Formal Methods and Conformance Testing — or — What are we testing anyway?

Stefan Heymer Jens Grabowski

University of Lübeck, Institute for Telematics
Ratzeburger Allee 160, D–23538 Lübeck, Germany
eMail: {heymer, grabowsk}@itm.mu-luebeck.de
WWW: http://www.itm.mu-luebeck.de

Abstract

In this paper, we will show the correlation between the notion of implementation relations known from formal methods and ideas of conformance testing. We will show that the implementation relations realized through the practical testing of systems come from a family of parameterized implementation relations. We will also show that for glass box testing, implementation relations parameterized by test purposes converge to the may–testing preorder of DeNicola and Hennessy [7], while for black box testing, implementation relations parameterized by test cases converge to a may–testing preorder of the behavior visible at the interface to the environment.

1 Motivation

When developing concurrent systems with formal methods, the notion of correctness of an implementation with respect to a specification plays a major role. Many of such implementation relations can be found in literature, e. g., bisimulation equivalence [6], failure equivalence and preorder [4], testing equivalence and preorder [7], as well as many others [11].

To ensure properties of implementations, given implementation relations between a specification and its implementation have to be checked. For implementations for which a formal model exists this can be done by tools like the Concurrency Workbench [2]. Yet, most implementations, especially those in the field of telecommunication systems, are black boxes — only the behavior visible to the environment can be modeled. Hence the implementation relation between specification and implementation can only be confirmed by means of testing the implementation.

But the testing of systems has to be done in finite time. Therefore often the tests done on the implementation cannot be exhaustive. So, what are the relations for which tests can be performed? The work found for example in [1] or [9] strives for finding tests that *exactly* discriminate between correct and erroneous implementations based on a given implementation relation.

Yet in practical testing, test cases are formulated to validate the presence of some *specific* properties in implementations, the so–called test purposes. This clearly does not root out

all possible causes for errors or non–conformant behavior of the implementation, but it does give some certainty about some chosen aspects of the desired system. How does this kind of practical testing relate to the known implementation relations? In the scope of this paper, we will examine the relationship between the idea of test generation based on test purposes and the idea of implementation relations between specifications and their implementations.

Our paper proceeds as follows: First, we give some preliminaries and recapitulate some notions from conformance testing. Then, we show how these notions correlate to those known from the field of formal methods, and how they induce a family of parameterized implementation relations. We investigate the relationship of these parameterized relations to one of the well studied implementation relations known from formal methods. We close this paper with some conclusions and outlook on further work.

2 Conformance Testing

Before investigating the connection between the implementation relations known from formal methods and practical conformance testing, we want to remind the reader of some of the central points in the formal theory for conformance testing. We give a short definition of the notion of conformance, as it can be found in [5].

The definition of conformance concerns implementations under test (IUT) and specifications. We make the so–called *test assumption* that implementations under test IUT can be modeled by a formal objects i_{IUT} in a formalism $MODS$, to which we refer as the universe of models. Yet, this assumption only assumes that such models *exist*, but not that they are known *a priori*.

The test assumption allows us to reason about implementations as if they were formal objects. The conformance of implementations with respect to formal specifications can be expressed by means of an implementation relation. An implementation relation **imp** is a relation between the set of models of implementations $MODS$ and the set of specifications $SPECS$, i. e.,

$$\mathbf{imp} \subseteq MODS \times SPECS.$$

An implementation IUT now is considered **imp**–correct with respect to a specification $s \in SPECS$, if and only if $i_{IUT}\ \mathbf{imp}\ s$.

