use existing tools that support this interface to implement the EXPLORER. For LOTOS such a tool exists and is available; for SDL such a tool exists, but, to our knowledge, is not publicly available. The other EXPLORER-PRIMER interfaces are ad hoc; we will discuss them in the section on TORX Configurations.

Primer. The PRIMER uses the functions provided by the EXPLORER (in particular, to receive sets of enabled actions) to implement the test derivation algorithm. It offers test primitives to the DRIVER (or COMBINATOR), e.g. functions to generate inputs (stimuli) for the implementation, and to check outputs (observations) from the implementation.

Combinator. The optional COMBINATOR can be used to steer the on-the-fly derivation process. The COMBINATOR is connected between the DRIVER and one or more PRIMERS. One PRIMER represents the formal specification. Each other PRIMER represents a property of interest, a test purpose. The COMBINATOR combines the test primitives from the PRIMERS and offers them to the DRIVER. From the PRIMERS point of view the combinator plays the role of a DRIVER, and vice versa.

Driver. The DRIVER is the central component of the tool architecture. It controls the progress of the testing process. It decides whether to do an input action or to observe and check an output action from the implementation. The DRIVER can be run in two modes: a manual mode, in which the user is in full control, and an automatic mode, in which the DRIVER makes all necessary choices randomly. The test trace that is derived and executed can be logged and replayed during a subsequent test run. The DRIVER uses the PRIMER to obtain an input and to check whether the output of the implementation is correct. It uses the ADAPTOR to execute inputs by sending these inputs to the IUT, and to observe outputs that are generated by the IUT. For batch testing the derived tests are first stored in a file system (indicated as "TTCN" in Figure 1). To execute tests in batch mode the test actions are obtained from storage rather than from the PRIMER.

The PRIMER-DRIVER interface is a textual one.

Adaptor. The ADAPTOR provides the connection with the SUT. It is responsible for sending inputs to and receiving outputs from the SUT on request of the DRIVER. The ADAPTOR is also responsible for encoding and decoding of abstract actions from the DRIVER to concrete bits and bytes for the SUT, and vice versa, including the mapping of time-outs onto *quiescent* actions, see [2]. The connection part of the ADAPTOR is specification independent. The en/decoding routines are specification dependent: they depend on the abstract actions that they have to en/decode, which will vary between specifications.

Currently, We use two interfaces between DRIVER and ADAPTOR. The first one is a API of calling conventions for (Tcl) functions that implement en/decoding functions. The second one supports (a subset of) the C language bindings (API) of the Generic Compiler/Interpreter Interface (GCI) [6].

We plan to decouple the DRIVER and ADAPTOR into separate (Unix) processes connected by pipes. The interface between these components will be similar to the PRIMER-DRIVER interface. This should make it easier to connect a simulator as SUT (see below), and to connect a filter between DRIVER and ADAPTOR, e.g. to translate between abstract PROMELA actions from the specification, and abstract LOTOS actions from a simulator that is used as SUT.

SUT. Most implementations that we use are "real" ones, that we connect to TORX using an ADAPTOR. In addition to these, we have made a LOTOS simulator that we can use as implementation. It is mainly used for educational purposes, to illustrate the testing power of TORX: when we have a LOTOS specification of a system, it is now very easy to make a "mutant" that behaves in a slightly different way. The challenge is then to predict whether TORX can detect the difference.

The simulator is currently connected to the DRIVER via its own ADAPTOR. The simulator supports a subset of the PRIMER-DRIVER interface but it also requires a simulator-specific ADAPTOR. Changing the DRIVER-ADAPTOR interface to a PRIMER-DRIVER compatible interface makes the simulator-specific ADAPTOR superfluous.

B. TORX *Configurations*

As Figure 1 shows, the components can be put together in three different ways: for on-the-fly testing, for batch test derivation and for batch test execution. We now describe how these configurations have been used for testing based on formal specifications in LOTOS, PROMELA and SDL. We will pay special attention to the differences and similarities in the use of the basic building blocks.

