This pre-print paper will appear as Chapter 16, pp. 453-483, in Part IV of the book "Model-based Testing for Embedded Systems" by Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman (eds.), CRC Press, 2011/2012.

Section 5, as well as several Figures and a number of Acknowledgments, which will appear in the above-mentioned book, are not shown in this pre-print paper.

Model-Based Passive Testing of Safety-Critical Components

Stefan Gruner and **Bruce Watson**

Department of Computer Science

Universiteit van Pretoria

Republic of South Africa

Motivation and Overview of this Chapter. Passive testing is a complementary technique to active testing. Under the term 'active testing' we know the 'classical' forms of testing whereby a newly developed system is scrutinized and evaluated *before* its final installation and deployment. For some types of systems, for example dynamic or adaptive distributed systems which are able to re-configure themselves at runtime in response to changes in their environments, exhaustive active testing before deployment is either theoretically impossible or practically not feasible. For such types of systems the additional application of the technique of passive testing is recommendable, whereby the system is observed at runtime (after deployment) and the correctness of its operations is assessed on the basis of *traces* (messages, reports) which the passively tested system continues to emit during its operations. Similar to active testing, also passive testing can be planned and designed in a *model-based* fashion, which is the topic of this chapter. However, a comprehensive theory and taxonomy of methods and techniques for model-based passive testing does –as far as we know– not yet exist and is from today's perspective still very much a topic for future research in this domain. For this reason the presentation of the topic in this chapter is very much *example*-based such as to provide the reader with some 'first intuitions' about what model-based passive testing is, what kinds of techniques could be used to implement it, and what could be some typical application scenarios for model-based passive testing in the domains of software systems, hardware systems, as well as embedded software+hardware systems. On these premises, this chapter is structured and organised as follows:

- **Section 1** introduces the *general principles* of model-based passive testing.
- **Section 2** discusses model-based passive testing in the domain of *software systems* and provides examples of applicable techniques for the implementation of passive testing in this domain.
- **Section 3** discusses the relevance of model-based passive testing techniques for another increasingly important application domain, namely the domain of *network security*.
- **Section 4** discusses the topic of *hardware systems* and the relevance of model-based passive testing for this domain,
- **Section 5** **(which is not shown in this pre-print paper)** presents stepwise, especially from an *educational perspective* for the benefits of students, a longer, *illustrative example* of the model-based passive testing of a small concurrent system in form of a *cellular automaton*. By following the steps of this educative example, the student should be able to reach a deeper understanding of the general principles of model-based passive testing as they are outlined in the introductory section.
- **Section 6** discusses from a theoretical or methodological point of view the *limits of applicability* of model-based passive testing as a general technique; it is important to keep in mind that also model-based passive testing is not (and cannot be) the proverbial 'silver bullet' to solve all problems in systems quality assurance.
- **Section 7** briefly recapitulates and concludes this chapter in several points; these points can also be fast-read as 'exective summary' by industrial technology managers who cannot go into all the theoretical details of the more 'academic' sections of this chapter.

1. Introduction

In *model-based testing* (MBT, also known as 'specification-based' testing, or 'model-driven' testing: MDT), the *test cases*, according to which a hardware or software unit (module, component) shall be tested after its development or implementation, are typically not created 'ab initio'. They are derived by some means of formal reasoning from a formal model, the test model, which is more or less closely related to the specification model of that unit, from which its implementation had been derived (Augusto et al. 2004). This principle is known now since about 20 years; see for example (Bochmann et al. 1989), or (Richardson et al. 1989). This general idea behind all these approaches is sketched below in Figure 1.

[INSERT FIGURE-1 HERE]

Figure 1: *The principle of model-based testing (MBT).*

Since several years, specialist workshops on MBT are being held, whereby the techniques to support model-based testing are adopted from diverse areas such as formal verification, model checking, control and data flow analysis, formal grammar analysis, or Markov decision processes. More specifically those techniques include:

- Methods for online and offline test sequence generation,
- Methods for test data selection,
- Runtime verification,
- Domain partition analysis,
- Combinations of verification and testing,
- Models as test oracles,
- Scenario-based generation of test cases,
- Meta-programming support for testing,
- Game-theoretic models of test strategies,
- etc.

Two recent textbooks on MTB are (Baker et al. 2008), which is suitable also for students and project managers, as well as (Broy et al. 2005), which rather addresses topic experts and specialists.

Model-based testing, as a concept or method or technique, is rather general and does not imply anything about the type of system on which it is applied, nor on the specific ways in which the test cases will be eventually applied to the system under test. To this end we can further distinguish *hardware* systems (modules) versus *software* systems (modules), and *active* testing versus *passive* testing, in relation to the bottom part of Figure 1.

The differences between active testing and passive testing can be characterized as follows:

- *Active testing* is the classical, development-related process in which a tester (usually a person) examines the functionality of a unit (system, module) under test *before* that unit is being deployed, whereby that unit does (usually) not have the ability to test itself.
- *Passive testing*, on the contrary, can (and usually does) take place *after* the deployment of that unit and requires either the ability to test itself while in operation, or the existence of a monitor unit that operates simultaneously to the other unit that is subject to passive testing.

Passive testing can also be applied during a longer period of time on a system prototype in a pilot-project, before the final release of a system on the basis of such a passively tested prototype. In this way, passive testing could also be useful for improving a system's robustness, i.e.: the property of 'graceful degradation' when the system gets under heavy load or stress, and possibly even for improving a system's resilience, i.e.: the property of 'self-healing' from faults under particular circumstances. The literature on passive testing since (Cavalli et al. 2003) is growing steadily, and the reader can easily find further references on the internet.

As it is further discussed below, active testing can take place as black-box testing or white-box testing, whereas passive testing is always a special form of black-box testing – a useful glossary about many further testing-related concepts and notions is provided by *EGEE* and can be found on the internet.[1] As far as its application type is concerned, passive testing –though in principle applicable anywhere– is even more interesting for the testing of *hardware* components than it is for testing of software modules, and it is also more interesting for *distributed* systems than for classical mono-processor systems. The reason for this special relationship between passive testing one the one hand and hardware systems and distributed systems on the other hand, is twofold:

[1] http://egee-jra1-testing.web.cern.ch/egee-jra1-testing/glossary.html

- For the purpose of assuring the quality of *classical software systems*, low-level source code testing –though still necessary– is not the most important method of quality assurance any longer: Software engineers nowadays are (or, at least, should be) guided by the principle of ‘correctness by construction’ (Shukla et al. 2010) which goes hand-in-hand with valuable and effective development techniques such as source code inspections (Rombach et al. 2008). Those techniques, especially source code inspection, are usually not (or, at least: not easily) applicable in the hardware domain.
- *Distributed systems*, especially those based on asynchronous message passing, are notorious for their possibility of *non-deterministic* behaviour (at least as far as their micro-activities on the small scale of their communication layer are concerned). This nondeterminism implies that classical tests of type ‘active testing’ applied to such systems are typically *not repeatable*. However, as in the physical sciences, the non-repeatability of an ‘experiment’ (here: a classical active test) diminishes its scientific value and usefulness. In such circumstances (i.e.: in which repeatable testing in laboratory conditions is not possible or not feasible) the continuous model-based monitoring of the operations of a running distributed system (i.e.: passive testing) can at least *reduce* (or diminish) the risk of applying such kind of systems in safety-critical circumstances and environments.

On these premises, we can now also make plausible why the main intent of this chapter is directed towards model-based, passive testing of *non*-software components, though software components will be mentioned as well. In this context,

- typical *techniques* of passive testing can be regular expressions and finite state automata (Paradkar 2003), grammars and parsing, model-checking of process traces specified in terms of the formal language *CSP*, and the like, whereas
- typical *objects* of passive testing can be ‘firewalls’ (for intrusion detection), memory controlers of processors (with only their ‘pins’ as interfaces), or the emerging technology of *networks on chips* (NOCs) which are also relevant in the context of the above-mentioned firewall applications.

All these cases and upcoming application scenarios are especially interesting because the possibilities of their *testability* are *not yet systematically researched* and explored, though there are numerous particular contributions and results that point into the direction of a more systematic and comprehensive theory for this domain (Cavalli et al. 2003).

It comes as no surprise that *safety-critical* systems are the typical cases for passive testing. Bruce Watson (co-author of this chapter), for example, has cooperated with the *CISCO* Systems Corporation on a confidential industry project on the topic of intrusion detection by means of *regular expressions*, which is indeed a special case of passive testing. More recently another formal-language-based approached to intrusion detection as a kind of passive system testing has been published (Brzezinski 2007).

