Chapter 1

Introduction

Welcome to study *formal conformance testing*! This is the grand subject of these lecture notes, which I have prepared for the course

T-79.190: Testing of Concurrent Systems,

lectured at Laboratory for Theoretical Computer Science, Helsinki University of Technology.

1.1 Goals

The aim of this course is to work as an in-depth introduction into the subject area of *formal conformance testing*, whatever it happens to mean. What I present here is partly a "reorganized synthesis" of what has been said elsewhere (during the last decades), partly something I have myself understood or developed while working in the industry.

After taking this class, you should be able to

- Understand academic papers on the subject and study the area further.
- Understand tools and methodologies that are based on the theories and ideas related to formal conformance testing.
- Be able to successfully take part in industrial projects where formal conformance testing is either implemented or exploited.

Now these goals must be taken with a grain of salt, because (1) formal conformance testing is not in very wide spread use yet, and (2) there is no established, single theory of formal conformance testing. The latter is especially true in the context of systems with large (= infinite) state spaces.

1.2 Software Testing in General

For our purposes, [software] testing is the process of

1. **Interacting** with a system and
2. **evaluating** the results of that interaction, in order to
3. **determine** if the tested system **conforms to its operational specification**.

This is a very broad definition for testing. It covers most if not all forms of testing in practice. Namely, to test a system, you must be able to interact with it. A system that you can not access nor observe is not interesting from the viewpoint of testing. On the other hand, if there are no requirements for the system's behavior (i.e. there is no operational specification), then we know *a priori* that the system will pass all tests and it is again pointless to test at all.

Hence, we must (1) interact, and (2) there must be an operational specification (albeit potentially very informal one). The evaluation process links the results of interaction to the specification, yielding a verdict on the system being tested. Clearly, the testing verdict must be based on the specification by the argument above. So, in a sense, the above definition is almost trivial. It is nevertheless a useful way to define testing on abstract level, and it mentions implicitly all important components of a general testing setup.

© Antti Huima 2004. All rights reserved.

1.3 Formal Conformance Testing

We shall now turn our attention to a certain form of testing, namely *formal conformance testing*. In itself, this is a term that often appears in academic literature. It is, however, not very descriptive, or at least it can be misleading. Therefore it is important to define what I mean, in the context of this class, by this term.

Let us look first on the various words in this term.

"Testing" denotes the process of testing, as described above. Thus, it means that we interact with a system, evaluate the results, and determine based on the results of these evaluation if the system works correctly or not.

"Conformance" means here that we specifically focus on a testing setup where the correctness of the system under test's behavior is defined in terms of an *operational specification*. A system that works correctly *conforms* to its specification. This is the essence of this word here. There are other "forms" of testing, such as *load testing*. The purpose of load testing is to determine how a system works under various working loads. Now it is possible that a system fails a load test, for example by responding too slowly, or by crashing under heavy loads. Actually, of course, a system that fails a load test fails to conform to some requirements imposed on this system. In this sense, all testing could be seen to be "conformance testing", because the activity of testing always implies some form of requirements or specifications, either explicit or implicit. But, in practice, the word "conformance" emphasizes the use of a relatively clear, behaviorally oriented specification as the basis for evaluating the correctness of a system.

"Formal" here does *not* mean, that the testing process itself would be formal (involving rigorous organizational processes and structures, lots of forms to be filled, fixed-form reports *et cetera*), but that the basic testing setup has been mathematically formalized, and that the testing activities and processes are founded upon this formalization.

Let us now try to identify the main concepts that we need to understand the [formal] [conformance] testing setup in general.

1.3.1 System Under Test

Maybe the most self-evident entity in an FCT setup is the *system under test*, which is commonly abbreviated to SUT. The SUT is also known as "the black box". It is some form of a system implementation whose correctness should be evaluated with respect to an operational specification, that is, a specification that tells how the black box should operate.

Traditionally, an SUT can be subdivided into various parts, of which one is the "implementation under test", abbreviated by IUT. The difference between an IUT and the larger SUT is that the SUT can contain environment simulations, test adapter layers and other such artifacts that are required to make the real, original implementation IUT testable. But the distinction between SUT and IUT is not very important in our context now, so let us disregard it and just assume SUT = the black box we are testing.

1.3.2 Tester

In addition to an SUT, another active entity is required, which is (not surprisingly) a *tester*. A tester interacts with an SUT, evaluates whether the results of the interaction correspond with the related specification, and announces the *test verdict*, which is usually either "passed" (black box works this far correctly) or "failed" (black box works incorrectly).

It is useful to make a separation between (1) interacting with the SUT, and (2) determining the correctness of the SUT's behavior. However, these both activities are carried by the same entity, which is the tester.

1.3.3 Operational Specification

The wished-for, envisioned, expected, axiomatically correct behavior of a system is described in its *operational specification*. This is the tester's Bible; it tells which behaviors are acceptable for the system under test and which are not.

Together, an SUT, a tester, and an operational specification form the basic "triangle drama" of formal conformance testing (see Fig. 1.1).

© Antti Huima 2004. All rights reserved.

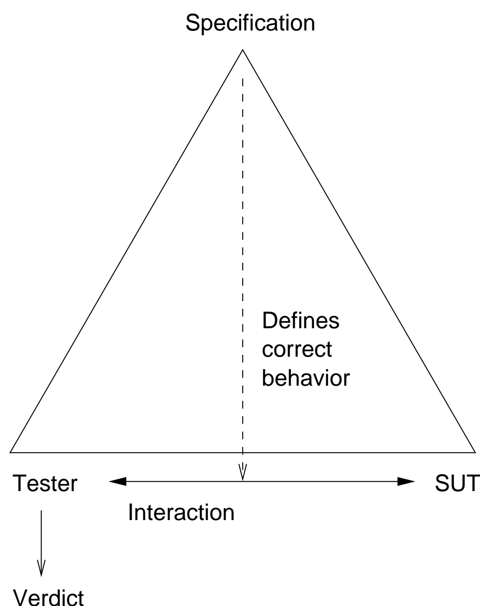

Figure 1.1: The main players.

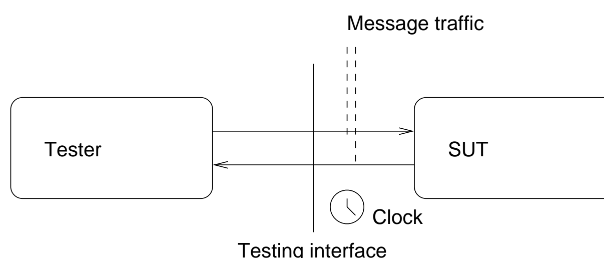

Figure 1.2: Testing interface.

1.4 Testing Interface

The point of interaction between a tester and an SUT can be called the *testing interface*. We make some rather general assumptions about the testing interface. Namely, it must be bidirectional. It must be capable of transmitting *messages*. The delivery of a single message over a testing interface must happen atomically, in either direction. It must be possible to determine the time of a transmission over the interface; in other words, there must be a clock on the interface.

1.5 Traces

A *trace* is a set of events that "happen" at the testing interface. Thus, every event consists of a message that has been transmitted, direction for that event, and the time (clock reading) when the message passed through the testing interface.

We make the general assumption that two events can never occur at *exactly the same time* at the testing interface. This assumption could be dispensed with, but then we would need to consider multisets of events instead of "normal" sets. That would complicate the presentation somewhat without adding much value. Contemplating about the quantization of the space-time continuum and its implications to the potentiality of two events occurring at the "same" (whatever that is) "time" (whatever *that* is) is beyond our scope.

There is a technical reason to augment the definition of a trace a bit, however. Namely, we must know *how long* we have observed the testing interface for events. Clearly, it is more interesting to observe that no message a transmitted for 100 seconds than to observe that no message has been transmitted in 2 seconds. Still, as event sets both traces are empty. Therefore

© Antti Huima 2004. All rights reserved.

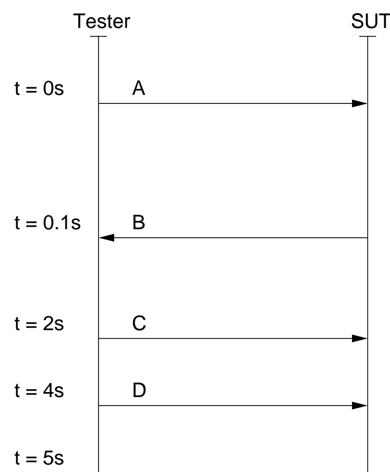

Figure 1.3: A trace.

the final definition for a trace is: a set of directional, timed message transmissions, and a clock reading when the trace finishes. We will come back to this soon when we formalize this and other concepts.