The behavior of a concrete implementation is investigated by performing experiments on the implementation, observing the reactions of it to these experiments. Such experiments are called tests, and are formally specified as elements of an universe of test cases $TESTS$. A set of test cases is called a test suite. The process of running a test against a concrete implementation is called *test execution*, and it gives rise to an observation in a domain of observations OBS. The result of the test execution is defined by a verdict assignment $verd_t$, which may depend on the test case t. This assignment has the signature

$$verd_t : OBS \to \{\mathsf{pass}, \mathsf{inconclusive}, \mathsf{fail}\}.$$

An implementation under test IUT passes a test case t, if and only if the test execution of the IUT with t leads to an observation $O \in OBS$ to which $verd_t$ assigns a pass verdict:

$$IUT\ \mathbf{passes}\ t \iff verd_t(O) = \mathsf{pass} \tag{1}$$

For the interpretation of test case execution, a function has to be defined. The function *exec* calculates the observations for models i_{IUT} of implementations under test contained in a model of a test context C. This function has the signature

$$exec : TESTS \times MODS \rightarrow OBS.$$

With this function definition (1) can be made more concrete, i. e.

$$IUT \textbf{ passes } t \iff verd_t(exec(t, C(i_{IUT}))) = \mathsf{pass}.$$

The subset of models in *MODS* for which test execution yields a pass verdict is called the formal test purpose P_t of t:

$$P_t = \{m \in MODS \mid verd_t(exec(t, C(m))) = \mathsf{pass}\}$$

Thus, the objective of testing an IUT with a test case t is to conclude whether the model i_{IUT} of the IUT is a member of the formal test purpose P_t of t, i. e.

$$IUT \textbf{ passes } t \iff i_{IUT} \in P_t.$$

In the following sections, we abstract from the test context C, assuming it to be encoded into the test case. Note that the definitions given above only define the signatures of the functions $verd_t$ and *exec*. Hence, in Section 3.3 and Section 3.4 we will instantiate the classes for models, tests and observations as well as these functions. The goal is to find out which implementation relations are tested in practical testing.

3 Conformance Testing in the Light of Formal Methods

After having recapitulated some of the notions of the formal treatment of conformance testing, we are now going to examine which implementation relations are tested in conformance testing, directed by the use of test suites related to certain test purposes. With conformance testing we refer in the context of this paper to functional black box testing, not the formalization shown in the previous section. We first define some languages for the description of systems and tests, and will then take a look at the situation found in testing from a formal perspective. Then, we will examine implementation relations based on test purposes and test cases.

3.1 Languages for the Description of Systems and Tests

For the description of systems, test purposes and test cases we need some languages. Since we are interested in linking our intuitions on conformance testing to the notions known from formal methods, we need a unified model for systems and tests. Thus we are going to use a process algebraic one instead of standardized ones like SDL, MSC or TTCN.

So let us start by developing a process algebra for describing the behavior of systems. As a basis, we choose the process algebra given by Winskel and Nielsen in [12]. Its syntax is given by

$$p ::= \mathsf{nil} \mid ap \mid p_0 \oplus p_1 \mid p_0 \times p_1 \mid p \restriction \Lambda \mid p\{\Xi\} \mid x \mid rec\ x.p,$$

where a is a label, Λ is a subset of labels and Ξ is a total function from labels to labels. We will denote the class of processes generated with this syntax with $\mathbb{P}_{\text{proc}}$. In this language, nil denotes a process that terminates instantly; ap denotes a process that performs the action a and then behaves like p; $p_0 \oplus p_1$ denotes the choice between the behaviors of the processes p_0 and p_1; $p_0 \times p_1$ denotes the behavior of two processes p_0 and p_1 observed in parallel, the observations being pairs of labels or $*$, $*$ being an idling action of a process; $p \upharpoonright \Lambda$ denotes the restriction of the behavior of p to just those actions labelled with symbols in Λ; $p\{\Xi\}$ denotes a relabelling of the actions in p according to the labelling function Ξ; and $rec\ x.p$ denotes a recursive behavior with x being a process variable.