It should be obvious that passive testing does not come for free –there are computational and economic costs to be invested into the monitor modules that actually implement the passive testing processes– but those additional costs are justified in those safety-critical cases in which other, even larger and higher values (such as human life) are at risk. Moreover it should be clear –and to this end we issue an explicit warning!– that passive testing *cannot* be used as the proverbial 'silver bullet' to the solution of all safety-critical system problems, because otherwise a circular meta-problem would immediately arise: Who has meta-tested the model-based test devices that are used for the passive testing of the object devices in the correct performance of which we are interested?

The *purpose* of this chapter is mainly *educational*. It is mainly written for university students, and their lecturers, in the faculties of informatics, cybernetics, and/or computer engineering. It is *not* the main purpose of this chapter to publish original results of our own research, as it would be appropriate for a confernce or journal paper. Consequently, the largest parts of this chapter consists of literature reviews, whereby we try to keep the presentation 'student-friendly' (i.e., not too technical nor difficult). Nevertheless we hope that our chapter will also carry some value as 'reference material' for the established researcher in this interesting interdisciplinary field, at the interface between computer software and computer hardware engineering. Industrial technology managers, who are not interested in these 'academic details', may speed-read the points of the conclusion section at the end of this chapter as an 'executive summary'.

2. Model-Based Passive Testing of Software Systems

Testing is always based on what is *observable*. Classically we know of two types of software testing, namely *black-box* testing and *white-box* testing (Amman et al. 2008). More recent literature also describes the concepts of *grey-box* testing and *glass-box* testing as hybrid forms or subtle variations of the classical concepts, but those are not important as far as the educational purpose of this chapter is concerned.

- In *white*-box testing of a software module, the tester is allowed to see its program code which is then –at least in modern test suites– also highlighted in different colours, etc., as the run of a program under test is proceeding step by step.
- In *black*-box testing, this kind of observation is either not allowed or not possible. The *observables* in black-box testing are input-output pairs $\mathbf{p} \in \mathbf{R}$, whereby $\mathbf{R}$ is the a-priori unknown input-output relation of the module $\mathbf{M}$ under test, a relation induced by the *operational semantics* of that module: $\mathbf{R} = \{ (\mathbf{i}, \mathbf{o}) \mid \mathbf{i} \in \mathbf{I}(\mathrm{M}), \mathbf{o} \in \mathbf{O}(\mathrm{M}), \mathbf{o} = \mathbf{M}(\mathbf{i}) \}$. Note that the elements $\mathbf{i} \in \mathbf{I}(\mathrm{M})$ are not required to be finite (or statically limited) objects –they could be data *streams* that are continuously being fed into a running reactive system– which implies that it is also *not* required that $\mathbf{o} = \mathbf{M}(\mathbf{i})$ represents a terminating computation. In case the computation represented by $\mathbf{o} = \mathbf{M}(\mathbf{i})$ is *non*-terminating, $\mathbf{o}$ is ejected as *side-effect* of M (not as its explicit functional return-value), which implies that it is also possible for $\mathbf{o}$ to be an infinite *stream* (instead of a finite string).

Passive testing is black-box testing. Consequently, the test engineer must carefully prepare a module, which shall be passively tested, in such a way that it will emit a useful string or stream of test data (while running), a string or stream of data which can then be *interpreted* as meaningful *information* about the otherwise invisible operations happening within the confines of the black box. The test engineer should solve this problem *wisely*, such that:

- the test data are *not* emitted *too sparsely* (otherwise too little information can be deduced about the invisible activities in the black box),
- the test data are *not* emitted *too densely* (otherwise the external activity of observation and interpretation becomes too difficult), and
- the *auxiliary* additional program code, which is responsible for ejecting the test data for observation during passive testing, should
    - *not interfere semantically* with the module's main functional code (otherwise the module's specified operational semantics would be violated), and
    - *not slow down* the execution speed of the module under test to an unacceptably low value (which is especially important in safety-critical real-time applications of such a software module).

The *interpretation* of the emitted test data as 'information' about the running module under test is then a matter of the *test model* which completes this scenario of model-based passive software testing – the same is basically true for model-based passive hardware testing (see subsequent sections of this chapter), with the only difference that the emission of observable test data cannot be implemented by program code in those cases. The *quality* of this test method (and its results) then depends crucially on *two* items, namely:

- the quality (richness, informativeness) of the emitted observable *test data*, and
- the quality (subtlety, distinctiveness) of the *test model* on which the interpretation of the test data depends.

In other words: the richest test data are useless if the test model is too 'dumb' to distinguish them, and the finest test model is useless if the emitted test data are too coarse or too sparse for interpretation.[2] Thus, the construction of a suitable and useful test model as an abstraction of the system's specification model (see Figure 1 again) can be regarded as a sub project (of the entire development and testing project) in its own right, which also entails some careful *requirements* elicitation *within* the sub

[2] There is thus an interesting correspondence between this method of passive testing and the epistemology by the philosopher *Immanuel Kant*, from whom we remember the well-known aphorism: "*Gedanken ohne Inhalt sind leer, Anschauungen ohne Begriffe sind blind*" – thoughts without objects/contents are empty/void, intuitions/observations without theoretical ideas/concepts are blind (Kritik der reinen Vernunft: 1781).

project of test model construction. A recent example of a test requirements elicitation process in support of model-based testing can be found in (Santos-Neto et al. 2007).

Before we get to the discussion of some further literature on model-based passive software testing, let us briefly illustrate these principles, as we have outlined them above, by a very small and simple *example:*

Let **P** be a process defined in the well-known notation of *CSP* as follows:

$$\mathbf{P} = (\mathrm{Q} \sqcap \mathrm{R}), \text{ whereby}$$
$$\mathrm{Q} = (\boldsymbol{a} \rightarrow \mathbf{P}), \text{ and}$$
$$\mathrm{R} = (\boldsymbol{b} \rightarrow \mathbf{P}).$$

Obviously this is a non-terminating process in which the 'test data' that our 'software module' emits are arbitrary long sequences of ***a*** and ***b*** in any order. As a valid 'test model' to this trivial 'software module' we could then use the equally trivial regular expression

$$\mathbf{E} = \{\boldsymbol{a} \mid \boldsymbol{b}\}^*$$

which is sufficient to 'recognize' (identify) all *traces* that can be emitted correctly by process **P**. Should, for whatever reason, our process emit any different symbol, then **E** would not be able to match it any more. Consequently our 'software module' would have *failed* this test.

Alternatively we could also define an explicit, *positive acceptor* model, for example by means of the regular expression.

$$\mathbf{A} = \{\boldsymbol{abba}\}^*$$

What we have called 'acceptor' model here is thus also a test model, however from a different *pragmatic* perspective; depending on whether we want to passive-test 'positively' for the presence or 'negatively' for the absence of some desired or un-desired phenomenon. In this example, we would let our 'software module' **P** run so long until we observe an emission of a trace t that can be interpreted as a word in the language of **A**, thus $t \in L(\mathbf{A})$, in which case we would declare our module as 'acceptable' (and possibly stop the process of passive testing at that point). Finally note, at the end of this little example, that $L(\mathbf{A}) \leq L(\mathbf{E})$, which is of course a logical matter of *model consistency* under the condition of model refinement.

After we have thus briefly clarified the most basic theoretical principles of model-based passive testing especially in the software domain we will now discuss somewhat more realistically

- how the technique of *aspect-oriented programming* (AOP) can be used in practice to insert trace-emitting test-code into a software module in preparation for passive testing (Subotic et al. 2006), and
- how the *B*-method can be used (as one possibility out of many) for the specification of a test-model by means of which a string or stream of test-data can be interpreted (Howard et al. 2003).

In the remainder of this section we will thus discuss several *practical techniques* that can be used and applied in support of model-based passive testing in various scenarios. As it was mentioned already, a more comprehensive systematic taxonomy or classification scheme of methods and techniques for model-based passive testing does, as far as we know, not yet exist and is still a task for 'future work' in this field.

2.1 Aspect-Oriented Programming for Model-Based Passive Software Testing

This sub section, which is by-and-large a recapitulation of (Subotic et al. 2006), outlines how the programming techniques of AOP can be practically used in support of passive testing, especially for the emission of the observable runtime traces on which the techniques of passive testing is based. However, this sub section is *not* an introduction into aspect-oriented programming as such. Those readers who are not familiar with AOP are referred to the ample literature on AOP which has been published since the original papers by (Kiczales et al. 1997, 2001).