In Fig. 1.3, a trace has been depicted. The trace consists of four message transmissions, of which three are inputs (A, C and D) and one is an output (B). These events take place after zero, one tenth, two, and four seconds, respectively. The trace itself ends at five seconds since the beginning. This indicates that it is known that after the message D, there was at least one second of silent time, but that it is not known what happened *after* five seconds had passed.

1.6 A Concurrent Variant of Scheme

"There is no point in being too abstract." We want to follow this brilliant advice, and therefore wish to see examples of the constructs that we study.

To present examples, we need some form of notation for computational processes. Typically people call some of these notations "programming languages". Other notations for computational processes, not traditionally seen as programming languages, include for instance Petri nets, labeled transition systems, timed hybrid automata, UML state charts, Turing machines. It has been estimated that there are at least 500 programming languages currently in active use around the world. So we have a big arsenal from which to choose from.

"Structure and Interpretation of Computer Programs" is a good book about Scheme. You can find it online at http://mitpress.mit.edu/sicp/.

After strong deliberation, we choose to employ Scheme, a clean dialect of LISP, as our principal notation for computational processes.

Standard Scheme, however, has no constructs for timing aspects, message passing, nor for concurrent programming. We must therefore augment Scheme with some simple constructs for these operations, and describe their semantics. We shall call the result just Scheme, in the context of this class, because the result is not prominent enough to warrant a name of its own.

1.6.1 Extensions

Thus, we assume the existence of the following procedures and special forms.

Spawn. There is a procedure named `spawn` that is called as

(`spawn` *thunk*). (1.1)

This procedure call creates a new *thread*, a principal unit of dynamic execution, and returns nothing. The thread begins to execute by calling the procedure *thunk*, and terminates it execution when (if ever) the procedure call returns.

Make rendezvous point. There is a procedure named `make-rendezvous-point` that is called as

(`make-rendezvous-point`). (1.2)

This procedure call creates a new *rendezvous point*, which is a dynamically created point for internal synchronization and communication.

The French word *rendez-vous* denotes appointment or meeting. In the world of concurrent software, rendezvous denotes a particular form of remote procedure call. In our setting, what we have is more exactly *synchronous message passing*, but calling the "handles" at which this message passing occurs "rendezvous points" makes sense anyway.

© Antti Huima 2004. All rights reserved.

Sync. There is a special form named `sync` that is used to both communicate with the external world (e.g. a tester), and to communicate with other threads, and to wait for some time. The general call format is

$$(\texttt{sync}\ \textit{syncitem syncitem}\ \ldots). \tag{1.3}$$

Every *syncitem* is one of the following.

- (`output` e b): This denotes an attempt to send the value of the expression e to the environment. If the attempt is successful (it should be), computation continues by executing the body b.
- (`input` v b): This denotes an attempt to receive a value from the environment, and then continue with the body b. If a value is received, the variable v is bound lexically to the value and is visible from b, at where execution continues.
- (`write` p e b): This denotes an attempt to write the value of e to a rendezvous point p. If the attempt is successful, i.e. there is some other thread reading at the same point p, which then receives the value sent, execution continues from body b.
- (`read` p v b): This denotes an attempt to read a value from a rendezvous point p. If a value is available and received, i.e. if there is some other thread writing at the same point p, the variable v is bound lexically to the received value and is visible from b, at where execution continues.
- (`wait` t b): This denotes an attempt to wait for time of t seconds and then continuing with the body b. A wait clause like this becomes active (the body is entered) only if no output, input, write or read has been possible during the wait period.

A `sync` form works so that whenever an I/O event becomes possible, it is executed immediately. If there are multiple potential I/O events that could be carried out at the same moment, one is chosen nondeterministically. There can be multiple wait clauses in a sync form, but only the one with the least timeout time t is significant. If there are no wait clauses, thread waits indefinitely for I/O. In particular,

$$(\texttt{sync}) \tag{1.4}$$

causes the calling thread to block forever.

1.6.2 Examples

The following program denotes a computational process that accepts all inputs from the external world, but remains silent forever:

```
1 (define (run)
2   (sync (input x (run))))

3 (run)
```

Note how this works. The procedure *run* is self-recursive. It calls `sync` with the only I/O/ option being to receive a message from the external environment. The message is stored in the variable x, but actually the value of the variable is never read by the program. The self-recursive call to *run* occurs within the (input ...) subform.

Let us next try to define a system that echoes all messages it receives form the external world back. We could try the following:

```
1 (define (echo)
2   (sync (input x
3               (sync (output x (echo))))))

4 (echo)
```

© Antti Huima 2004. All rights reserved.

This program has, however, the problem that it responds back *at exactly the same time the message is received in the first place*. The reason is that we shall interpret the Scheme programs so that they execute *in zero time*, the only exception to this rule being the wait clauses in sync forms. But we have decided that two events should never occur at exactly the same time at a testing interface, because that is not physically possible. Therefore we could postulate that there is a time delay from one to hundred milliseconds before the answer comes, and write:

```
1 (define (echo)
2   (sync (input x
3                   (sync (wait (/ (+ 1 (random 99)) 1000)
4                                   (sync (output x (echo))))))))

5 (echo)
```

This is in principle correct, but note that now the system *does not accept input* from the external world while it is in the manually modeled wait. What if another message is received at that time? We decide not to implement a buffer (usually operating systems and applications have multiple buffers to keep track of messages that arrive while other tasks are being carried out) but to ignore all messages that come “too fast”. However, dropping the messages must again take *some* time, because otherwise the original problem of two events at exactly the same time at the testing interface would be reincarnated. So we get:

```
1 (define (echo)
2   (sync (input x (wait-and-send x))))

3 (define (wait-and-send x)
4   (sync (input y (wait-and-send x)) ; drop message and wait again
5         (wait (/ (+ 1 (random 99)) 1000)
6               (sync (output x (echo))))))

7 (echo)
```

This program should work correctly. Do you understand how it works?

Eventually we might find out that it is a bad idea to drop messages, and that it would be better to buffer them. Suppose we have an abstract queue data type available with the usual queue manipulation operations. We can then establish an echo system with a queue as below. This becomes more complex because we need multiple threads (if we postulate that we must always be able to receive more data), but this works as a good example on the use of the internal rendezvous points:

```
1 (define (producer point queue)
2   (if (empty-queue? queue)
3       (sync (input x (queue-insert! queue x)))
4       (sync (input x (queue-insert! queue x))
5             (write point (queue-front queue) (queue-remove! queue))))
6   (producer point queue))

7 (define (consumer point)
8   (sync (read point x
9                 (sync (wait (/ (+ 1 (random 99)) 1000)
10                                (sync (output x (consumer point))))))))

11 (define (run)
12   (let ((queue (make-queue))
13         (point (make-rendezvous-point)))
14     (spawn (λ () (producer point queue)))
15     (spawn (λ () (consumer point)))))

16 (run)
```

© Antti Huima 2004. All rights reserved.

Chapter 2

Formal Foundations

Our next aim is to defined formally what we mean by traces, operational specifications and other such interesting objects.

2.1 Traces

We will begin with traces.

2.1.1 Basics

We begin by assuming that we are aware of two sets: a set of *input messages*, namely messages that an SUT (of which we are interested of) can receive, and a set of *output messages*, which are those messages that the SUT could in principle send. We denote the set of input messages by Σ_{in} and the set of output messages by Σ_{out}. We shall assume that these sets are disjoint, i.e. that $\Sigma_{\text{in}} \cap \Sigma_{\text{out}} = \emptyset$. We do not lose any generality, because we could always just add some form of "direction marks" to messages to make the input and output message sets separate.

Traditionally, these sets are known as the input and output alphabet, respectively, as stated below.

DEFINITION 2.1. Σ_{in} is the *input alphabet*. Σ_{out} is the *output alphabet*. The sets are disjoint, $\Sigma_{\text{in}} \cap \Sigma_{\text{out}} = \emptyset$, and countable. The *I/O alphabet* is the set $\Sigma_{\text{in}} \cup \Sigma_{\text{out}}$, and is denoted by Σ.

It will be helpful later to assume that there exists a formal object that is outside the I/O alphabet. We will reserve the Greek symbol τ for this object, again on the basis of tradition.