For this process algebraic language, we give a structural operational semantics along the lines of [8]. First, we introduce a notation concerning the combination of labels forced by the product operator $\times$. For labels a, b, we define:

$$a \times b = \begin{cases} * & \text{if } a = b = *, \\ (a, b) & \text{otherwise} \end{cases}$$

This notation together with the use of idle transitions results in a compact SOS rule for the product operator. The transitions between states of a system, which are identified with closed terms of the process algebraic language, are given by the following rules:

$$\frac{}{ap \xrightarrow{a} p} \qquad \frac{}{p \xrightarrow{*} p}$$

$$\frac{p_0 \xrightarrow{a} p_0'}{p_0 \oplus p_1 \xrightarrow{a} p_0'}\ a \neq * \qquad \frac{p_1 \xrightarrow{a} p_1'}{p_0 \oplus p_1 \xrightarrow{a} p_1'}\ a \neq *$$

$$\frac{p_0 \xrightarrow{a} p_0' \qquad p_1 \xrightarrow{b} p_1'}{p_0 \times p_1 \xrightarrow{a \times b} p_0' \times p_1'}$$

$$\frac{p \xrightarrow{a} p'}{p \upharpoonright \Lambda \xrightarrow{a} p' \upharpoonright \Lambda}\ a \in \Lambda \qquad \frac{p \xrightarrow{a} p'}{p\{\Xi\} \xrightarrow{\Xi(a)} p'\{\Xi\}}$$

$$\frac{p[rec\ x.p/x] \xrightarrow{a} p'}{rec\ x.p \xrightarrow{a} p'}\ a \neq *$$

A closed term p determines a transition system with initial state p consisting of all states and transitions reachable from p. The usual parallel composition operator $\|_A$ synchronizing on actions labelled with symbols in A known from process algebra can be recovered from these operators as a combination of product, restriction and relabelling.

For the description of test purposes and test cases, we need a slightly different process algebraic language, the syntax of which is given by

$$t ::= \mathsf{pass} \mid \mathsf{fail} \mid \mathsf{inconclusive} \mid at \mid t_0 \oplus t_1 \mid t_0 \times t_1 \mid t \upharpoonright \Lambda \mid t\{\Xi\} \mid x \mid rec\ x.p,$$

where again a is a label, Λ is a subset of labels and Ξ is a total function from labels to labels. We denote the class of processes generated with this syntax with $\mathbb{P}_{\text{te}}$. As one can see, the only difference between the two languages are the exclusion of the process constant nil from the language t and the inclusion of the constants pass, fail and $\mathsf{inconclusive}$. This is done because we want to force test purposes and test cases to end in one of the test verdicts pass, fail or $\mathsf{inconclusive}$. We will interpret the constructions in the same way as for those used for the construction of $\mathbb{P}_{\text{proc}}$, except for the new processes pass, fail and $\mathsf{inconclusive}$. For these, we give the follownig SOS rules:

$$\overline{\mathsf{pass} \xrightarrow{\mathsf{pass}} \mathsf{nil}} \qquad \overline{\mathsf{fail} \xrightarrow{\mathsf{fail}} \mathsf{nil}} \qquad \overline{\mathsf{inconclusive} \xrightarrow{\mathsf{inconclusive}} \mathsf{nil}}$$

So now we have a language to describe systems with and another one to describe tests. We imbed these two langauges into a third one, describing systems *and* executions of test cases. The syntax of this language is given by

$$s ::= t \rhd_\Lambda p \mid p,$$

where $\rhd$ is a parameterized operator denoting the application of a test to a system and Λ is a subset of labels. The class of processes generated with this syntax will be denoted with $\mathbb{P}_{\text{sys}}$. The new SOS rules for the operator $\rhd$ are the following:

$$\frac{t \xrightarrow{a} t'}{t \rhd_\Lambda p \xrightarrow{a} t' \rhd_\Lambda p}\ a \notin \Lambda \qquad \frac{p \xrightarrow{a} p'}{t \rhd_\Lambda p \xrightarrow{a} t \rhd_\Lambda p'}\ a \notin \Lambda$$

$$\frac{t \xrightarrow{a} t' \qquad p \xrightarrow{a} p'}{t \rhd_\Lambda p \xrightarrow{a} t' \rhd_\Lambda p'}\ a \in \Lambda$$

Hence, this new operator acts as a typed kind of parallel composition operator, expecting a test as its left argument and a process as its right argument.