On-the-fly testing for LOTOS and PROMELA is based on the implementation relation **ioco** [2]. In both cases, the same DRIVER is used, and major parts of the ADAPTOR, but the EXPLORER and PRIMER are different. We use a single EXPECT implementation of the DRIVER. In our experience, EXPECT allows rapid prototype development and interaction with external programs. The ADAPTOR is implemented as a (Tcl) library of the DRIVER, and uses separate programs (in EXPECT style) to connect to the SUT,

one for each supported protocol (e.g. TCP, UDP). Those connection programs can be controlled via standard input and output.

TorX has been instantiated with configurations based on specifications in LOTOS (both non-symbolic and symbolic), Promela (on-the-fly only) and SDL (batch only). These configurations are discussed below.

LOTOS, non-symbolic. The specification-dependent Explorer component can be automatically generated from a LOTOS specification using the CADP tool set. This Explorer component is linked with the **ioco**-Primer using the Open/Caesar interface. The Primer component is independent of the specification, and is in principle even specification language independent (for all specification languages for which there is a compiler that compiles to the Open/Caesar interface).

The CADP generated Explorer has not-so-nice state space explosion properties, because it cannot handle free variables. CADP expands expands free variables by enumerating the values in the domains of the variables after which it generates an action for each possible combination of these values, which may lead to a state space explosion.

LOTOS, symbolic. We are currently performing initial experiments using the Smile symbolic LOTOS simulator [7] as Explorer component, in an attempt to overcome the problems attached to the non-symbolic approach. The actions generated by Smile may contain free variables, which are instantiated during test execution, which helps to reduce the state space explosion problem. A new, symbolic, Primer is connected to Smile using its (ad hoc) Tcl scripting interface. This is on-going research.

Promela. For Promela on-the-fly testing, we can automatically generate a single component that implements both the Explorer and Primer (thereby hiding the interface between them). This is done by the Trojka tool which is described in detail in [4]. If the tool Spin were able to supply an Open/Caesar interface for Promela, we would have been able to use the same Primer as for LOTOS. Such an extension is left for future work. The actions derived by the Trojka generated Explorer may contain free variables, which are instantiated during test execution, which helps to reduce the state space explosion problem.

Batch LOTOS testing. For LOTOS batch derivation we use the TGV tool [3] which covers the functionality of the Explorer, Primer and (derivation) Driver. It derives tests in a format that can be read by the CADP tool set.

For LOTOS batch execution we treat these TGV-generated tests as specifications that contain precomputed test primitives. We execute the tests using our on-the-fly derivation and execution method, with one important difference: we don't use an **ioco**-Primer, but instead we use a Primer that directly uses the precomputed test primitives from the specification (via the Explorer). As before, the Explorer is automatically generated using the CADP tool set, and the same Driver and Adaptor can be used as for the on-the-fly testing.

Batch SDL testing. For SDL batch testing we use the Tau tool set to derive and execute test suites in TTCN. The functionality of the Explorer, Primer and Driver is covered by Tau's Autolink test derivation tool [8]. It generates TTCN test suites, guided by *message sequence charts* (MSCs) that have to be provided by the user. These MSCs can be derived by hand from the SDL specification using the SDL simulator that is integrated in Tau, or in a fully automatic way for predefined paths in the specification.

The batch test execution Driver component is automatically generated from the complete TTCN suites by the TTCN compiler of Tau. This Driver is linked with an Adaptor component using the GCI interface; the result is a single program that can execute the test suites. The Adaptor differs from the one for the on-the-fly testers in the following aspects. Firstly, the Adaptor uses the GCI interface. Secondly, the Adaptor does not use external programs to provide the connection to the SUT because generic support for several connection types (protocols) has been built in. Finally, the en/decoding routines are implemented in C instead of Tcl.