DEFINITION 2.2. τ denotes an object such that $\tau \notin \Sigma$. The set Σ extended by τ is denoted by Σ_τ, i.e.

$$\Sigma_\tau = \Sigma \cup \{\tau\}. \tag{2.1}$$

We stated above that events, which are members of traces, consist of a message, direction information, and time stamp. Now the direction of a message has been embedded into the message itself — because the input and output alphabets are disjoint. Time stamps we represent by nonnegative real numbers. Thus we get the following definition.

DEFINITION 2.3. An *event* is a pair $\langle \alpha, t \rangle$ where $\alpha \in \Sigma$ and $t \in \mathbb{R}$, $t \geq 0$.

It is now easy to give the definition of a trace. We require that the event sets of traces finite, because they must be objects that testers are capable of observing wholly.

DEFINITION 2.4. A *trace* is a pair $\langle E, t \rangle$ where E is a finite set of events, all with distinct time stamps, and $t \in \mathbb{R}$, $t \geq 0$ such that also for all $\langle \alpha, t' \rangle \in E$ it holds that $t > t'$.

We will call the time value t, which denotes the clock reading at the end of a trace, the trace's *end time stamp*. Note that there can be no event in a trace exactly at the end time stamp. (This is a technicality that ensures that the empty trace $\langle \emptyset, 0 \rangle$ will function as a *prefix* for all traces, even those where the first event occurs exactly at the moment 0.)

A trace can be a *prefix* of another trace. A trace T is a prefix of another trace T' if the time span that T covers is contained in that of T', and if the events on the common time span are the same. In other words, we could expect to observe trace T' even after T has been already observed, because T can be extended to become T'. The formal definition is straightforward:

© Antti Huima 2004. All rights reserved.

DEFINITION 2.5. A trace $T = \langle E, t\rangle$ is a *prefix* of trace $T' = \langle E', t'\rangle$ if and only if both $t \leq t'$ and

$$E = \{\langle \alpha, \tilde{t}\rangle \mid \langle \alpha, \tilde{t}\rangle \in E' \wedge \tilde{t} < t\}. \tag{2.2}$$

T is a *proper prefix* of T' if additionally $T \neq T'$. We write $T \preceq T'$ to denote that T is a prefix of T', and $T \prec T'$ to denote that T is a proper prefix of T'. (That is, $T \prec T' \iff T \preceq T' \wedge T \neq T'$.

The trace $\langle \emptyset, 0\rangle$ is of particular interest, because it is the unique *empty trace*. We reserve the Greek letter ϵ for denoting this trace.

As additional terminology, we shall say that if T is a prefix of T', then T' is an *extension* of T. Clearly, being an extension is the "inverse" of being a prefix. But this concept is useful when we need to speak of, say, the set of all extensions of a given trace T. (Formally, this is the set $\{T' \mid T \preceq T'\}$.)

DEFINITION 2.6. ϵ denotes the empty trace $\langle \emptyset, 0\rangle$.

Example 2.7. Fig. 1.3 depicts the following trace:

$$\langle\{\langle A, 0\rangle, \langle B, 0.1\rangle, \langle C, 2\rangle, \langle D, 4\rangle\}, 5\rangle. \tag{2.3}$$

Here it must hold that

$$\{A, C, D\} \subseteq \Sigma_{\text{in}} \tag{2.4}$$

and

$$B \in \Sigma_{\text{out}}. \tag{2.5}$$

2.1.2 More Constructs

We define here some more mathematical constructs on traces, even though they are not needed immediately. But this is a good place to introduce them.

We will reserve a notation for the set of all the prefixes of a given trace T:

DEFINITION 2.8. For a trace T, $\mathbf{Pfx}(T)$ denotes the set of all prefixes of T, namely

$$\mathbf{Pfx}(T) = \{T' \mid T' \preceq T\}. \tag{2.6}$$

Example 2.9. In particular, for any T, $T \in \mathbf{Pfx}(T)$ and $\epsilon \in \mathbf{Pfx}(T)$.

A trace contains a set of events, and every event has a certain time stamp. It becomes useful to be able to "pick out" a single event from a trace when its time stamp is known. The following construct does this job.

DEFINITION 2.10. Let T be a trace $\langle E, t\rangle$ and κ a clock reading on the interval $[0, t)$. We define the meaning of the expression $T|_\kappa$ to be an element of Σ_τ according to the following:

$$T|_\kappa = \begin{cases} \alpha & \langle \alpha, \kappa\rangle \in E \\ \tau & \neg\exists \alpha : \langle \alpha, \kappa\rangle \in E \end{cases} \tag{2.7}$$

Example 2.11. Let $T = \langle\{\langle A, 1\rangle\}, 2\rangle$. Then $T|_1 = A$ and $T|_{1.2} = \tau$. $T|_3$ is not defined because $3 > 2$.

If T is a trace $\langle E, t\rangle$, we can be interested of that prefix of T whose end time stamp is t'. We write this prefix as $T[t']$. The exact definition follows.

DEFINITION 2.12. Let $T = \langle E, t\rangle$ be a trace and let $t' \in [0, t]$. Then $T[t']$ denotes the unique trace T' of the form $\langle E', t'\rangle$ such that $T' \preceq T$.

Example 2.13. For example, let $T = \langle\{\langle A, 1\rangle\}, 2\rangle$. Then $T[1.5] = \langle\{\langle A, 1\rangle\}, 1.5\rangle$, $T[0] = \epsilon$ and $T[0.5] = \langle \emptyset, 0.5\rangle$.

Suppose that we have two traces T and T' such that T is neither a prefix nor an extension of T'. That is, $T \not\preceq T'$ and $T' \not\preceq T$. Clearly, this implies also that $T \neq T'$. Because the two traces are different, and neither is a prefix of the other one, there must exists the *least time stamp* at which the two traces differ. We will denote this time by $\Delta(T, T')$:

DEFINITION 2.14. Let T and T' be two traces such that $T \not\preceq T'$ and $T' \not\preceq T$. We define then $\Delta(T, T')$ as

$$\Delta(T, T') = \min t^* : T|_{t^*} \neq T'|_{t^*}. \tag{2.8}$$

Example 2.15. Let $T = \langle\{\langle A, 1\rangle\}, 2\}\rangle$ and $T' = \langle\{\langle B, 0.8\rangle, 2\}\rangle$. Then $\Delta(T, T') = 0.8$. Namely, $T|_{0.8} = \tau$ but $T'|_{0.8} = B$. For all $t < 0.8$, $T|_t = T'|_t$.

© Antti Huima 2004. All rights reserved.

"Lingua Franca is a pidgin, a trade language used by numerous language communities around the Mediterranean, to communicate with others whose language they did not speak."
–Alan D. Corré

2.2 Specifications and Implementations

Traces are the *lingua franca* for discussing the behavior of systems. This is the reason we we started laying out our formal foundations exactly from the trace concept. We will now move on to consider specifications, which will denote sets of traces, and implementations, which will generate traces as an exhibit of their behaviors.

We shall start with specifications.

2.2.1 Specifications

We now turn our attention to operational specifications, which I will call from now on just "specifications" unless grave misunderstandings were bound to occur.

A specification specifies the "correct behavior" of a system. Correct behavior, however, is not a fixed, single, linear sequence of operations that a system must perform, but rather a collection of different execution sequences and subsequences that are valid responses or reactions to different *stimuli*. Therefore, in general, the correct behavior of a system can not be expressed as a single trace, but it *can* be expressed as a *set of traces*.

Thus, we take the view that a specification denotes (usually implicitly) a *set of traces*, which is the set of all those traces that are "correct". That is, whenever a system executes, it executes correctly with respect to a specification if and only if the trace generated by the execution is found from the set of traces the specification denotes.

Because the set of correct traces denote the set of those linear behaviors that are "correct", this set must be prefix-complete. This means that if a certain trace T belongs to the set, then also all the prefixes of that trace must belong to the set. The reason is that these prefixes contain strictly less information than the trace T itself. Because T is not faulty, there is nothing wrong with T but it is correct with respect to the specification, so must all its prefixes be correct.

Furthermore, we shall require that every valid trace has at least one extension into a valid, longer trace, arbitrarily much further in time. Intuitively, this means that if a trace that ends at time t is valid, then there must be *some* valid way to continue execution from time t further. In yet other words, time must continue forever.[1]

Finally, the empty trace ϵ must be valid. This and the previous requirement together imply that a specification must specify at least one infinitely long, valid behavior.