In the following, we will use Λ to denote the set of labels for actions. We assume that $\{\mathsf{pass}, \mathsf{fail}, \mathsf{inconclusive}\} \subseteq \Lambda$, but that only processes in $\mathbb{P}_{\text{te}}$ are allowed to synchronize on pass, fail or $\mathsf{inconclusive}$, as is needed for concurrent test architectures. Furthermore, $\tau \in \Lambda$ denotes a special silent action, which cannot be used to synchronize on.

In the context of the formal theory of conformance testing presented in Section 2, we will instantiate the class of models $MODS$ with $\mathbb{P}_{\text{proc}}$ and the class of tests $TESTS$ with $\mathbb{P}_{\text{te}}$.

3.2 The Situation in Testing

When investigating the relationship between a specification and an implementation, we have to differentiate between two scenarios. On the one hand, if we can see the inner workings of specification and implementation, we are able to state the implementation relation between specification s and implementation i as

$$i \sqsubseteq_1 s$$

for some given implementation relation $\sqsubseteq_1$. This glass box view of specifications and implementations is usually used for verification purposes.

Yet, often we only are able to observe events at the interface of an implementation. If we assume $\Lambda_{IO} \subseteq \Lambda$ to be the set of events observable on the interface of the implementation, we may state the implementation relation between these two "black boxes" as

$$i\{\Xi_{IO}\} \sqsubseteq_2 s\{\Xi_{IO}\},$$

with Ξ_{IO} being the relabelling function defined as

$$\Xi_{IO}(a) = \begin{cases} a, & a \in \Lambda_{IO} \\ \tau, & a \notin \Lambda_{IO}, \end{cases}$$

τ being used to abstract from internal actions. Hence, with respect to this relation the implementation may do some radically different things internally as long as the visible behavior is

conforming to that of the specification. Thus, we want this new implementation relation to be weaker than $\sqsubseteq_1$, i. e.

$$i \sqsubseteq_1 s \Rightarrow i\{\Xi_{IO}\} \sqsubseteq_2 s\{\Xi_{IO}\}.$$

So far, we only look at the *complete* behaviors of the systems. If we reduce our interest to just those behaviors that follow a specific purpose, we get some different results. *In praxi*, test purposes are used to define narrowly defined objectives of testing, focusing on a single or some closely related conformance requirements. Here we assume a test purpose to be a process in $\mathbb{P}_{\mathrm{te}}$ whose behavior reflects the desired properties.

When looking at just one test purpose, we may state an implementation relation as

$$p \rhd_\Lambda i \sqsubseteq_3 p \rhd_\Lambda s,$$

hence we restrict the behavior of s and i to just the "interesting" parts where they behave like p. We expect this implementation relation to be weaker than the original one, as

$$i \sqsubseteq_1 s \Rightarrow p \rhd_\Lambda i \sqsubseteq_3 p \rhd_\Lambda s.$$

Similarly, we want to find a weaker counterpart for the relation $\sqsubseteq_2$ by restricting the focus of interest to just specific behaviors. Here, we synchronize a test process with specification and implementation. While the test purpose used in the definition of $\sqsubseteq_3$ is able to observe even events internal to specification and implementation, the test process only may observe events at the interface to the environment. Hence, we may state a fourth implementation relation $\sqsubseteq_4$ as

$$t \rhd_{\Lambda_{IO}} i\{\Xi_{IO}\} \sqsubseteq_4 t \rhd_{\Lambda_{IO}} s\{\Xi_{IO}\}$$

for some test process t. Here, we want $\sqsubseteq_4$ to be weaker than $\sqsubseteq_2$, i. e.

$$i\{\Xi_{IO}\} \sqsubseteq_2 s\{\Xi_{IO}\} \Rightarrow t \rhd_{\Lambda_{IO}} i\{\Xi_{IO}\} \sqsubseteq_4 t \rhd_{\Lambda_{IO}} s\{\Xi_{IO}\}.$$

The relationship between these four views of specification, implementation, test purposes and test cases can be visualized as in Figure 1. There, the dotted lines show the implementation relations, while the solid arrows denote inclusion functions mapping processes onto other processes, e. g. by hiding internal behavior.