Aspects are the key concept in AOP. They can be regarded as 'pseudo objects' in the sense that they encapsulate internal states and behaviour which are relevant for a software system as a whole, and not only for some of its particular modules. Aspects are thus 'woven' into a software system, thereby building a kind of action-repository for the so-called *cross-cutting concerns* of the entire system (Subotic et al. 2006). In contrast to the so-called 'core concerns', which can be sufficiently implemented in functional separation by particular modules according to the software engineering principle of 'coupling and cohesion', cross-cutting concerns cannot be elegantly implemented by particular modules without AOP. Examples of those are: logging, tracing, profiling, security policy enforcement, transaction management, and the like (Subotic et al. 2006). Those cross-cutting concerns are typically *dynamic* in their character which means that they become especially important at system runtime.[3] The main idea of AOP is then that whenever some particular pre-conditions are fulfilled *anywhere* in the running software system, an aspect-module related to that pre-condition will be triggered 'on-the-fly', and some *additional* actions are carried out which would otherwise not have happened – for example, emitting some trace data which can then be used for the purpose of passive testing. In AOP terminology, the so-called 'point-cuts' define the above-mentioned preconditions, whereas the additional (in-woven) actions carried out around those point-cuts are also called the 'advice' of the corresponding aspect (Subotic et al. 2006). Figure 2 sketches the structure of an aspect definition in the *JAVA*-based aspect-oriented programming language *AspectJ*.

[INSERT FIGURE-2 HERE]

Figure 2: *Sketch of an aspect definition in the programming language AspectJ.*

Depending on whether aspects in AOP shall be used for the implementation of functional or non-functional system features, we can distinguish *development* aspects from *product* aspects (Subotic et al. 2006). The difference between these types is found in their *pragmatics*, not in their syntax or semantics:

- Development aspects are only 'switched on' during the phase of prototyping, typically in support of the programmer's comprehension of the system, especially for the purpose of *active* testing (white-box testing). Development

[3] There also exist *static* aspects (Subotic et al. 2006), but their discussion is not necessary in the scope of this chapter.

aspects are ‘switched off’ again after the prototype has been found satisfactory, and are thus *not* part of the delivered software product.

- Product aspects, on the other hand, are implementations of specified system requirements and can thus *not* be ‘switched off’ when the product system is finally installed and deployed (Subotic et al. 2006).

From this discussion it should be clear that passive testing with AOP can be done either on the basis of development aspects, or on the basis of product aspects. The decision between these two possibilities depends on a system’s *requirements* specification:

- If the requirements document stipulates that only a prototype shall be passively tested until satisfaction, then the trace-emitting aspects are clearly classified as development aspects.
- On the other hand, if the requirements document stipulates that also the delivered product shall be subject to continuous passive testing and observation while being in operation, then the trace-emitting aspects are clearly classified as product aspects which must *not* be switched off when the product system is deployed.

From a theoretical point of view, the execution of aspect code around point-cuts in an aspect oriented software system must be regarded as a *side-effect,* and the usual warnings about the practical risks of side-effects are applicable in this context as well. Therefore we recommend using aspects in passive testing *only* for the purpose of emitting interpretable trace-data, but *not* for any other system-relevant computations with effects on that system’s internal state. Moreover it is possible in AOP that two different aspects A and A’ can be triggered around the same point-cut P, in which case the *order of execution* of A and A’ is generally *not well-defined* (unless there are some conflict-resolving meta-rules in place). In the case of AOP-based passive testing, such a point-cut ambiguity could compromise the validity of the emitted test data trace, which could in consequence lead to a false interpretation of the correctness or defectiveness of the system under passive testing. For this reason we recommend ensuring that the trigger pre-conditions of any two active aspects are logically disjoint. However one can also use passive testing to *discover* the appearance of unintended orders of execution of overlapping point-cuts for systems under test which were programmed by AOP in the first place: if conflicting aspects A and A' at the same point-cut emit observable traces, then a passive testing unit can detect if their order of execution was not as intended. This holds, however, as mentioned before, only if the passive test unit does not itself introduce artefact errors on the basis of this AOP mechanism.

If we carefully navigate around those two pitfalls of AOP –namely: unintended side-effects, and overlapping trigger pre-conditions of different active aspects– we can clearly see several *advantages* of using AOP for the purpose of passive software testing:

- The emission of trace data can be elegantly implemented separately in aspect modules, thus keeping the system's core module 'clean' from any 'non-functional litter', and only *one* trace-aspect must be injected into the system instead of inserting trace-generating command lines into the program code of *all* core modules of the system.
- The trace-generating test aspects can be programmed by a group of programmers different from the group of programmers who construct the functional core modules of that system, in accordance with the well-known software engineering principle that development and testing should not be carried out by the same work group (a principle that aims at avoiding the psychological bias of good faith and belief in one's own work).
- The same core system can be differently traced from various viewpoints, simply by replacing one set of trace-aspects A by a different set of trace-aspects A', without any need for cumbersome and error-prone 'muddling' with the core modules (Subotic et al. 2006) which are implementing the main functionality of the system under the observation of passive testing.

Having thus described an elegant programming technique for the emission of trace data without any regard to their model-based interpretation, we can now turn our attention to another practical example paper in which the *B*-method was used for the *interpretation* of test traces emitted by a small web-service software system. Once again we should remark that the underlying principles of this technique have been known for about 20 years now; see, for example, the trace-viewer in (Helmbold et al. 1990), or the already mentioned approach of (Bochmann et al. 1989).

2.2 Model-Based Trace-Checking with B and SPIN

The idea and technique of *trace checking* was already known to (Jard et al. 1994). Subsequently this idea became more and further integrated into the emerging 'paradigm' of model-based or model-driven software engineering. In a paper from the year 2003 we can find one of the earlier approaches that systematically combined the issues of testing, 'tracing', executable formal models, and automated analysis into a methodological framework of model-based testing which was then called 'model-based trace checking' (Howard et al. 2003), or 'trace-driven model checking' in a related publication (Augusto et al. 2004). In this sub section of our chapter we shall briefly review the ideas of (Howard et al. 2003) as an interesting variant of the general principle that we have already explained above.

As usual in testing in general, the issue also of model-based testing is the detection of defects, not the proof of their absence. Also, according to the principle of modularity, different test models can be designed for the exploration of different aspects of system behaviour, as it was mentioned in the section on AOP of above. The key method of (Howard et al. 2003), which was a novel idea at its time of publication, was

- to implement the system by any suitable method of software development,
- to insert 'trace code' (possibly with help of AOP, as explained above) for capturing each significant system operation or event,
- to design one or several formal models of that system in a formal notation for which sufficient tool-support exists (a.k.a. 'executable formal methods'), and
- to run the generated traces through the available tools to check them against those models (Howard et al. 2003).

This procedure is also interesting in the sense that it inverts the steps of classical model-based testing: first the traces are produced from the system under test, and only then are they compared with the model to assess their validity. The web-based case study reported in (Howard et al. 2003), which was intended to corroborate the validity of that principle, is interesting for a number of reasons:

- There are several components *distributed* over the internet;
- There are several possible distributed *configurations* of the system, depending on the actual instantiations of the underlying JAVA code classes;
- Each of the participating components has its local version of the working-set of data and yet all versions must be kept consistent with each other;
- The system is implemented on top of a basis of complex middleware, including web servers, web-supported data bases, Java server pages (JSP), Java servlets, and related techniques (Howard et al. 2003).

Because of the distributed character of the studied system and the complexity of the middleware on which its implementation is based, the testing of such a system is obviously not a trivial task. Indeed such message-passing-based network systems tend to exhibit environmentally induced non-determinism and, therefore, non-repeatability in classical (active) tests. For this reason the above-mentioned example system can be regarded as a typical case for the application of self-monitoring in the form of trace-based testing. In that case study (Howard et al. 2003), trace-generating program code was not injected by means of AOP, but rather in the form of Java 'Beans', because of the web-based character of the distributed system in that case study. At each point where trace information should be captured, two different Java Beans were invoked: one to collect a current state of the system at the point of activation, and another one to access a web-based data base system (SQL DBMS via JDBC) for storing a 'word' of the trace (Howard et al. 2003). The traces in that case study were thus not written into a simple text file, but rather into a purposefully designed *relational data base*, for a number of reasons:

- It enables collecting different sub sets of trace information according to different needs and interests by different members of the group of test engineers.
- It becomes easier to *parse* and *re-format* the trace entries according to the syntactic input requirements of the checker tools to be used after tracing.
- It becomes possible, thanks to a web-based trace database, to collect trace information from source components that can be located anywhere in the world-wide web (Howard et al. 2003).