Let us further assume that there is an *a priori* given set of "potential specifications". This could be, for example, the set of all valid English documents, or the set of all syntactically correct UML diagrams. Its concrete structure is not relevant here. We denote this set by $\mathbb{S}$.

DEFINITION 2.16. $\mathbb{S}$ is a countable set of specifications.

For every $S \in \mathbb{S}$, $\mathbf{Tr}(S)$ denotes a set of traces. This is the set of *valid traces for* S.

The set $\mathbf{Tr}(S)$ must contain ϵ, the empty trace.

The set $\mathbf{Tr}(S)$ must be prefix-complete. In other words, $T \in \mathbf{Tr}(S)$ and $T' \prec T$ must always imply that $T' \in \mathbf{Tr}(S)$.

For every trace $\langle E, t\rangle \in \mathbf{Tr}(S)$, it must hold that for every arbitrary constant $\delta > 0$, there exists at least one set E' such that $\langle E', t+\delta\rangle \in \mathbf{Tr}(S)$ and $\langle E, t\rangle \prec \langle E', t+\delta\rangle$ (seriality requirement).

Let us try to illustrate these ideas graphically. Consider Fig. 2.1. This is a rough sketch of a prefix-closed set of traces. Time flows in the figure horizontally. Every path from left to right is a prefix-closed set of traces, or an "infinitely long trace", if you wish, although we have not defined such a concept. (The paths are assumed to continue "outside" the diagram *ad infinitum*.)

We can draw a line vertically across the figure as is shown. Assume this corresponds to three seconds since beginning. The paths from $t = 0$ up to the vertical line now correspond to traces whose end time stamp is 3.

One path has been drawn as a dashed line; this is the bottommost path that ends with a cross. Cutting it at $t = 3$ yields a valid trace. However, the path ends eventually at $t = 5$. This means that the prefix-closed set depicted by the figure contains a trace of the form $\langle E, 5\rangle$ such that there is no extension of it in the set, i.e. no trace of the form $\langle E', 5+\delta\rangle$ for $\delta > 0$ such that $\langle E, 5\rangle \prec \langle E', 5+\delta\rangle$.

[1]This definition implicitly means that all systems with Zeno behaviors must be invalid. See also the footnote on page 12.

© Antti Huima 2004. All rights reserved.

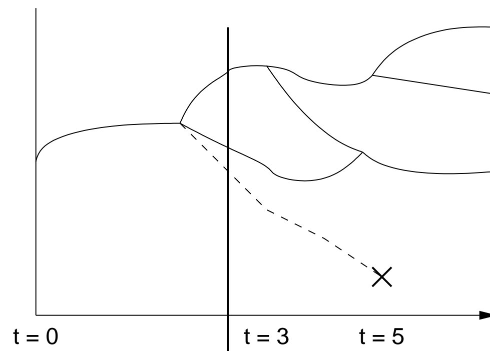

Figure 2.1: A sketch of a prefix-closed set of traces.

Note that this depicted set of traces contains the empty trace and is prefix-complete by the graphical construction; what is different from the trace sets of correctly formed specifications is that there exists at least one trace that can not be extended infinitely.

We can now, hopefully, understand the logical difficulty that would arise if we would allow sets like this as the meaning of specifications. Namely, assume that the depicted set of traces would be the set of traces denoted by a specifications. There could exist a system that would execute, against some environment, on the dashed path. At $t = 3$ the system would have generated a trace that would be found from the set of valid traces. However, it would be already known that after two seconds the system will generate a trace that is invalid. Hence, the system is already known to be on an invalid execution path — we can not stop the flow of time!

One could ask, would there be a problem if it could be guaranteed that the executing system would be "stopped" before $t = 5$. We shall take the viewpoint that there is no such a thing as a system stop that stops time. A system that is stopped is a system that does not accept any input any more and sends no output any more. It still exists in the time; the trace extensions for a stopped system just happen to be void of any events.

So, back to the question, would the system work correctly or incorrectly at $t = 3$ (suppose we stop testing at that point of time). We could argue: correctly, because the generated trace is part of the set of valid traces. We could also argue even more convincingly: incorrectly, because it is known with 100% certainty that the system will produce a failing trace after two seconds, *no matter what*.

This is a paradox that we are better off avoiding.

2.2.2 Implementations

The next aspect to consider is what is an SUT, and how SUTs produce traces. Let us assume that there is in general a set of possible systems or possible implementations, and let us denote this set by $\mathbb{I}$. This set could be, for instance, the set of all syntactically valid C or JAVA programs that are capable of handling the I/O alphabet Σ, whatever it happens to concretely contain. (It could contain, for example, all bit strings, or all valid CORBA messages for a certain IDL API).

DEFINITION 2.17. $\mathbb{I}$ is a countable set of implementations.

The main feature of an implementation is, certainly, that it can be executed, and that it can exhibit behavior. We will return to this shortly. Before that, let us consider for a while what is the relationship between a specification, and implementations that "try to implement it".

2.2.3 Failure Models

The main problem in black box testing, which formal conformance testing is also about, is that we do not know the internal structure of the system. In other words, it is not possible even in theory to deduce information about the correctness of the system by inspecting it "directly". This is exactly what the phrase "black box" means here.

On the other hand, in pragmatic black box testing testing engineers make all kinds of explicit and implicit assumptions about the structure of the system. It is known, for example,

© Antti Huima 2004. All rights reserved.

that bugs are "likely" to occur near the boundaries of valid ranges for certain inputs. This is well known. The idea of focusing testing in the boundary values is known as the Boundary Value Pattern.

The Boundary Value Pattern is basically about the following hypothesis: "programmers tend to make mistakes in handling boundary values". This can be translated into a slightly more rigorous form as: "it is more probable that a bug appears at a boundary value than at a non-boundary value". This is a probabilistic assumption about the, in general unknown, black box SUT. Note that the assumption is related to the specification, because the *value boundaries come from the specification*. Therefore, an assumption like this about the implementation of the SUT is (1) probabilistic in nature and (2) linked to the specification.

A model or assumption of how a system might fail is traditionally called a *failure model*. We can well follow this terminology, although we shall generalize the idea: we will consider models of how a particular specification might be implemented, regardless of whether the implementations are correct or incorrect ones.

The current discussion can be summarized as follows: a failure model maps specifications to discrete probability distributions over implementations. Let us formalize this.

DEFINITION 2.18. A *failure model* is a function $\mu : \mathbb{S} \to (\mathbb{I} \to [0, 1])$ such that for every $s \in \mathbb{S}$,

$$\sum_i \mu(s)[i] = 1. \tag{2.9}$$

A failure models gives directly an *a priori* estimate on the correctness of the SUT — *before* it has been tested. We will come back to this point once we have defined more exactly what is a correct, and what is an incorrect SUT.

2.3 Execution

2.3.1 Testing Strategies

Every implementation has some form of a behavior. The behavior is manifested in traces that the implementation produces. However, usually a system does not work in isolation, but within an environment. The environment here is a tester which interacts with the particular system. We can not know the set of traces a system can produce before we know how it is being interacted with.

Now the subject of this interaction is the tester process. However, the tester process also contains the aspect of giving a test verdict, but the test verdict in itself does not affect the execution of the system under test. Therefore we isolate the aspect of interaction, and call a particular way of interacting with a black box a *testing strategy*. We denote the set of all possible testing strategies with $\mathbb{T}$, but do not impose (yet) any further structure on this set.

DEFINITION 2.19. $\mathbb{T}$ is a countable set of testing strategies.

2.3.2 Generated Traces

Semantic function is a function that defines semantics, for example, execution semantics. "Semantic" does not characterize here the structure of a function, but its intended use.

A testing strategy and an implementation (against which the testing strategy is executed) together determine behavior at the testing interface. However, there is not necessarily a single trace that the combined system produces, but there can by many potential traces that could be generated. We formalize this in terms of ξ, a semantic function that maps testing strategies and implementations eventually on discrete probability distributions of traces.

We are going to see more complex definitions than until now, so it is useful to spent some time on informal discussion first.

We shall assume that a long interaction with an SUT does not happen once as an atomic operation, but that it can be decomposed into steps which we will call in general *test steps*. A test step must consume a strictly positive amount of time, albeit arbitrarily small (for example 10^{-8} seconds). During a test step, zero or more I/O steps can occur. For example, during a test step there could be no I/O, or one hundred messages could be exchanged. So a single test step in general does not correspond to a single I/O event.