V. The EasyLink case study

This section describes a case study that has been executed as part of the *Côte de Resyste* project. The aim of the case study was to check whether Audio Video (AV) devices are able to correctly communicate with each other by applying the TorX tool, that is being developed in *Côte de Resyste*, to identify potential problems in applying the tool, and to suggest and implement improvements in order to overcome these deficiencies.

A. Context

Customers of AV devices often purchase their devices from many different manufacturers, connect them in different combinations, at different moments in time. This requires an interconnection system that is extendable and suited for the interconnection of existing devices as well as new devices. The AV.Link standard specifies a protocol for control-oriented data communication between a chain of AV devices (such as a TV, a VCR, a satellite decoder, etc.) that meet these requirements. Communication is defined over a single, unused pin of the peritelevision connector (which is present on most AV devices). Figure 2 depicts three AV devices hooked up by means of AV.Link.

Fig. 2. An example AV.Link configuration

The AV.Link standard allows the definition of specific application protocols on top of AV.Link. One of these ap-

plication protocols is *EasyLink*. This proprietary protocol has first been defined in 1996 by Philips to facilitate communication between a TV and one or more VCRs, and has been licensed under applicable patents as a defacto standard to many other Consumer Electronics companies since then. Commercial implementations of the EasyLink protocol are available under different names from different vendors, e.g. EasyLink (Philips), ShowView (Sharp), etc.

B. Formal testing of the EasyLink protocol using TorX

EasyLink has been developed to provide the user of AV devices with additional services that make them easy to program and use in combination. One of the features provided by EasyLink (v1.3) is the *Preset Download* feature. This feature allows for the automatic download of predefined settings (presets) such as channel number, frequency, etc. from the TV to one or more VCRs. It can be initiated either by one of the connected VCRs (by issueing a request to the TV to send its presets) or by the TV (autonomously, e.g. if the preset list of the TV changes it automatically informs the connected devices about these changes). The Preset Download, in combination with user behaviour to modify the preset list of the TV, has been tested with TorX in this case study.

To test the Preset Download feature of the EasyLink protocol a test environment has been set up. The test environment consists of a TV that is indirectly connected to a single VCR via an intermediate device (MBB). This intermediate device is also connected to, and controlled by, a host computer running our tester (TorX) via a serial link. The MBB allows the host to monitor the messages between the TV and the VCR. It also allows the host to upload messages to the TV or VCR. The TV can be operated by a uni-directional remote control remote control. The remote control is controlled by the host computer via a human interface. Figure 3 depicts the test environment that was used.

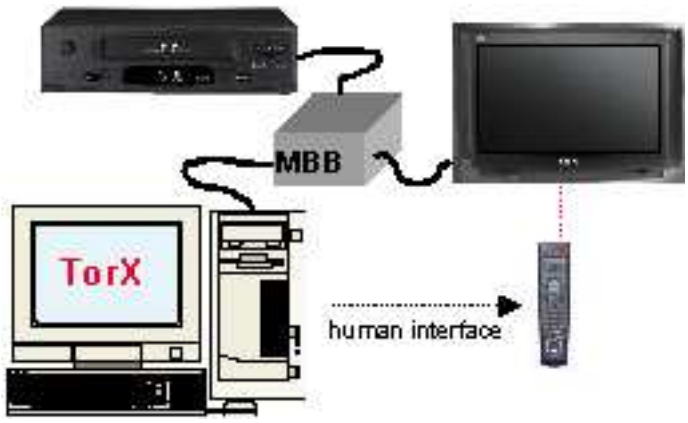

Fig. 3. EasyLink test environment

To test the Preset Download feature the relevant behaviour of the TV has been modelled as a formal specification in the language Promela. From this specification, TorX is able to generate automatically the stimuli to the MBB or the remote control on-the-fly. The stimuli to the MBB are executed automatically, for the stimuli to the remote control a human interface is needed. The responses of the TV obtained as a result to these stimuli are detected by TorX and checked for correctness. In that way, TorX is able to (semi)automatically check the correctness of the EasyLink implementation for the TV.