The problem of *where* in the system's program code to activate trace generation was also discussed in (Howard et al. 2003). Obviously one should record information about a running system whenever there is 'news', that is, *whenever the system under test changes its internal state* in an interesting and relevant way. The answer to the question: "what is interesting and relevant?" must obviously be given with reference to the test model on which the entire method is based. As reported in (Howard et al. 2003), the test model can –and should– also be used to discover meta-errors, or false-error artifacts, a phenomenon that can occur when the trace-recording itself is not correctly implemented. The discovery of such false-error artifacts can, however, not be based on testing again (Howard et al. 2003). Such discovery would require 'rational' techniques, such as code-inspection and the careful comparison of the system model with its corresponding implementation, as it was also emphasised again more recently in (Rombach et al. 2008).

'B', a model-oriented formal notation (Abrial 1996), was used in (Howard et al. 2003) for the definition, construction, and simulation of valid traces. In B, a system is modelled as an abstract state machine with *operations* and *invariants*. B specifications can be written non-deterministically on a high level of abstraction, but can be refined to more specific deterministic specifications. In such a deterministic form, B models can be made executable and thus serve the purpose of rapid prototyping, or –as in the case of (Howard et al. 2003)– the generation and analysis of runtime traces.

Indeed it is interesting to note that testing by trace-checking in (Howard et al. 2003) was done only in an *in*direct, not in a direct fashion. The checked traces were in fact not the traces of the web-based system that was mentioned above, which we could call in hindsight the 'primary' traces. Those primary traces were in fact used to 'steer' (control) the executable B-machine which was designed as a *model of* the web-based software system. That B-machine then generated a trace which we could now call 'secondary' trace, and only that secondary trace was then checked for well-formed-ness as described in the methodology sections of above. The 'oracle machine' for passive testing in this case study is thus a module that implements an executable B-machine (Howard et al. 2003). Needless to say, this *in*direct way of passive testing requires a semantic *correspondence* of the primary software system with its secondary prototype –here the executable B-Machine– which was then passively tested on behalf of the primary software system (Howard et al. 2003). One might perhaps call this technique now, in hindsight, ‘passive testing of second degree’. This structural correspondence between the primary software system and its secondary B-representation was organized in such a way that the corresponding B-machine under test made a state-transition whenever one of the trace ‘beans’ in the primary software system was invoked (Howard et al. 2003).

Under these circumstances of indirect passive testing of second degree, whereby a protoype is passively tested on behalf of its corresponding software system of first degree (which is justified by some kind of pre-established structural homomorphism between the primary system and its secondary prototype), the method of (Howard et al. 2003) worked as follows:

- Execute a test run of the primary software system to record a primary trace into the trace database;
- Select a (possibly ‘filtered’) trace of ‘interest’ from the database;
- Use this primary trace as input for the B-animation tool and model checker *ProB* (Leuschel et al. 2003), whereby a secondary trace suitable for processing with these tools is generated;
- Let the animator re-run (simulate) the run of the secondary trace in terms of the available state transition operations of the executable B-machine model;
- Regard the model-based test as
    - *passed* if the B-based re-run of the secondary trace was possible;
    - otherwise regard the model-based test as *failed* if no sequence of B-machine operations could be found to reproduce (simulate) the sequence of activities of the trace-writing Java Beans in the primary software system (Howard et al. 2003).

As mentioned above, there must be a *well-defined correspondence* between the primary trace (in the trace database, from the system under test) and the secondary B-machine-trace that ‘emulates’ the primary trace and that is checked ‘on behalf’ of the primary trace. Without such a correspondence between the primary and the secondary trace, which resembles the *semantics-preserving* relationship between *concrete* and *abstract* syntax in the domain of compilers, the exercise of checking the secondary (abstract) trace on behalf of the primary (concrete) trace would be futile.

In another branch of the research project in the wider context of (Howard et al. 2003), the same trace data from the trace database were extracted (via SQL) to be transformed into a suitable representation for the *Promela*-based model checker *SPIN* (Holzmann 1997). According to the well-known principles of model checking (Clarke et al. 1986, 1999), the *invariants* of a system under test can be expressed in terms of *temporal logic*, which also implies that a also temporal logic formula can play the role of a 'test model' for such a system. As reported in (Howard et al. 2003), it is possible to re-write a runtime trace of the system under test in terms of the language Promela and use this as an input to SPIN to check the trace against a given temporal logic formula *T*, to find out whether or not the trace is a legal 'inhabitant' of the state-'world' specified by that formula *T*. In (Howard et al. 2003) those formulae were provided manually, 'ab initio', and were thus not directly derived from the system's specification. Vice versa, in an inductive way, it was also possible to use those traces, which were already known to be correct, as the concrete basis for the construction of another temporal logic formula *T'* which could then be used as the abstract 'test model' for further experiments in the passive testing of the original system (Howard et al. 2003). Compared with other well-known methods of runtime-testing of programs, such as, for example, executable assertions, the indirect method of (Howard et al. 2003) has a number of advantages: amongst others,

- it is possible to re-check already existing traces (in the trace database) against other and further system properties which had not yet been thought of at the time at which those traces were captured, and
- one and the same trace (in the trace database) can be re-parsed to be checked against a variety of aspects, via a variety of test models, written in a variety of formal notations, supported by a variety of checker tools (Howard et al. 2003).

2.3 Passive Testing with Extended Finite State Machines

Finite state machines, though not specified in *B*, are also the basis of the passive testing method by (Cavalli et al. 2003). Similar to the approach of (Howard et al. 2003), the approach of (Cavalli et al. 2003) also deals with *invariants*, expressed in logic, against which the emitted traces are checked, whereby the possible non-determinism of the underlying finite state machines is an additional source of technical difficulty. The key distinction of (Cavalli et al. 2003) is the one between the analysis of *control flow* and the analysis of *data flow*, a distinction that is also well-known in compiler technology (Aho et al. 2007). Whereas control flow can be passively tested rather easily on the basis of ordinary finite state machines (FSM), data flow is considerably more difficult to test – both in active testing, as well as in passive testing (Cavalli et al. 2003). For this purpose of passively testing data flow properties, the approach of (Cavalli et al. 2003) works with *extended* finite state machines (EFSM) instead of ordinary FSM, whereby the main difference between EFSM and FSM is that the state transitions of an EFSM can have additional *side-effects* on the values of a *configuration of variables* (Cavalli et al. 2003) in a similar way in which the state transitions of a stack automaton have side effects on the contents of the stack storage attached to such an automaton (Aho et al. 2007). As in (Howard et al. 2003), also (Cavalli et al. 2003) considered the possibility of *non-*determinism in the operations of the model machines, and provide techniques to cope with this difficulty when testing (checking) the EFSM traces against the logic-specified model invariants. A similar approach to model-based system self-checking on the basis of finite state automata (FSA) models can be found in (Paradkar 2003).

3. Network Security by Model-Based Passive Testing

The global computer network is growing day by day, and so does the amount of information and communication that is sent as 'traffic' through the net. Accordingly, our vulnerability to malicious misuse of these possibilities has also increased. So-called *firewalls* (also known as 'intrusion detection and prevention systems') are devices that are designed to protect individual computers as well as larger sub nets from un-authorized external intrusion. They usually consider the stream of network traffic at one of two levels, by:

i. examining the *contents* of the network stream, and/or
ii. recording (capturing) a stream of metadata about the communication activities at that security-point and saving those traces in *log files* in support of an a-posteriori analysis; for comparison see (Andrews 1998).

These two levels represent the extremes: in the former case, the firewall considers the minute details of the actual network traffic, while in the latter case, much of the detail of the network traffice is projected away, leaving only the 'shadow' in the log files. Naturally, solutions in-between are also possible.

Intrusion detection and prevention is conceptually similar to the scenario of passive testing. The difference is mainly a pragmatic one: Whereas in passive testing we want to find out whether or not a device under test is defective, in intrusion detection we want to decide whether or not a communicator 'knocking at the door' of a firewall is doing so in a malicious way. We can thus even imagine that a firewall may also be used as a passive testing device for 'normal' software, although there is not yet evidence of this application concept in practice. From this description of the scenario it is obvious that there are two basic possibilities of applying the methods of passive testing to the problem of firewall analysis:

- The firewall could analyze the stream incrementally ('online') as it flows.
- The stream (the actual network contents in a 'content-based firewall' or the metadata log files) could be analyzed as one batch ('offline').