Thus, we need a function that tells us how test steps will follow one another, when a testing strategy and an implementation have been fixed. We formulate this so, that the execution function ξ gives a discrete probability distribution for the next single test steps in a given "state of the system". This "state of the system" corresponds to the observed trace this far; if

© Antti Huima 2004. All rights reserved.

there is any other unknown internal state in the SUT that affects how the execution continues, it is modeled (as is appropriate) in the probability densities. The test steps in practice are trace extensions for the current trace history.

We will impose some important restrictions on the function ξ, in addition to the basic principle that it must return a valid probability distribution. The other restrictions are the following.

1. All sequences of test steps must be *progressive*. In practice this means that the end time stamps of all those traces whose probability is greater than zero must grow eventually arbitrarily large. What the progressivity requirement excludes is an execution function that produces test steps whose temporal lengths (= consumed time) form a converging series. This would mean than an infinite number of test steps could be carried out in a finite time, which is absurd from a practical point of view.[2]

   For example, test steps could produce a series of traces with end time stamps from the series $1 - 2^{-n}$. If the first test step would last from $t = 0$ to $t = \frac{1}{2}$, the second would last from $\frac{1}{2}$ to $\frac{3}{4}$, the third from $\frac{3}{4}$ to $\frac{7}{8}$ and so on. Every test step would have a strictly positive time span. Still, infinitely many test steps would be taken before $t = 1$. In addition to causing complications in the formal theory, this would be also patently absurd in the real world.

2. If two trace extensions have non-zero probabilities for a given already-executed trace, implementation and testing strategy, then neither of the extensions is a (proper) prefix of the other one. This basically means that any two possible continued executions (test steps) that are distinct must differ in I/O events, and do so at a moment of time that is present in both of the extended traces.

   This restriction has a different flavor than the first one. The first restriction excludes physically impossible scenarios that cause all kinds of havoc for formal analysis. The current (second) restriction makes definitions easier, but has in principle no pragmatic meaning. From some point of view it is sensible to consider a testing strategy that chooses randomly between "testing steps" of variable temporal span (= length). But this possibility is in no way excluded by this restriction, because the "testing steps" here do not need to coincide with the formal test steps that we are defining.

We shall now attempt a formal definition. Let $\mathscr{T}$ denote the set of all traces.

DEFINITION 2.20. ξ is a function

$$\xi : \mathbb{I} \times \mathbb{T} \times \mathscr{T} \to (\mathscr{T} \to [0,1]) \tag{2.10}$$

with the following properties.

The function gives a discrete probability distribution:

$$\sum_{T'} \xi(i,s,T)[T'] = 1. \tag{2.11}$$

Traces with non-zero probability are proper extensions of the previous trace: For all $i \in \mathbb{I}, s \in \mathbb{T}, T, T' \in \mathscr{T}$ it holds that

$$\xi(i,s,T)[T'] > 0 \implies T \prec T'. \tag{2.12}$$

All test step sequences are progressive: There does not exist an infinite sequence of traces $T_0, T_1, T_2, \ldots$ and a constant $K \in \mathbb{R}$ such that $\xi(i,s,T_i)[T_{i+1}] > 0$ for all $i \geq 0$ but such that for all $T_i = \langle E_i, t_i \rangle$, it holds that $t_i < K$.

A potential trace extension is not a prefix of another potential trace extension:

$$\xi(i,s,T)[T_1] > 0 \land \xi(i,s,T)[T_2] > 0 \land T_1 \neq T_2 \implies T_1 \not\prec T_2 \land T_2 \not\prec T_1. \tag{2.13}$$

[2] The progressivity of test steps indirectly also eliminates the possibility of what is known as *Zeno behavior*. The reason is that every trace can contain only a finite number of events, because we have *defined* that traces must be finite in this sense. The remaining details can be reconstructed by an interested reader. If you do not know what Zeno behavior *is*, you can safely ignore this subject here. Not to say it would not be important in general.

© Antti Huima 2004. All rights reserved.

2.3.3 Trace Probabilities

Now that we have defined execution via this function ξ that gives us the probabilities for single test steps, it is natural to ask what is the "composed" probability for a trace that is generated by multiple test steps. We expect this to be the product of the individual transition probabilities, which is correct in principle. But what about traces that do not appear "in between" test steps, but that are terminated "during" a test step? We can and must take these into account, but this requires an extra technicality.

Let us state the problem of this composed probability in clear terms. Suppose that we repeat many times the following experiment. Starting from an empty trace, we execute a testing strategy against an implementation until the end time stamp of the trace generated (i.e. observed) is over a constant K. That is, a trace $\langle E, t\rangle$ has been produced such that $t > K$. What is the probability that T, a given trace with end time stamp K, is a prefix of this trace? In other words, how likely is that we will observe a trace that begins with T on any particular trial?

Let us denote this sought-for probability by $P[i, s, T]$. (i is the implementation, s the specification.)

As an intermediary technical device, let us denote by $P^*[i, s, T]$ the maximum over all finite sequences $\epsilon = T_1, \ldots, T_k = T$ of the following expression:

$$\prod_{i \in [1,k-1]} \xi(i, s, T_i)[T_{i+1}]. \tag{2.14}$$

Intuitively, if this expression yields something else than zero, then it gives the probability for producing trace T by the execution process that I explained above. Given this intuition, we would expect the value to be unique. This can be stated clearly:

Proposition 2.21. *For any trace T, there exists* at most *one sequence of traces for which expression* (2.14) *yields something else than zero.*

Proof. For the sake of contradiction, assume T to be such a trace such that there exist two distinct sequences $X_1, \ldots, X_k$ and $Y_1, \ldots, Y_n$ such that $X_1 = \epsilon$, $Y_1 = \epsilon$, $X_k = T$, $Y_n = T$, and such that

$$\prod_{i \in [1,k-1]} \xi(i, s, X_i)[X_{i+1}] > 0 \tag{2.15}$$

and

$$\prod_{i \in [1,n-1]} \xi(i, s, Y_i)[Y_{i+1}] > 0. \tag{2.16}$$

Because the X-sequence and the Y-sequence are distinct, there must exist a $\ell \le k, n$ such that for all $1 \le i < \ell$, $X_i = Y_i$, but such that $X_\ell \ne Y_\ell$. Now both X_ℓ and Y_ℓ must be prefixes of T. Hence, there are two distinct test steps from $X_{\ell-1}$ with non-zero probabilities such that one of the extended trace is a prefix of the other. This is against the definition of ξ. Thus we have reached a contradiction. □

Now the problem that remains is that there can be traces that, by the same intuition, can be expected to have non-zero probabilities of occurrence, but P^* is still zero for them. These are traces that occur "during" test steps. For example, suppose

$$\xi(i, s, \epsilon)[T] = 1 \tag{2.17}$$

and that $\epsilon \prec T' \prec T$. Clearly, we would expect $P[i, s, T'] = 1$ also. But $P^*[i, s, T'] = 0$ (this is easy to verify).

The idea in general is to find the longest prefix for a trace T that has a probability defined by P^*, and then derive the probability for T from that. The following proposition is needed:

It is standard nomenclature to speak of "a maximal X" in a context like this. But maximal according to what? The ordering can be left unmentioned if it is implied and there is no danger of confusion. Here, naturally, the prefix should be maximal in the ordering $\preceq$, i.e. temporally longest possible.

Proposition 2.22. *For any trace T, there exists a maximal prefix of T, which we will denote by $\tilde{T}$, that has the property that $P^*[i, s, \tilde{T}] > 0$.*

Proof. There exists at least one such prefix, namely ϵ, for which $P^*[i, s, \epsilon] = 1$. Hence, it is enough to show that the set of prefixes T' of T for which $P^*[i, s, T'] > 0$ is finite. But this stems directly from the requirement that test steps are progressive, because if the set of such prefixes would be infinite, and infinitely many test steps could be produced before reaching T, i.e. in finite time. □

© Antti Huima 2004. All rights reserved.

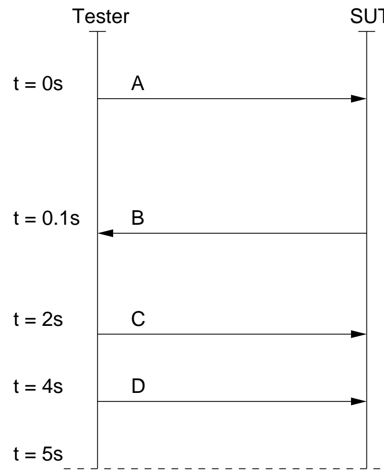

Figure 2.2: A trace (copy of Fig. 1.3).