3.3 Implementation Relations Based on Test Purposes

Now we have languages for the description of systems and tests at our disposal and we have investigated the view from the field of formal methods onto conformance testing. Therefore we can go on to develop an implementation relation that reflects what is being tested in practical conformance testing.

Before doing this, we have to translate some of the terms from Section 2 to the setting described above. This especially holds for the functions *exec* and $verd_t$. We also have to define the notion of observability which we follow in our model.

As our notion of observability (and hence class of observations OBS) we choose *traces*, defining for a process $p \in \mathbb{P}_{\mathrm{sys}}$ the set of traces of p as

$$Traces(p) = \{\sigma \mid p \stackrel{\sigma}{\Longrightarrow}\},$$

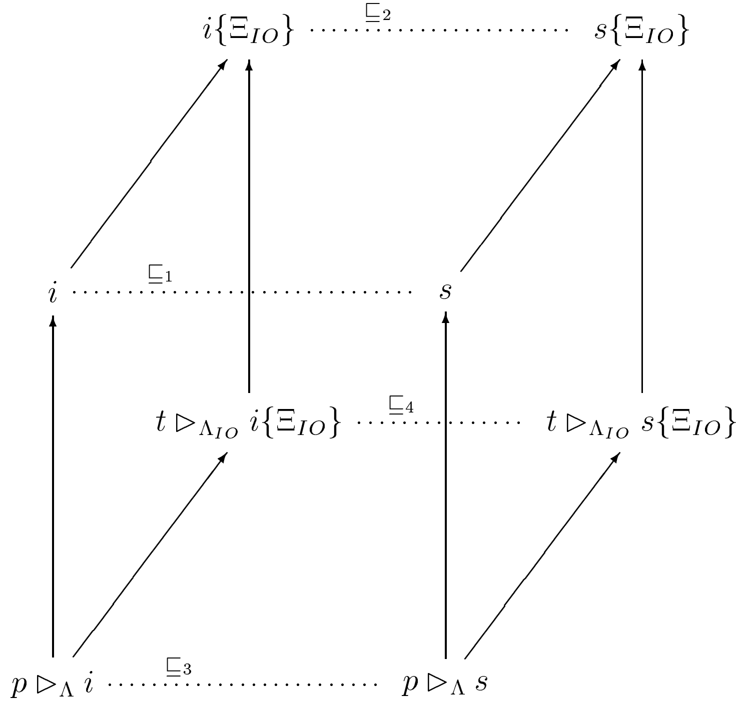

Figure 1: Relationship between Testing Scenarios

with the transition relation $\Longrightarrow$ being defined as

$$
\begin{aligned}
p \overset{\sigma}{\Longrightarrow} p' \iff \; & \exists n \in \mathbb{N}. \exists a_1, \ldots, a_n \in \Lambda, p_0, p_1, \ldots, p_n \in \mathbb{P}_{\text{sys}}. \\
& \qquad p = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} p_n \wedge \sigma = (a_1 a_2 \ldots a_n) \upharpoonright (\Lambda - \{\tau\})
\end{aligned}
$$

and the shorthand notation

$$p \overset{\sigma}{\Longrightarrow} \iff \exists p' \in \mathbb{P}_{\text{sys}}. p \overset{\sigma}{\Longrightarrow} p'.$$

The filtering function $\cdot \upharpoonright \cdot$ used in the definition above is defined recursively as

$$
\begin{aligned}
\varepsilon \upharpoonright L &= \varepsilon \\
aw \upharpoonright L &= \begin{cases} a(w \upharpoonright L), & a \in L \\ w \upharpoonright L, & a \notin L. \end{cases}
\end{aligned}
$$

Next, we define the application $exec(t, p)$ of a test t to an implementation i as the set of traces of the (process algebraic) application of t to i, i. e.

$$exec(t, i) = Traces(t \rhd_\Lambda p).$$

Now, the verdict function $verd_t$ can be defined as

$$verd_t(O) = \begin{cases} \mathsf{pass}, & \text{if } \forall \sigma \in O. A_{\sigma \upharpoonright \{\mathsf{pass}, \mathsf{fail}, \mathsf{inconclusive}\}} = \{\mathsf{pass}\} \\ \mathsf{fail}, & \text{otherwise}, \end{cases}$$

where A_σ is the alphabet of the trace σ, i. e. the set of labels constituting σ. Hence, an application of a test case to an implementation results in a pass verdict only when the application cannot lead to a fail result.