Compiler technology (Aho et al. 2007) could then be used for the syntactic analysis (parsing) of the contents of those data 'sentences' produced by a firewall. The test model in such a setting is then a formal *grammar* against which those firewall sentences must be *parsed*.[4] This could be done in a *positive* as well as in a *negative* mode:

- In the positive mode ('green' mode), a formal grammar specifies the syntactic properties of 'harmless' sentences, and the test yields the result 'acceptable communication' if the sentence can be successfully parsed against that grammar.
- In the negative mode ('red' mode), a formal grammar specifies the syntactic properties that would result from an already well-known pattern *X* of attempted intrusion. The test would then yield the result 'intrusion attempt of type *X* detected' if a new log can be successfully parsed against that pattern-specific grammar.

[4] A student in our research group has recently started his final-year-project on this topic.

The fundamentals of network security are covered in detail in (Cheswick et al. 2003), whereas (Varghese 2004) provides a detailed treatment of the algorithmics involved.

3.1 Contents-Based Analysis

Just as log file analysis suffers from 'information loss' as the stream's contents are projected away, *contents-based* analysis can suffer from extreme information overload with several causes:

- Due to the layered approach to networking software, the structured inter-process communication over the network (of the applications using the network) appears as completely 'flattened', i.e., devoid of internal structure.
- An important strength of modern networking is 'packetization', meaning that the network streams (of various applications) are split into small (often fixed sized) chunks that are delivered interspersed, possibly out-of-order, or even unreliably.
- Modern networks are easily capable of very high throughput up to 10Gbps (roughly 1GB/second).

Currently, all analysis 'models' for intrusion detection are expressed as *regular expressions.* The set of regular expressions (a.k.a. *signatures*) usually number in the thousands, and are crafted for the 'red' mode of operation: they characterize unallowable patterns in the network traffic. The first of the overload conditions above means that the regular expressions are also designed to conservatively capture the flattened network stream. The second complication (packetization, etc.) is usually solved with a significant amount of bookkeeping so that the passive testing of the individual network sessions (i.e., between different applications) are dealt with separately; furthermore, such an implementation is capable of reassembling and re-ordering the packets on-the-fly. The final complication (of performance) is often solved by brute-force: money spent on a full-network-speed firewall comes close to the money spent on the applications it is examining. The alternative (as of this writing ill-explored) is to 'sample' the stream and analyse only part of the stream, such as every second byte or only some of the network sessions. As with other imperfect passive testings models, some problems may slip through.

3.2 Log File Analysis

Research and development in this area is rather new and has not yet progressed very far. Two of the many difficulties that typically arise in this context are the following:

- Classically, parsing requires a *lexically finite* string entity as input, such that the parser can 'know' where to begin and where to stop "eating" that input string (Aho et al. 2007). In the potentially endless data streams generated by firewalls, however, those distinguished points are not easy to define, and probably require some additional heuristics for their declaration before the parser can start to parse.
- Secondly, the syntactic properties of firewall traces stemming from sophisticated attempts of intrusion might be so complex that they cannot be described by context-free grammars any more. Even if they could still be described in terms of context-*sensitive* grammars, then, however, we would have to face the notorious problem of *general undecidability of context-sensitive parsing* (Salomaa et al. 1997), which we can hope to solve only in particular special cases.

It is needless to emphasize that those two problems could occur in all application scenarios of grammar/parsing-based passive testing, not only the scenario of firewall analysis and network security. Nevertheless research into this direction is on its way, and there are a number of publications which the reader can refer to as starting points in this context; see for example (Brzezinski 2007) (Campanile et al. 2007) (Gudes et al. 2002) (Memon 2008).

4. Model-Based Passive Testing of Hardware

In this section we consider a number of hardware testing scenarios. In contrast to software, user-expectations of hardware are considerably higher, perhaps because of the tangible nature, making model-based testing crucial. Fortunately, most modern silicon chips are specified, designed, and built with a sequence of formal models, which are then usable in generating (by hand or automatically) the testing model.

4.1 Recent Trends in Hardware Development

Thanks to decades of ongoing research advances in chip production, chip densities (the number of transistors packed in a given area) have been doubling every 18-24 months,[5] in what is well-known as *Moore's Law.*[6] This doubling has brought a more-than-doubling of design complexity, as it was already described in the 1999 edition of the International Technology Roadmap for Semiconductors (Gargini 2000).[7] A little known fact (outside of the electronics industry) is that such doubling has made modern chips much more sensitive to several effects, most notably the following three.

- *Random quantum effects:* these arise randomly, allowing electrons (for example) to 'tunnel' from one place to another, even through insulating material. The probability of tunneling depends on the energy of the tunneler, the material through which it may tunnel, and the distance it may tunnel. This has proven useful in the design of some types of transistors, but has also lead to unpredictable behavior in very small circuits, where the small distances raise the probability significantly.
- *Structural quantum effects:* these are also random effects, whose probabilities rise because of structural properties of a chip design. For example, two parallel wires carrying data create a perfect environment to encourage coupling (electrical or quantum) which allows the data on one wire to cross-talk with the data on the other wire.
- *Timing and race-conditions:* these are timing related problems similar to the ones arising in parallel software systems. Two electrical signals may have nondeterministic arrival times at a gate, potentially yielding undefined or random behavior. In many cases, one of the signals is the clock and the race condition can be partially solved by slowing the circuit's clock rate. Although some race-conditions are design problems, more often they arise from manufacturing irregularities or thermal effects in which a wire-length or transistor size changes with temperature.

[5] This density-doubling is often confused with a doubling of clock-speeds, or a halving of prices. While clock speeds did, in fact, double for several years in keeping with Moore's law, recent issues with power consumption, quantum effects in transistors, etc. have made the clock speed plateau. From the economics standpoint, the prices have also not halved — rather, consumers want twice as much 'bang for the buck' in an electronics product.
[6] Gordon E. Moore was one of the co-founders of Intel.
[7] http://www.allbusiness.com/electronics/computer-equipment-computer/6150464-1.html

With all of these issues, model-based testing plays an important role. The interested reader can find much more about functional verification of chips in (Wile et al. 2005), while (Chiang et al. 2007) gives an excellent treatment of issues in modern nano-scale chips. Last but not least, (McFarland 2006) presented an overview of the entire design flow of a microprocessor. In the subsequent sub sections, we consider testing and verification of both clocked and unclocked circuits; this is interspersed with a discussion of redundant hardware in highly reliable systems — an area using techniques very similar to model-based verification.

4.2 Passive Testing and Verification of Clocked Circuits

At the time of this writing, the vast majority of digital chips are *clocked* circuits. Such circuits consist of several mutually dependent functional units (e.g., in a microprocessor this includes the register file, the adder, multiplier, memory management, instruction decoder, and so on). Their communication is carefully synchronized (this style of chip is thus called *synchronous*). Ideally, the synchronicity is driven by a chip-wide *clock* that is usually a square-wave electrical signal. Operations as well as data movements through gates only occur at a clock 'tick' (at the rising edge or at the falling edge, or on both). As an aside, two practical concessions are made for clocks:

i. The clock is never a square wave. Clocks that closely resemble a square wave contain many higher harmonics, which correspondingly require much more power. The power consumption of a chip is presently a leading restriction on chip density,[8] as higher-powered chips lead to more crosstalk and quantum tunneling, but also thermal failure (melting) of the chip.
ii. Some parts of the chip may use a local clock that is an integer multiple or divisor of the main clock. In this way, some elements can run faster or slower but still remain essentially synchronous.

The first point is responsible for some of the quantum effects mentioned previously. This second point is taken to extremes in subsequent sections when we consider unclocked chips and networks on chips. Careful design and layout are required, so

[8] The other leading issue is lithographic printing of chip features. At the time of writing, minimum feature sizes are less than one sixth of the wavelength of laser light used for lithography, leading to resolution and contrast issues, and therefore higher costs in lithography.

that the wire-lengths (propagating the clock signal and also data-signals) respect the desired clock rate, ensuring that data arrives neither too early or too late. Late or early arrivals lead to race-conditions, and borderline race conditions (e.g., as mentioned earlier, depending on the temperature of the chip expanding or contracting the wire lengths) can give nondeterministic behavior, which is inordinately difficult to debug.

Model-based passive testing has long been used in synchronous-chip design, though it is often so ad hoc as to be unrecognizable. Most chips are designed using a version of the following workflow:

i. *Functional specification:* The global functioning of the chip is specified in a high level language such as VHDL or Verilog;
ii. *High-level simulation:* The specification is directly executed to explore functionality and performance of the chip;
iii. *Compilation:* The VHDL or Verilog is compiled to produce an RTL (register-transfer language) version, at the level of flip-flops, latches, gates, and buses;
iv. *Layout and physical design:* The medium-level RTL specification is further compiled to individual transistors, which are then layed out in detail, respecting wire-lengths, individual gate-delays, etc., so that proper synchronous behavior is preserved;
v. *Electrical simulation:* to discover design faults before production;
vi. *Production:* i.e., the industrial manufacturing of the devises, and finally
vii. *Post-production testing:* to discover any faults after production of the chip.