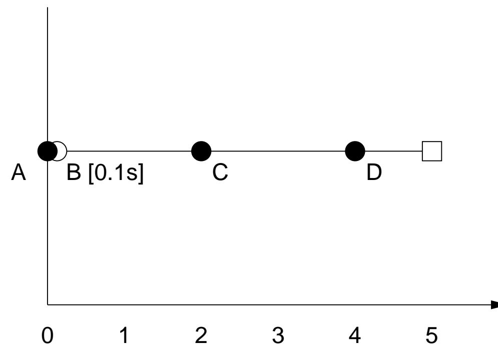

Figure 2.3: An alternative representation of the trace in Fig. 2.2.

Intuitively, this states that when we go through the prefixes of T, starting from the longest prefixes and moving toward the shorter ones, we will find a prefix with non-zero probability. If nothing else, then the prefix ϵ, for which $P^*[i, s, \epsilon] = 1$ regardless of i and s.

Let thus $\tilde{T}$ be the maximum (proper or trivial) prefix of T with non-zero $P^*[i, s, \tilde{T}]$ as in the proposition above. Then the probability of the trace T is given by

$$P[i, s, T] = P^*[i, s, \tilde{T}] \times \left(\sum_{T' : T \preceq T'} \xi(i, s, \tilde{T})[T'] \right). \tag{2.18}$$

The idea here is the following: every test step from $\tilde{T}$ will produce a trace that is no longer a prefix of T. This stems from the fact that $\tilde{T}$ was chosen a the maximal prefix of T that has a probability defined by P^*, i.e. that is the "result" of a test step. We go through the next test steps from $\tilde{T}$, and add together the probabilities of those test steps that produce extensions of the trace T. This sum is multiplied by the known probability of $\tilde{T}$, and we are done.

As a sanity check, note that if $T = \tilde{T}$, the sum in (2.18) always yields 1. Hence, in this case, $P[i, s, T] = P^*[i, s, T]$.

On the other hand, if T cannot be produced by a testing strategy and an implementation, it should be that $P[i, s, T] = 0$. Because $\tilde{T}$ is bound to have a non-zero probability (recall that $\tilde{T}$ could be ϵ), the sum in (2.18) must yield zero. This means that after the longest prefix $\tilde{T}$, every possible test step will produce something that differs from T. This is, again, correct.

© Antti Huima 2004. All rights reserved.

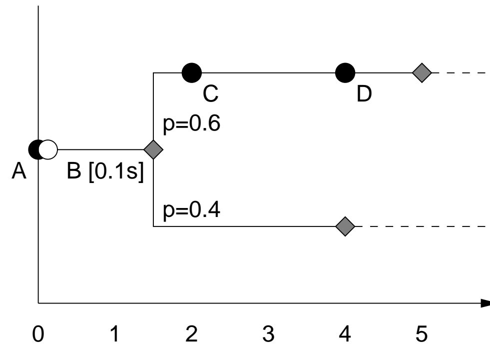

Figure 2.4: Graphical representation of execution.

2.3.4 Graphical Sketches and Examples

Let us attempt to concretize these ideas. In Fig. 2.2 I have reproduced an example trace from the first chapter. This trace is, in the formal framework, the following mathematical object:

$$\langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle, \langle C, 2 \rangle, \langle D, 4 \rangle \}, 5 \rangle. \tag{2.19}$$

(Make sure that you understand this notation before you proceed further.)

We could invent an alternative graphical representation for this trace along the lines in Fig. 2.3. Here, I have used solid (black) circles to denote inputs to SUT, and hollow (white) circles to denote outputs. Time scale is now linear and plotted in the horizontal direction. The hollow box denotes the end of the trace. Note that the trace of Fig. 2.2, the alternative drawing in Fig. 2.3, and the expression in (2.19) all attempt to denote the same mathematical object.

Assume now that we have a particular implementation i and a testing strategy s, and that we want to illustrate how a given execution function ξ "works" for i and s. We could do this as in Fig. 2.4. This figure attempts to illustrate three individual test steps. Let us go through the figure in detail.

The gray diamonds denote test step boundaries. When the execution begins, there is a test step with probability 1 that takes 1.5 seconds of time, during which A and B are exchanged at the moments indicated by the horizontal time axis. Formally, we write this down as:

$$\xi(i, s, \epsilon)[\langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle \}, 1.5 \rangle] = 1 \tag{2.20}$$

This equation says that the probability for the trace

$$\langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle \}, 1.5 \rangle \tag{2.21}$$

which is exactly the trace up to the first gray diamond in the current figure is 1, when execution continues after the empty trace (ϵ), i.e. actually starts at the beginning.

After the first gray diamond there is a branch. The upper branch has been indicated to have the probability of 0.6. This can be written in the mathematical notation as:

$$\xi(i, s, \langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle \}, 1.5 \rangle)[\langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle, \langle C, 2 \rangle, \langle D, 4 \rangle \}, 5 \rangle] = 0.6 \tag{2.22}$$

This says that after the trace (2.21), the probability for the extended trace

$$\langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle, \langle C, 2 \rangle, \langle D, 4 \rangle \}, 5 \rangle \tag{2.23}$$

is 0.6, the same probability that is indicated in the figure. (You should ascertain yourself that (2.21) is really a proper prefix of (2.23).)

The lower branch, similarly, can be written down mathematically as

$$\xi(i, s, \langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle \}, 1.5 \rangle)[\langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle \}, 4 \rangle] = 0.4 \tag{2.24}$$

Implicitly, then, for all *other* traces T,

$$\xi(i, s, \langle \{ \langle A, 0 \rangle, \langle B, 0.1 \rangle \}, 1.5 \rangle)[T] = 0. \tag{2.25}$$

This is mandated by (2.11).

The probability for the trace (2.23) is 0.6. This means that if we produce a long enough trace by executing s against i, then the probability that (2.23) is its prefix is exactly 0.6

© Antti Huima 2004. All rights reserved.

2.3.5 Use of Test Steps

Let us now stop for a moment of reflection. It seems that there are at least three different ways to measure the "length" or "size" of a trace, when the trace has been generated in the context of the execution of a testing strategy against an implementation.

First, the *temporal length* of a trace, which is the time difference between the end time stamp and the clock reading zero, meaningfully characterizes a size of the trace. For example, the temporal length of the trace (2.23) is 5.

Secondly, the *number of events* in a trace somehow characterizes the "weight" or "complexity" of the trace. For example, the number of I/O events in the trace (2.23) is 4.

Third, the *number of test steps* that were required to produce a trace gives a measure on the size or length of the trace. For example, the number of test steps required to create the trace (2.23) in the context of the preceding example is 2.

What is the use of all these distinct measures? Do they have a use? Why have we created the concept of test steps? What is the use for test steps? Let me try to answer these questions next.

The first thing to say is that the number of events in a trace is almost completely *immaterial*. Often people assume otherwise, instinctively. But given a sound theory of timed execution, silence becomes as important as non-silence, transmission as important as non-transmission. So the only thing that really interests us in the number of events is that the number of events must be finite. Otherwise, the count does not matter much.

The temporal length of a trace is important, mostly because being temporally shorter than another trace is a prerequisite for being a prefix of the other one. But note carefully that we do not want to link the temporal length of a trace (i.e. the amount of time the trace "consumes" in the real world) with any sort of *cost* of the trace, because we do not know the cost of time, and any such cost could be a nonlinear function of time spent.

At this point it might be guessable that we will link the number of test steps with the *cost of testing*. So, one use of the test steps will be that we shall assume that every test step costs a constant amount of... whatever. Because the amount is constant, we can dispense with units and just call it the "unit cost", or, more compactly, "1".

Of course, test steps have been also introduced as a technical device for defining probabilities for arbitrarily long traces that result from execution (because of the definition of a trace, every single test step spans only a finite time interval).

2.4 Correctness of Behavior

We know now how a testing strategy and an implementation together create behavior. Namely, to recapitulate the previous discussion, we assume that there exists a function ξ that describes this, in terms of trace extension probabilities. This is not begging the question, even if it would seem to be. We really *can not* give a more detailed description of ξ, because we have not fixed yet what the specifications and testing strategies actually are. The properties of ξ and the related definitions form a framework where concrete execution semantics can be later embedded.