So, which implementation relation in the sense known from formal methods is checked with this kind of testing? From the computation of the test verdict, one can define a two place predicate **passes** defined as

$$i \,\mathbf{passes}\, t \iff verd_t(exec(t,i)) = \mathsf{pass}.$$

Hence, we can define an implementation relation based on a test t as

$$s \sqsubseteq s' \iff s' \,\mathbf{passes}\, t \Rightarrow s \,\mathbf{passes}\, t.$$

Moreover, we define a parameterized implementation relation $\sqsubseteq^T$ as

$$s \sqsubseteq^T s' \iff \forall t \in T. s' \,\mathbf{passes}\, t \Rightarrow s \,\mathbf{passes}\, t,$$

where the relation is parameterised over a set of tests T to be executed.

How does this compare to the implementation relations known from formal methods? The closest match is the may–testing preorder $\sqsubseteq_{\text{may}}$ from [7], for which

$$s \sqsubseteq_{\text{may}} s' \iff \mathit{Traces}(s') \subseteq \mathit{Traces}(s)$$

holds. An alternative formulation (adapted to the notation we are using) of this relation is

$$s \sqsubseteq_{\text{may}} s' \iff \forall t \in \mathbb{P}_{\text{te}}. s' \,\mathbf{passes}\, t \Rightarrow s \,\mathbf{passes}\, t.$$

Comparing this to the relation presented above, one sees that

$$s \sqsubseteq_{\text{may}} s' \Rightarrow s \sqsubseteq^T s' \tag{2}$$

for every set of tests T.

Clearly, the reverse direction of this implication does not hold. Yet, looking at a sequence of sets of tests $(T_i)_{i\in\mathbb{N}}$ with $T_i \subseteq T_{i+1}$ for $i \in \mathbb{N}$, we get the following result (abusing mathematical notation a bit):

$$\lim_{n\to\infty} \sqsubseteq^{T_n} = \sqsubseteq_{\text{may}}$$

Hence, increasing the size of the test sets would approximate the may–testing preorder $\sqsubseteq_{\text{may}}$.

So now we are able to define an implementation relation that catches the situation in glass box testing, when we are able to control even the inner workings of specification and implementation. Here we define a weaker version of the may–testing preorder, restricting the tests applied to the systems to a set of test purposes, hence

$$i \sqsubseteq^P_{\text{may}} s \iff \forall p \in P. s \,\mathbf{passes}\, p \Rightarrow i \,\mathbf{passes}\, p.$$

This way, we get the desired property

$$i \sqsubseteq_{\text{may}} s \Rightarrow i \sqsubseteq^P_{\text{may}} s,$$

and inherit the convergence property

$$\lim_{n\to\infty} \sqsubseteq^{P_n}_{\text{may}} = \sqsubseteq_{\text{may}}$$

rewritten from above for some sequence of test purposes $(P_n)_{n\in\mathbb{N}}$ increasing with respect to set inclusion.

3.4 Implementation Relations Based on Test Cases

Now we know which implementation relation can be tested by glass box testing using test purposes. Next, we transfer these results to black box testing, that is, conformance testing. For this, we first have to define an implementation relation parameterized by sets of test cases instead of sets of test purposes. Next, we have to investigate how this new preorder relates to the one defined in the previous section.