Model-based testing is woven into three of the above steps (cf. the numbering above):

- In *step iii:* During compilation, tests on the level of sub components are automatically generated as a finite automaton. The automaton makes transitions based on the signals appearing on the component's output pins. The automaton either blocks or transitions to a 'dead state' if the outputs are invalid according to the specification. The outputs (for any given clock cycle) of all of the pins are folded into a single automaton alphabet symbol, typically by devoting a single bit to each pin. Clearly, the automaton can be generated to be 'liberal' — it accepts more traces than just the allowable ones; this is especially useful when the component may have context-free behavior (e.g., if it has a large internal memory) that the finite automaton must approximate by containment[9]. An important, non-passive, extension of this consists of using a Mealy machine (an automaton not only with inputs but with outputs on transitions) to *drive* the component as well as monitor it (i.e., recognizer versus transducer).

- In *step iv:* In practice, using the monitoring automaton requires access to the 'pins' of the components. In most cases, the component is *floor-planned* somewhere in the chip's layout such that its inputs and outputs are not close to a physical pin (such pins are usually at the edge of the chip). This gives two possibilities for the passive tester: place it close to the component under test[10] (thereby taking up chip real-estate/area with the passive tester) or give external (off-chip) access to the component's inputs/outputs by routing them to physical pins. The latter option is most often implemented, as it places the monitoring automaton off-chip. Thereby the infrastructure that is routing signals to physical pins, known as a *scan ring*, allows for the contents of *latches* between components to be read out, usually over a bit-serial link.
- In *step vii:* Testing techniques using the scan ring require some external infrastructure: one standard for such a bit-stream is the JTAG interface, for which there are numerous tools to facilitate demultiplexing and passive testing of the bit-stream by running the monitoring automaton. Such tools are often part of the entire system, allowing a measure of self-testing in the field.

In this last step we may ask what would happen in the field if a problem is detected while the hardware is running. In the next sub section, we discuss such high-reliability systems that are self-healing. The general design issues for entire 'systems-on-chips' is detailed in (Jerraya et al. 2004). An excellent overview of modern chip verification is given in (Wile et al. 2005).

4.3 Passive Testing in High-Reliability and Redundant Hardware

As outlined in the previous sub section, lack of robustness plagues systems for a variety of reasons, primarily because of small feature sizes, manufacturing difficulties, etc. Safety- and financial-critical systems have been designed since the

[9] It is known from formal language theory, a regular language can be used to approximate a context-free language by containing it, or vice-versa.
[10] Also called the *device under test* (DUT) in the literature.

1960s for up-time, robustness, and recovery. As the costs of hardware dropped rapidly, it became feasible to double-implement critical hardware: two or more identical units are placed in the system shipped to customers. Such systems bring about some important choices:

- What are the 'units' that are duplicated? Depending on the application (of the system), designers' experiments, and cost factors, only some of the components (e.g., memory, register file, instruction decoder, I/O units, etc.) require duplication. This is referred to as the *sphere of replication*. Intuitively, one would imagine that a larger sphere of replication is necessarily better (providing perhaps total redundancy), however, a small sphere has the advantage of earlier detection of disagreement between the components, and therefore more opportunity for recovery as opposed to simply flagging a failure.
- What does identical mean? In the simplest case, 'identical' really means two systems designed and constructed identically. This case protects against random errors, but of course not against systematic or design errors. To protect against those possibilities, we require 'identical' to mean 'functionally or architecturally' identical — but *not* identically implemented. The latter entails completely different costs and is really only used in (some) life-critical systems such auto-pilots and medical equipment, though even those scenarios are beginning to use identical implementations. See (Leveson 1995) for more on such systems.
- How do we know if two units disagree? The outputs of the redundant units are compared to one another in a voting fashion. Obviously, with an even number of units (in the worst case two), we have a limited hope for knowing *which* is malfunctioning; even with an odd number of units, we have the (admittedly very unlikely) scenario where a majority of units fail identically and out-vote the good units. The comparators are placed at the border of the sphere of replication. For example, in a redundant register file, the comparator would be at the address lookups and the I/O bus to the register file(s) in which case we may be able to place the comparators off-chip and use the JTAG ports, thereby 'replicating' off-the-shelf hardware. In the simplest case, the comparators do bitwise real-time comparisons of the component. Comparators on a large

sphere or on diverse underlying implementations are much more complex, making it necessary to take timing differences as well as other issues into account. In such cases, the comparators' complexity approaches that of the system being passively tested. The comparators do *not* form the models in our model-based testing — it is simply a mechanism; the system is its own testing model.

- What happens when the units disagree? If a unit is malfunctioning, it is taken completely offline. Any hope for recovery and restart depends on there being more than two units duplicated, and also on the size of the sphere, as follows.
    - With a small sphere of replication, the amount of divergence before detection remains rather limited. For example, in a CPU, a redundant instruction decoder will immediately discover disagreement as typically one instruction is decoded every clock-cycle. In that case, the good unit(s) will pause and cross-load their internal state into the malfunctioning unit, most often via the JTAG ports or a similar scan-ring. From that point, execution continues again.
    - By contrast, a large sphere of replication allows the redundant units to diverge significantly before it is observable to the comparators at the border of the sphere. At that point, it is often too late to recover. Still, high up-time transaction systems (e.g., those running for credit-card processing) do implement such elaborate 'state reconstruction', bringing a failed unit back online and synchronized within a few seconds.

Many of these developments were pioneered and perfected in the 1980s by companies such as *IBM*, *HP,* and *Tandem*. An excellent technical treatment of such systems is given by (Mukherjee 2008).

4.4 Passive Testing of Asynchronous (Unclocked) Circuits

The foregoing sections have shown that clocks (in synchronous chips) are arguably the biggest 'evil' of modern chip design and implementation. In the long-term, asynchronous circuits (those without a clock) offer the best hope for correctness and high-performance. Despite sounding exotic, such circuits were already designed and in-use in the 1960s.

Ideally, asynchronous circuits are designed to be robust against arbitrary delays in any part of the circuit. In particular, correctness-by-construction is a highly desirable property, especially in the face of unknown or unbounded (but finite) delays in wires, gates, etc. The calculi to develop such circuits (known as *delay insensitive* circuits) is virtually identical to CSP, meaning that the same tools, trace-checkers, automated theorem provers, and model-based testers can be used directly. This makes delay insensitive circuits arguably the most robust type of digital circuit.

The most successful language (derivative of CSP) thus far is Tangram, primarily developed at Philips Research with input from the University of Eindhoven (NL), and now spun-off into *Handshake Solutions Inc*. A delay-insensitive circuit gives the following advantages:

- It can be physically floor-planned without regard to wire-lengths (between components), gate delays, etc., and it will still work when manufactured.
- Lithography processes (in chip production) give some natural variation in the geometry (length and width) of individual transistors, which in turn gives variation in the transistors' performance (current used, delay/transition-time, etc.). The timing aspects of such variations have no effect on the correctness of delay-insensitive circuits.

Model-based testing of such systems is particularly simple since delay-insensitive circuits are generated from *CSP*-like specifications. A *dual* specification (which is by definition a specification of the *environment*) is generated for each component to subject to model-based testing. The dual specification can, in turn, be used to generate/compile complementary testing components which are then integrated into the larger circuit for passive testing. The additional testing components are themselves delay-insensitive and may be floorplanned anywhere, including off-chip for cost-savings. Their outputs are most often fed to a *JTAG*-type interface for external monitoring.

Delay-insensitive circuits are but one of the possible forms of asynchronous circuit. Other possible design styles (each with their advantages) are, for example, such as to:

- accept arbitrary wire delays, but finite gate delays;
- arbitrary gate delays with finite wire delays;
- handshake circuits, which are partially clocked but exchange additional signals regarding the readiness of signals on the wires;
- a variant of handshake circuits known as bundled asynchronous circuits, in which the timing of data lines is better than the timing of control lines.

Despite their advantages, asynchronous circuits still only occupy a niche. The intrinsic problems of clocked circuits were highlighted already in the 1970s by Charles Molnar, and subsequently explored by Ivan Sutherland and Alain Martin at Caltech, Martin Rem at Eindhoven, Janusz (John) Brzozowski in Waterloo and Steve Furber at Manchester – see for example (Chaney et al. 1973). Over and above the interaction between research groups, there are a few notable products and spin-off companies: *Sun Microsystems* devoted considerable effort to (partially) asynchronous SPARC processor designs; *Philips* developed a variety of asynchronous chips and later spun-off *Handshake Solutions*, a provider of asynchronous design tools. *ARM* collaborated with the University of Manchester on an asynchronous ARM processor. Much of the asynchronous chips body-of-knowledge is captured in (Brzozowski et al. 1995).