C. Results

The aim of the case study was to validate the tool TorX in a realistic setting and to suggest and implement improvements to the tool as a whole. This case study did led to several significant improvements to the tool TorX, e.g. a restructuring of the Adaptor part of TorX (so that it is easier to use), and the extension of TorX to cope with human interfaces in a natural way. Furthermore, experiments to split TorX over multiple machines have been carried out successfully. Also, improvements to the specification language have been carried out which led to performance improvements of the tool. One of these improvements was the ability to cope with symbolic outputs [4], which led to a significant reduction in the state space of the specification (and hence to a significant performance improvement).

Apart from tool improvements some errors have been found in the protocol as well. These findings will be reported to a dedicated Test Competence Center for AV devices within Philips, and our results will be compared with theirs.

VI. Conclusions

In this paper we presented the *Côte de Resyste* project. We gave an overview of testing, its problems and its approach to these problems. From the completed case studies we can conclude that the approach is applicable and that automation is feasible. Furthermore the test architecture turns out to be flexible and very modular.

Current project work concentrates on test selection, symbolic test derivation and improving TorX. Testing a realistic implementation exhaustively is practically unfeasible. Therefore, selections have to be made: which tests are going to be applied, and which are not? Such selections can be based on system properties (test purposes), strategies or coverage measures. Research is needed how to select suitable test suites based on these selection criteria. Furthermore, to be able to test bigger systems, the state space explosion problems has to be reduced. Symbolic test derivation is a technique that is able to reduce the number of states, and research is ongoing to implement such techniques within *Côte de Resyste*. Another research issue concerns the analysis of faults in order to identify and locate their causes. In particular, we would like to have a theory to relate faults back to the specification.

Currently, we carry out case studies at Philips, Lucent Technologies and Interpay to continue validation of the developed tools, techniques and theory in the *Côte de Resyste* project, and to implement improvements.

References

[1] Dutch Technology Foundation STW, "*Côte de Resyste* – COnformance TEsting of REactive SYSTEms," Project proposal STW TIF.4111, University of Twente, Eindhoven University of Technology, Philips Research Laboratories, KPN Research, Utrecht, The Netherlands, 1996, `http://fmt.cs.utwente.nl/CdR`.

[2] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software—Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996, Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[3] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho, "Using on-the-fly verification techniques for the generation of test suites," in *Computer Aided Verification CAV'96*, R. Alur and T.A. Henzinger, Eds. Lecture Notes in Computer Science 1102, 1996, Springer-Verlag.

[4] Tretmans J. Vries, de R.G., "On-the-fly conformance testing using SPIN," *Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 382–393, March 2000.

[5] H. Garavel, "OPEN/CÆSAR: An open software architecture for verification, simulation, and testing," in *Fourth Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, B. Steffen, Ed. Lecture Notes in Computer Science 1384, 1998, pp. 68–84, Springer-Verlag.

[6] F. Brady and R.M. Barker, "Infrastructural tools for information technology and telecommunications conformance testing, INTOOL/GCI, generic compiler/interpreter interface (GCI) interface specification, version 2.2," 1996, INTOOL document number GCI/NPL038v2.

[7] E.H. Eertink, *Simulation Techniques for the Validation of LOTOS Specifications*, Ph.D. thesis, University of Twente, Enschede, Netherlands, March 1994.

[8] M. Schmitt, A. Ek, B. Koch, J. Grabowski, and D. Hogrefe, "– AUTOLINK – Putting SDL-based Test Generation into Practice," in 11^{th} *Int. Workshop on Testing of Communicating Systems*, A. Petrenko and N. Yevtushenko, Eds. 1998, pp. 227–243, Kluwer Academic Publishers.