Let $s \in \mathbb{P}_{\mathsf{sys}}$ be a system and $p \in \mathbb{P}_{\mathsf{te}}$ be a test purpose. A test case $t \in \mathbb{P}_{\mathsf{te}}$ for p is a process labelled over Λ_{IO}, the alphabet of the interface of the system to its environment. Citing from [3] with some adjustment of notation, this process basically has to fulfill the properties

$$p \parallel_{\Lambda_{IO} \cup \{\mathsf{pass},\mathsf{fail},\mathsf{inconclusive}\}} t \ \textbf{must succeed}, \tag{3}$$

i. e. the test case t has to be a part of the externally visible behavior of p,

$$t \rhd_{\Lambda_{IO}} s \ \textbf{must succeed}, \tag{4}$$

i. e. the test runs have to lead to reproducible results, summarised in

$$(t \parallel_{\Lambda_{IO}} p) \rhd_{\Lambda} s \ \textbf{must succeed}, \tag{5}$$

i. e. the test process t has to generate a trace in the test purpose p that is guaranteed to end in a pass verdict when synchronized with the specification s.

The predicate **must succeed** used in these properties is defined as

$$s \ \textbf{must succeed} \iff \forall \sigma \in \Lambda^*, s' \in \mathbb{P}_{\mathsf{sys}}.(s \stackrel{\sigma}{\Longrightarrow} \sigma' \Rightarrow \exists a \in \Lambda \cup \{\mathsf{pass}\}.s' \stackrel{a}{\Longrightarrow}),$$

with the shorthand notations

$$s \stackrel{a}{\longrightarrow} \iff \exists s' \in \mathbb{P}_{\mathsf{sys}}.s \stackrel{a}{\longrightarrow} s'$$

and

$$s \stackrel{a}{\not\longrightarrow} \iff \neg\exists a \in \Lambda.s \stackrel{a}{\longrightarrow} .$$

The predicate **must succeed** states that the test application is successful if it cannot deadlock visibly without having signalled successful termination of the test beforehand.

For black box testing, we choose sets of traces over Λ_{IO} to be our notion of observability. Next, we adapt the definition of $exec(t, p)$ from the previous section to the restricted capability of the test process, i. e. we define

$$exec(t, i) = Traces(t \rhd_{\Lambda_{IO}} p),$$

while the verdict assignment function $verd_t$ can be defined as in Section 3.3.

Again, we define a two place predicate **passes** as

$$i \ \textbf{passes} \ t \iff verd_t(exec(t, i)) = \mathsf{pass},$$

as well as a parameterized implementation relation $\sqsubseteq^T$ as

$$s \sqsubseteq^T s' \iff \forall t \in T.s' \ \textbf{passes} \ t \Rightarrow s \ \textbf{passes} \ t,$$

where the relation is parameterized over a set of test cases T to be executed.

Now let us compare this new parameterized relation to the may–testing preorder. Here we find that

$$\forall T \subseteq \mathbb{P}_{\text{te}}.i\{\Xi_{IO}\} \sqsubseteq_{\text{may}} s\{\Xi_{IO}\} \Rightarrow i \sqsubseteq^{T} s,$$

thus we are able to formulate a convergence property similar to the one presented in Section 3.3. For this, we define an implementation relation $\sqsubseteq_{\Lambda_{IO}-\text{may}}$, called *interface may–testing preorder with respect to the interface alphabet* Λ_{IO}, as

$$p \sqsubseteq_{\Lambda_{IO}-\text{may}} q \iff p\{\Xi_{IO}\} \sqsubseteq_{\text{may}} q\{\Xi_{IO}\}.$$

With this implementation relation, we are able to state the convergence property as

$$\lim_{n\to\infty} \sqsubseteq^{T_n} = \sqsubseteq_{\Lambda_{IO}-\text{may}}$$

for some sequence of test cases $(T_n)_{n\in\mathbb{N}}$ labelled over an interface alphabet Λ_{IO}, which are increasing with respect to set inclusion.

How does this new implementation relation parameterized over test cases relate to the one given in the previous section? We easily see that

$$\forall p, q \ni \mathbb{P}_{\text{proc}}.p \sqsubseteq_{\text{may}} q \Rightarrow p \sqsubseteq_{\Lambda_{IO}-\text{may}} q$$

holds, where Λ_{IO} is the alphabet of the interface of p and q to their environment. We expect the same to hold for the parameterized counterparts of these relations.