4.5 Passive Testing of Networks-on-Chips

One of the latest alternatives to fully-synchronous chips are *networks on chips*. This design paradigm consists of two levels: individual components are designed using one of the well-understood design styles (most likely as synchronous/clocked circuits); the components are connected by a *network.*[11] This yields some specific advantages:

- The individual components can be quite limited in size, making them easier to design and test individually, including their own clock, clock distribution, and race condition avoidance. Indeed, the components may even be obtained from earlier (fully-clocked) versions of a chip.
- The communication between components is network-based and can therefore 'piggyback' on the infrastructure and know-how of distributed systems,

[11] The alternative connection techniques are most often bus-oriented or point-to-point.

especially those with communication between processes via message-passing over networks (Tanenbaum et al. 2007).

Very little is known at this time about model-based testing of such systems, and little automatic support for this is provided by the design-flow tools. There are, however, some promising directions for future research:

- All of the work on network intrusion detection and prevention systems (discussed earlier in this chapter) can be leveraged, since the on-chip networks are actually Ethernet-like networks.
- The work on scan rings (e.g., JTAG) can be extended to reading out model-checkable state directly on the on-chip network, providing it to an external network port for analysis.
- Several systems have common underpinnings: networks-on-chips, asynchronous circuits, and *CSP* programs. As a result, a promising research direction is to extend the asynchronous and *CSP* tools (e.g., *Tangram* and *Occam*) with the aim of supporting (and compiling for) networks-on-chips.
- Model-based testing of networks (for collisions, saturation, etc.) is well understood and can be implemented at this level as well. For more on this, see (Varghese 2004).

A good introduction and reference to the field of *networks-on-chips* is given in (Micheli et al. 2006). Model-based design for *embedded systems* is discussed in (Nicolescu et al. 2009), whereby the design models for such systems can be used as the foundations of their test models, as it was sketched above in Figure 1.

5. Extended Example (Exercise for Students)

Not shown in this pre-print paper

6. Theoretical Limits of the Passive Testing Method

Every *method* has its limits, and every *methodology* that does not attempt to reflect on the theoretical limits of the methods under its umbrella is not a well-elaborated methodology. Therefore, to ‘round out’ the content of this chapter, we briefly allude to some theoretical limits of passive testing by means of the following small example.

A variant of the well-known *Collatz algorithm* (after the mathematician Lothar Collatz, 1910 – 1990), can be defined in pseudo code as follows:

```
INPUT natural number N > 0.
WHILE (N > 1)
      {IF (N even number)
           THEN { N := N/2 }
           ELSE { N := (3 * N) + 1 } }
HALT.
```

If Collatz's famous conjecture (1937, still un-proven to date) is true, then this algorithm will eventually terminate (halt) for *any* (every) given input from the natural numbers.

Let us now try to apply our method of passive testing to Collatz's algorithm, and inject trace-emitting instructions at its crucial points, as follows:

```
INPUT natural number N > 0.
WHILE (N > 1)
     {IF (N even number)
          THEN { N := N/2 ; PRINT("down") }
          ELSE { N := (3 * N) + 1 } ; PRINT("up") }
HALT.
```

Under the assumption that Collatz's conjecture is true it is intuitively evident that this trace-writing variant of the Collatz algorithm implements a function *C* that maps each input number *n* to a finite sequence of ups and downs, thus:

$$C\colon \mathrm{N} \rightarrow \{\mathrm{up} \mid \mathrm{down}\}^*$$

We even know with certainty that the last token of every acceptable Collatz-trace must be a 'down', namely when the algorithms finally executes the operation 2/2=1 by means of which the loop condition (N > 1) is broken, thus:

$$C\text{: N} \rightarrow \{\text{up} \mid \text{down}\}^*\text{down}$$

We also know that there exist words in the language L = {up | down}* that do *not* correspond to any possible input number n that could be fed into the Collatz algorithm. Take for example this very short *false* trace 'up down'. The final action, 'down', indicates the operation 2/2=1, which means that N = 2 *before* that action. The first action, 'up', however, indicates that some action (3*X) + 1 = N must have happened. With N=2 this implies that X = 1/3, which is not a natural number and thus not allowed in the Collatz domain. In other words, we also know that our characteristic function *C* cannot be surjective, and every Collatz 'module' that would emit the trace 'up down' would surely be defective and fail the test.

At this point, the reader should have noticed that in this example we had indeed intended –however hopelessly– to use the function *C* as our *test model* by means of which we wanted to passively test and decide whether or not our initial Collatz 'module' was defective.

This problem, however, is known to be *computationally irreducible* – in other words: an *accurate* test model to our Collatz 'module' *cannot be any simpler or less complex than the module itself* which we wanted to observe and test in the first place. Thus, we would have to apply the Collatz algorithm itself again, backwards, to find out if we could unwind an observed Collatz-trace step by step until we might arrive at a valid input number which would then allow us to decide to 'accept' the emitted trace. The existence of a *simple and accurate* test model, by means of which we could *correctly and easily* distinguish the 'good' from the 'bad' up-down-traces emitted by our Collatz 'module', would imply nothing less than the discovery of an *analytic* correctness proof for Collatz' yet unproven conjecture from the 1930s.

We can thus conclude methodologically that the applicability of model-based passive testing strongly depends on our theoretical ability to discover and construct a suitable test model, and we know at least one counter-example of a computational system –see above– for which such a test model may possibly not even exist.

7. Summary of this Chapter

We conclude our discussion of problems and possible solutions in the area of model-based passive testing of safety-critical components with the following points.

- *Model-based testing* (MBT) and *passive testing* are closely related to each other, whereby passive testing is usually a special case of MBT. To bring the process of passive testing in operation, formal test models are necessary to specify the required monitor units (see Figure 1).
- As far as its *application domain* is concerned, passive testing is better applied to *hardware* components and *distributed systems* rather than software modules for classical mono-processor platforms. Our educative *example*, a *cellular automaton* (systolic array – Section 5), was relevant in this regard, because systolic arrays are instances of distributed systems which can also be implemented in hardware as FPGA.
- Nevertheless, model-based passive testing is applicable to *software* modules as well, whereby in this case we can elegantly combine passive testing with *aspect-oriented programming* (AOP). In this paradigm of programming, the trace-generating auxiliary code can easily be 'woven' into the functional modules of a software system under test, as it is, for example, explained in (Subotic et al. 2006) or other literature on AOP.[12]
- In any case, the key idea of passive testing is to let a module (software or hardware) under observation emit a *runtime trace* of *data* which can be *interpreted* as *information* of the observable operating (running) module about itself. The interpretation of those trace data as test "information" is, of course, based on the definitions of *test model* at the root of this technique. Many different formal techniques and notations can be used to define such test models, such as regular expressions, formal grammars, *B*, *CSP*, and the like. The B-notation, for example, was used for the definition of acceptable runtime traces in (Howard et al. 2003).
- By means of such kinds of model-based trace interpretations, the actually emitted runtime traces (from the modules under observation in model-based

[12] Some readers might perhaps have appreciated a more detailed example on trace generation in AOP, but such implementational details would have deviated too far from the purpose and central theme of this chapter, namely the conceptual relationship between abstract trace models and concrete system properties.

passive testing) are then *classified* as *acceptable* or *non*-acceptable. From a non-acceptable runtime trace we can conclude that the system under observation did not operate correctly, such that an alarm signal can then be sent. From an acceptable runtime trace we may, strictly speaking, not conclude anything, since testing is –in principle– *not* a suitable method for *proving* the absence of defects with mathematical certainty.

- Model-based passive testing is especially recommendable as an *additional guard* for *safety-critical* systems. Model based *intrusion detection* by log file analysis of *firewalls* is only one special instance of model-based passive testing in a safety-critical environment.
- A comprehensive theory, or classification, or taxonomy of the various techniques, which can be used to implement model-based passive testing in various software- and/or hardware-based application domains, does –as far as we know– not yet exist and must still be regarded as 'future work' for research in this field.

Acknowledgments: (Not shown in this pre-print paper)

References

Abrial, J. (1996): *The B Book – Assigning Programs to Meanings*. Cambridge University Press.