So we have behavior. But the crucial question from the testing point of view is: is the behavior *correct*? Let us now move on to consider this topic.

2.4.1 Correct Behavior

A particular behavior, i.e. a trace that has been observed, is not correct or incorrect in isolation. Even an operating system crash could be correct behavior for the operating system, because it might have been a deliberate decision not to protect an operating system from crashing in the situation of, say, a completely buggy device driver being injected into the kernel space. The correctness or incorrectness of a behavior is always *with respect to a specification*.

We have already assumed that every specification can be mapped to a (prefix-complete) set of *valid traces*. This gives us a straightforward way to define what *is* correct behavior with respect to a specification:

DEFINITION 2.23. Let T be a trace and S a specification, i.e. a member of the set $\mathbb{S}$. The trace T is *correct with respect to S* if and only if $T \in \mathbf{Tr}(S)$.

(Recall from Def. 2.16 that $\mathbf{Tr}(S)$ is the set of traces associated with S.)

© Antti Huima 2004. All rights reserved.

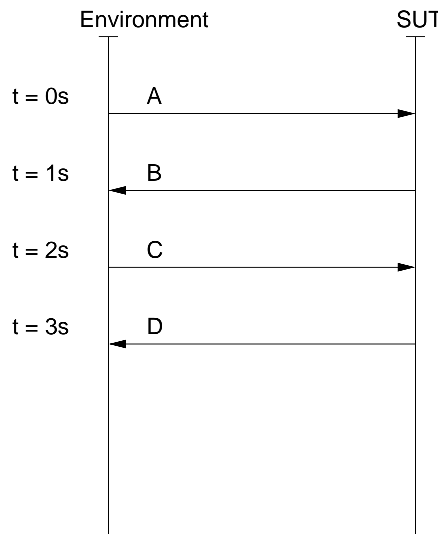

Figure 2.5: A trace from a specification.

2.4.2 Types of Incorrect Behavior

Incorrect behavior is behavior that is not correct. Is it so simple?

No. The reason is that, as it will turn out, there are *two* agents that can produce incorrect behavior in a testing setup. The first is the SUT. The second is the tester, actually the testing strategy.

Clearly the SUT could produce incorrect behavior. After all, this is the reason why we test. But how can the *tester* produce incorrect behavior? The answer is simple. It could *produce illegal input* — which includes *not* producing input at all when it would be required.

The question becomes: what is illegal input? Do specifications need to mention those inputs that are illegal? But the answer is deceptively simple: the specifications *already* mention all illegal inputs and lacks of inputs. Namely, the definition of a trace is completely symmetric with respect to inputs and outputs, and a specification specifies correct behavior in terms of a set of traces. Therefore, in the same way as a set of traces describes potential failures of a SUT, it describes potential failures of a tester. Let us make this concrete through an example.

Consider the trace in Fig. 2.5. Assume that we have a specification S such that *every* trace in $\mathbf{Tr}(S)$ is either a prefix of this trace, or then this trace is a prefix of it. This means that the only correct way to work with a black box that falls under this specification is to exchange the messages A, B, C and D at exactly those times that are shown in the figure.

Assume that we produce during testing the following traces:

$$T_1 = \langle \{ \langle A, 0 \rangle, \langle Z, 1 \rangle \}, 1.5 \rangle \tag{2.26}$$
$$T_2 = \langle \{ \langle A, 0.5 \rangle, \langle Z, 1 \rangle \}, 1.5 \rangle. \tag{2.27}$$

Neither T_1 nor T_2 belongs to the set $\mathbf{Tr}(S)$. Hence, both of them correspond to incorrect behavior. But intuition tells us (hopefully) that T_1 is somehow “a problem in the SUT”, where as the trace T_2 is “a bug in the tester”, because the tester does not send A to the SUT at time zero as required by the specification. Clearly, there is nothing the SUT could do better to remedy this problem, because the tester is outside its control!

Before we attempt a more rigorous treatment, consider the (another) trace in Fig. 2.6. Assume that the specification S contains also all the prefixes and suffixes of this different trace. Assume further that we produce during testing the following trace:

$$T_3 = \langle \{ \langle A, 0 \rangle \}, 1.5 \rangle. \tag{2.28}$$

First note that this trace T_3 is not correct with respect to the modified specification, because at time 1 there should be either the transmission of A from the environment (= the tester) to the SUT, *or* the transmission of B from the SUT to the environment. But if this behavior has been produced during testing, *whose fault is it?* You should be able to see that either the SUT or

© Antti Huima 2004. All rights reserved.

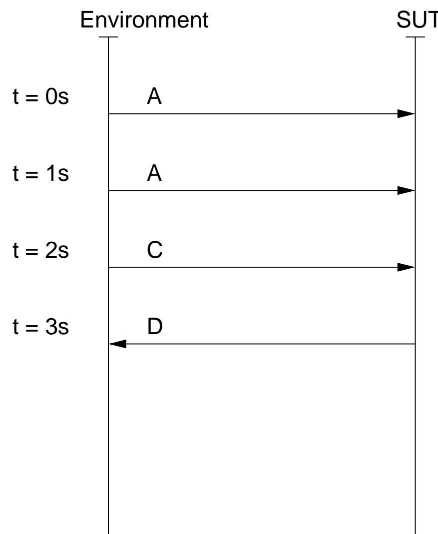

Figure 2.6: Another trace from a specification.

the tester could have remedied the problem by doing something at time 1, and it is impossible to put the blame on either one of the two communicating subjects alone.

This informal discussion, based on an example, suggest that we can divide failures into three categories: (1) Failures of the SUT. (2) Failures of the tester. (3) Strange, border case failures where the scapegoat is not well defined at all.

It must be noted that it is not uncommon for specifications to exclude certain inputs in certain situations. There are many reasons to do so.

A convincing example from the physical world is the following. There is a button in an elevator. Interactions with the button are modeled as two messages that can be transmitted from the environment to the elevator: ”button down” and ”button up”. Because it is physically impossible to put the button down twice without it popping up between, a tester that would somehow anyway produce a sequence of two ”button downs” without any ”button up” between would be doing something that could reasonably be expected to be excluded from a specification. So the tester would be erroneous. (This example is, to my best knowledge, originally due to professor Antti Valmari.)

Another similar example comes from modeling synchronous calls to a single-threaded JAVA API as one request message, denoting the start of the call, and an answer message, denoting the return of the call. Because the single-threadedness of the API prohibits two simultaneous calls, a tester attempting to initiate two calls concurrently (sending two request messages without waiting for an answer message) would work incorrectly.

There can be also, in general, contracts in interfaces and APIs that prohibit certain operations in certain conditions. For example, the UNIX manual for the C standard library call `free` states:

> free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if free(ptr) has already been called before, *undefined behavior occurs*. If ptr is NULL, no operation is performed.

This means that for a tester it would be illegal to attempt to `free` twice a memory block, because it has been stated that the system does not need to be able to cope with it (this is the essence of “undefined” behavior).

2.4.3 Verdicts

We move now on to formally classify incorrect traces into the three groups just mentioned. This is a good point to introduce the *verdicts*, so let us do so first, however.

A *verdict* is a testing result. It is a judgment on an executed test run. An executed test run is a trace. So a verdict is a judgment on a trace, and it is based on a specification. We will use the following verdicts:

© Antti Huima 2004. All rights reserved.

PASS. The PASS verdict means that the test run contains nothing that would imply that the SUT works incorrectly. Furthermore, the tester has also worked correctly, i.e. there have been on bad inputs or other out-of-specification behavior by the tester. If all test attempts against an SUT result in verdict PASS, project managers are often happy.

FAIL. The FAIL verdict means that a test run has been produced that implies that the SUT works incorrectly. At least until the first deviation of the SUT from the specification, the tester has played by the rules. The FAIL verdict means, thus, that the system is incorrect. It does not conform to its specification.

ERROR. The ERROR verdict means that the tester has deviated from the specification before the SUT has done anything against the specification. The test run has become void; it has no meaning, because inputs to the SUT have escaped the realm of the specification. In particular, the ERROR does not imply in any way that there would be an error in the SUT.

INCONCLUSIVE. The INCONCLUSIVE verdict is a PASS verdict with an extra connotation: that there was some feature to be tested or some other testing goal to be met during the test run, which was not met successfully. In the same sense as PASS and FAIL can be given only with respect to a specification, the INCONCLUSIVE verdict is only possible if there is an additional *test purpose*. We will return to test purposes only later, so we will not see the INCONCLUSIVE verdict actually for a while.