So let P be a set of test purposes, and let T be a set of test cases for a system s and the respective test cases in P. Guaranteed by property 3, we have that

$$i \sqsubseteq^{\{p\}} s \Rightarrow i \sqsubseteq^{\{t\}} s$$

holds for p being a test purpose from P, t being the test case for p and s and i being an implementation. Hence, we have that

$$i \sqsubseteq^{P} s \Rightarrow i \sqsubseteq^{T} s,$$

as we postulated in the framework from Section 3.2.

Thus we also are able to define an implementation relation that catches the situation in black box testing, when we are only able to control the workings of specification and implementation by communicating with their respective interfaces. Here we define a weaker version of the interface may–testing preorder, restricting the tests applied to the systems to a set of test cases T, hence

$$i \sqsubseteq^{T}_{\Lambda_{IO}-\text{may}} s \iff \forall t \in T.s \textbf{ passes } t \Rightarrow i \textbf{ passes } t.$$

This way, we get the desired property

$$i \sqsubseteq_{\Lambda_{IO}-\text{may}} s \Rightarrow i \sqsubseteq^{T}_{\Lambda_{IO}-\text{may}} s,$$

as well as inherit the convergence property

$$\lim_{n\to\infty} \sqsubseteq^{T_n}_{\Lambda_{IO}-\text{may}} = \sqsubseteq_{\Lambda_{IO}-\text{may}}$$

rewritten from above for some sequence of test cases $(T_n)_{n\in\mathbb{N}}$ increasing with respect to set inclusion.

4 Conclusions and Further Work

The work presented here shows how the notion of implementation relations known from formal methods and ideas of conformance testing correlate. We have shown that the implementation relations realized through the practical testing of systems come from a family of parameterized implementation relations. We also have shown that for glass box testing implementation relations parameterized by test purposes converge to the may–testing preorder of DeNicola and Hennessy [7], while for black box testing implementation relations parameterized by test cases converge to a may–testing preorder of the behavior visible at the interface to the environment.

But this is just *one* special implementation relation that is interesting from the formal point of view. There are a number of other preorders and equivalences formalizing different notions of the equivalence of observable behavior in specifications and implementations. It is interesting and an aim for future work to see how these implementation relations could be practically tested using parameterized implementation relations. Here, a modular approach reminding to that of van Glabbeek [10, 11] would be preferable.

References

[1] E. Brinksma. A theory for the derivation of test cases. In *Protocol Specification, Testing and Verification VIII*, pages 63–74. North–Holland, 1988.

[2] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. Technical Report 91-24, Technical University of Aachen (RWTH Aachen), 1991.

[3] S. Heymer and J. Grabowski. Generating Test Cases for Infinite System Specifications. In *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG–Fachgespräch*, number 315 in GMD–Studien, pages 221–230. GMD–Forschungszentrum Informationstechnik GmbH, 1997.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[5] ISO/IEC JTC1/SC21 WG7. *Information Retrieval, Transfer and Management for OSI, Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245–1*. ISO, 1997.

[6] R. Milner. *Communication and Concurrency*. Prentice–Hall, 1989.

[7] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[8] G. Plotkin. A Structural Approach to Operational Semantics. Technical report, Aarhus University, 1981.

[9] J. Tretmans. Test generation with inputs, outputs, and quiescence. In *Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, number 1055 in Lecture Notes in Computer Science, pages 127–146. Springer–Verlag, 1996.

[10] R. J. van Glabbeek. The linear time — branching time spectrum. In *CONCUR '90*, number 458 in Lecture Notes in Computer Science, pages 278–297. Springer–Verlag, 1990.

[11] R. J. van Glabbeek. The linear time — branching time spectrum II (the semantics of sequential systems with silent moves). In *CONCUR '93*, number 715 in Lecture Notes in Computer Science, pages 66–81. Springer–Verlag, 1993.

[12] G. Winskel and M. Nielsen. *The Handbook of Logic in Computer Science*, chapter "Models for Concurrency". Springer–Verlag, 1992.