Aho, A.V. & Lam, M.S. & Sethi, R. & Ullman, J.D. (2007): *Compilers – Principles, Techniques and Tools*. 2nd ed., Pearson / Addison-Wesley.

Amman, P. & Offut, J. (2008): *Introduction to Software Testing*. Cambridge University Press.

Andrews, J. (1998): *Testing using Log File Analysis – Tools, Methods, and Issues*. Proceedings 13th International Conference on Automated Software Engineering, pp. 157-167, IEEE Computer Society Press.

Augusto, J.C. & Howard, Y. & Gravell, A. & Ferreira, C. & Gruner, S. & Leuschel, M. (2004): *Model-Based Approaches for Validating Business-Critical Systems*. STEP'04 Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice. IEEE Computer Society Press.

Baker, P. & Dai, Z.R. & Grabowski, J. & Haugen, Ø. & Schieferdecker, I. & Williams, C. (2008): *Model-Driven Testing using the UML Testing Profile*. Springer-Verlag.

Bochmann G. v. & Dssouli, R. & Zhao, J.R. (1989): *Trace Analysis for Conformance and Arbitration Testing*. IEEE Transactions on Software Engineering, Vol. 15, No. 11, pp. 1347-1356.

Broy, M. & Jonsson, B. & Katoen, J.-P. & Leucker, M. & Pretschner, A. (eds.) (2005): *Model-Based Testing of Reactive Systems*. Lecture Notes in Computer Science, Vol. 3472 (sub-series: Advanced Lectures), Springer-Verlag.

Brzezinski, K.M. (2007): *Intrusion Detection as Passive Testing – Linguistic Support with TTCN-3*. Lecture Notes in Computer Science, Vol. 4579, pp. 79-88, Springer-Verlag.

Brzozowski, J.A. & Seger, C.-J. (1995): *Asynchronous Circuits*. Monographs in Computer Science, Springer-Verlag.

Campanile, F. & Cilardo, A. & Coppolino, L. & Romano, L. (2007): *Adaptable Parsing of Real-Time Data Streams*. Proceedings 15th Euromicro Conference on Parallel, Distributed and Network-based Processing, pp. 412-418.

Cavalli, A. & Gervy, C. & Prokopenko, S. (2003): *New Approaches for Passive Testing using and Extended Finite State Machine Specification*. Information and Software Technology, Vol. 45, pp. 837-852, Elsevier.

Chaney, T.J. & Molnar, C.E. (1973): *Anomalous Behavior of Synchronizer and Arbiter Circuits*. IEEE Transactions on Computers, Vol. C-22, No. 4, pp. 421-422.

Cheswick, W.R. & Bellovin, S.M. & Rubin, A.D. (2003): *Firewalls and Internet Security – Repelling the Wily Hacker.* 2nd ed., Addison-Wesley.

Chiang, C. & Kawa, J. (2007): *Design for Manufacturability and Yield for Nano-Scale CMOS*. Springer-Verlag.

Clarke, E.M. & Emerson, E.A. & Sistla, A.P. (1986): *Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications.* ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, pp. 244-263, ACM Press.

Clarke, E.M. & Grumberg, O. & Peled, D. (1999): *Model Checking*. MIT Press.

Dick, J. & Faivre, A. (1993): *Automating the Generation and Sequencing of Test Cases from Model-Based Specifications*. Proceedings FM'93 Conference on Industrial-Strength Formal Methods, pp. 268-284.

Gargini, P. (2000): *The International Technology for Semiconductors (ITRS) – Past, Present, and Future*. Proceedings GaAs IC: 22nd Annual Gallium Arsenide Integrated Circuit Symposium, pp. 3-5, IEEE Press.

Gudes, E. & Olivier, M. (2002): *Wrappers – A Mechanism to support State-based Authorization in Web Applications*, in Chen, P.P. (ed.), Data and Knowledge Engineering, pp. 281-292, Elsevier.

Helmbold, D.P. & McDowell, C.E. & Wang, J.Z. (1990): *Trace Viewer – A Graphical Browser for Trace Analysis*. Technical Report UCSC-CRL-90-59, University of California at Santa Cruz, John Baskin School of Engineering.

Holzmann, G. (1997): *The SPIN Model Checker*. IEEE Transactions on Software Engineering, Vol. 23, No. 5, pp. 279-295, IEEE Press.

Howard, Y. & Gruner, S. & Gravell, A.M. & Ferreira, C. & Augusto, J.C. (2003): *Model-Based Trace-Checking*. Proceedings of the SoftTest UK Testing Research II Workshop on Software Testing, Technical Report, University of York.

Jard, C. & Jeron, T. & Jourdan, G.V. & J.X. Rampon, J.X. (1994): *A General Approach to Trace Checking in Distributed Computing Systems*. Proceedings 14th International Conference on Distributed Systems, pp. 396-403, IEEE Computer Society Press.

Jerraya, A. & Wolf, W. (2004): *Microprocessor Systems-on-Chips*. Morgan Kaufmann.

Kiczales, G. & Lamping, J. & Menhdhekar, A. & Maeda, C. & Lopes, C. & Loingtier, J.M. & Irwin, J. (1997): *Aspect-Oriented Programming*. ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming.

Kiczales, G. & Hilsdale, E. & Hugunin, J. & Kersten, M. & Palm, J. & Griswold, W. (2001): *Getting started with AspectJ*. Communications of the ACM, Vol. 44, No. 10, pp. 59-65, ACM Press.

Kung, H.T. (1982): *Why Systolic Architectures?* Computer, Vol. 15, No. 1, pp. 37-46, IEEE Computer Society Press.

Leuschel, M. & Butler, M. (2003): *ProB – A Model Checker for B*. Lecture Notes in Computer Science, Vol. 2805, pp. 855-874, Springer-Verlag.

Leveson, N. (1995): *Safeware – System Safety and Computers*. Addison-Wesley.

Memon, A.U. (2008): *Log File Categorization and Anomaly Analysis using Grammar Inference*. M.Sc. Thesis, Queens University of Canada.

McFarland, G. (2006): *Microprocessor Design*. McGraw-Hill Professional.

Micheli, G. de & Benini, L. (2006): *Networks on Chips – Technology and Tools*. Morgan Kaufmann.

Mukherjee, S. (2008): *Architecture Design for Soft Errors*. Morgan Kaufmann.

Nicolescu, G. & Mosterman, P.J. (2009): *Model-Based Design for Embedded Systems*. CRC Press.

Paradkar, A. (2003): *Towards Model-Based Generation of Self-Priming and Self-Checking Conformance Tests for Interactive Systems*. SAC'03 Proceedings of the Annual ACM Symposium on Applied Computing, pp. 1110-1117, ACM Press.

Petkov, N. (1992): *Systolic Parallel Processing*. Advances in Parallel Computing, Vol. 5, North Holland Publ.

Richardson, D.J. & O'Maley, O. & Tittle, C. (1989): *Approaches to Specification-Based Testing*. TAV3 Proceedings of the 3rd ACM SIGSOFT Symposium on Testing, Analysis and Verification, pp. 86-96, ACM Press.

Rombach, D. & Seelisch, F. (2008): *Formalisms in Software Engineering – Myths versus Empirical Facts*. Lecture Notes in Computer Science, Vol. 5082, pp. 13-25, Springer-Verlag.

Salomaa, A. & Rozenberg, G. (1997): *Handbook of Formal Languages*. Springer-Verlag.

Santos-Neto, P. & Resende, R. & Pádua, C. (2007): *Requirements for Information Systems Model-based Testing*. SAC'07 Proceedings of the Annual ACM Symposium on Applied Computing, pp. 1409-1415, ACM Press.

Shukla, S.K. & Talpin, J.-P. (2010): *Synthesis of Embedded Software – Frameworks and Methodologies for Correctness by Construction*. Springer-Verlag.

Subotic, S. & Bishop, J. & Gruner, S. (2006): *Aspect-Oriented Programming for a Distributed Framework*. South African Computer Journal, Vol. 37, pp. 81-89, Sabinet Publ.

Tanenbaum, A.S. & Steen, M. van (2007): *Distributed Systems – Principles and Paradigms*. 2nd ed., Pearson / Prentice Hall.

Varghese, G. (2004): *Network Algorithmics – An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann.

Wile, B. & Goss, J. & Roessner, W. (2005): *Comprehensive Functional Verification – The Complete Industry Cycle*. Morgan Kaufmann.

Zhirnov, V. & Cavin, R. & Leeming, G. & Galatsis, K. (2008): *An Assessment of Integrated Digital Cellular Automata Architectures*. Computer, Vol. 41, No. 1, pp. 38-44, IEEE Computer Society Press.