As a matter of fact, the condition of not meeting a test purpose is orthogonal to division between the three main verdicts (PASS, FAIL, ERROR). Therefore we could envision having verdicts like PASS-INCONCLUSIVE, FAIL-INCONCLUSIVE and ERROR-INCONCLUSIVE. The reason that we do not mind to make a distinction between the hypothetical verdicts FAIL-INCONCLUSIVE and FAIL-CONCLUSIVE is that the verdict FAIL means that the SUT is *in any case* incorrect, regardless of any test purposes whatsoever, and that is usually a big problem. The distinction between meeting or not meeting a test purpose is practically important only when the SUT works correctly. Furthermore, after some thinking it may dawn that the hypothetical verdict FAIL-CONCLUSIVE could be a very rare beast indeed.

CONFUSED. The CONFUSED verdict is not a standard one. We reserve this verdict to denote a case where an observed test run is erroneous, but the specification cannot be used to determine whether the error has occurred on the SUT side or on the tester side. In practice, we would like to consider those specifications as erroneous that can produce CONFUSED as a testing result, so this is a verdict should not pop out in practice. For the case in which this verdict is given, see Sec. 2.4.5.

We let $\mathbb{V}$ denote the set of the four verdicts above:

DEFINITION 2.24. We denote by the set $\mathbb{V}$ the set $\{$PASS, FAIL, ERROR, INCONC, CONF$\}$.

2.4.4 Failure Types

We have already defined that a trace T is correct with respect to a specification S if and only if $T \in \mathbf{Tr}(S)$. Therefore, let us now focus on the case $T \notin \mathbf{Tr}(S)$, i.e. to the case where the trace T is in general incorrect and out of the specification S.

Suppose $T = \langle E, t \rangle$. Let V be the intersection $\mathbf{Pfx}(T) \cap \mathbf{Tr}(S)$, i.e. let it be the set of valid prefixes of T, and let K be the set of end time stamps in this set, i.e.

$$K = \{t \mid \exists E : \langle E, t \rangle \in V\}. \tag{2.29}$$

(Note that this set can, in principle, be either open or closed. In any case, it is a connected interval because V itself is prefix-closed.)

For every $\delta \in K$, denote by X_δ the set of every such a valid trace that extends a δ-prefix of T and whose end time stamp is past t. That is,

$$X_\delta = \{\langle E', t' \rangle \mid T[\delta] \preceq \langle E', t' \rangle \wedge \langle E', t' \rangle \in \mathbf{Tr}(S) \wedge t' > t\}. \tag{2.30}$$

Note that for any $\delta \in K$, $T[\delta] \in \mathbf{Tr}(S)$ by construction. Furthermore, note that because T is invalid, and every trace in X_δ for any suitable δ is valid, T as a whole can not be a prefix of any element of X_δ for any δ. This stems from the fact that $\mathbf{Tr}(S)$ is prefix-complete. On the other hand, because the end time stamp of every trace in X_δ is past that of T, an element of X_δ can not be a prefix of T. Hence, for any $U \in X_\delta$, $\Delta(T, U)$ must be defined.

Therefore, denote by D_δ the set

$$\{\alpha \mid \alpha \in \Sigma \wedge \exists U \in X_\delta : (T|_{\Delta(T,U)} = \tau \wedge \alpha = U|_{\Delta(T,U)}) \vee (T|_{\Delta(T,U)} = \alpha)\}. \tag{2.31}$$

© Antti Huima 2004. All rights reserved.

This is, intuitively, the set of all possible I/O events that cause the first difference between any trace in X_δ and the trace T. However, the definition is not symmetric. If at the first time stamp where the trace T differs from a valid trace U, there is an event at the trace T, we count that event and ignore any event whatsoever in trace U. If at the trace T there is silence (τ), and there is an event in trace U, we count that event.

The idea is now to find a δ near the upper bound of K such that the set D_δ consists solely of input events, or solely of output events.

For instance, assume that we choose δ from K so that the set D_δ consists only of the elements of Σ_{in}. What does this situation tell us about the reason for why T was outside the specification, i.e. why T was invalid?

We know that $T[\delta]$ is a valid trace. Hence, there would have existed at least one valid continuation after $T[\delta]$. However, in the testing setup no valid continuation was chosen, but eventually, before time t, something illegal happened. Now every element of X_δ represents a valid continuation from $T[\delta]$. That is, every element of X_δ is a valid trace that is identical with T at least until time stamp δ, but that differs from T before time stamp t. Now we assumed that $D_\delta \in \Sigma_{\text{in}}$. Thus, *every valid extension of $T[\delta]$ differs from T initially by either the removal, change, or addition of an input event*. These all changes are ones that only the tester can effect. Thus, after time δ, only the tester could have changed the course of the execution so that it would have been conforming to the specification. Hence, the fact that T was invalid was caused by the tester.

Similarly, if we find a δ near from K so that $D_\delta \subseteq \Sigma_{\text{out}}$, then the only instance that could change the course of execution to a conforming one after time stamp δ is the SUT. In this case, the SUT is the cause for the failure.

We can summarize this as follows.

DEFINITION 2.25. Let S be a specification and $T = \langle E, t\rangle$ a trace such that $T \notin \mathbf{Tr}(S)$. Let K be the set

$$\{t' \mid \exists E' : \langle E', t'\rangle \in \mathbf{Pfx}(T) \cap \mathbf{Tr}(S)\}. \tag{2.32}$$

For any $\delta \in K$ define X_δ as in (2.30) and D_δ as in (2.31). The trace T is an *input failure* if there exists $\delta \in K$ such that

$$D_\delta \subseteq \Sigma_{\text{in}}. \tag{2.33}$$

Similarly, the trace T is an *output failure* if there exists $\delta \in K$ such that

$$D_\delta \subseteq \Sigma_{\text{out}}. \tag{2.34}$$

If the trace T is neither an input nor an output failure, it is a *shared failure*.

Example 2.26. Let us consider an example. Let S be a specification such that

$$\mathbf{Tr}(S) = \bigcup \{\mathbf{Pfx}(\langle\{\langle A, t\rangle, \langle B, t+\delta\rangle\}, t+\delta+1\rangle) \mid t \in [0, \infty), \delta \in (0, 1]\}. \tag{2.35}$$

That is, any trace T is valid if either it is silent (no events), or if there is the input of A, and the trace ends not later than one seconds since the time stamp of A with no other events, or if A is followed by B not later than one second after A. Let $A \in \Sigma_{\text{in}}$ and $B \in \Sigma_{\text{out}}$.

For instance,

$$\langle\{\langle A, 4\rangle, \langle B, 4.2\rangle\}, 9\rangle \tag{2.36}$$

is a valid trace but

$$\langle\{\langle A, 4\rangle, \langle B, 6.2\rangle\}, 9\rangle \tag{2.37}$$

is not, neither is

$$T = \langle\{\langle B, 4\rangle\}, 5\rangle. \tag{2.38}$$

Let us consider this trace T further. It is invalid. On the other hand, $T[4] = \langle\emptyset, 4\rangle$ is a valid trace. For any $t > 4$, $T[t]$ is invalid. Hence, in terms of the previous discussion, we have $K = [0, 4]$. We choose a δ from K as $\delta = 4$.

Let us to compute X_δ, i.e. X_4. This is the set

$$X_4 = \{\langle E, t\rangle \mid \langle E, t\rangle \in \mathbf{Tr}(S) \wedge t > 4 \wedge \langle E, t\rangle[4] = \langle\emptyset, 4\rangle\}. \tag{2.39}$$

Every element of X_4 is of one of the following forms:

$$\langle\emptyset, t\rangle \quad t > 4 \tag{2.40}$$

$$\langle\{\langle A, t'\rangle\}, t\rangle \quad t > 4, 4 \leq t' < t, t' \geq t - 1 \tag{2.41}$$

$$\langle\{\langle A, t'\rangle, \langle B, t''\rangle\}, t\rangle \quad t > 4, 4 \leq t' < t'' < t, t' \geq t'' - 1 \tag{2.42}$$

© Antti Huima 2004. All rights reserved.

Let U be of the form (2.40). Then $\Delta(U,T) = 4$, $U|_4 = \tau$ and $T|_4 = B$. Here we count B to the set D_4. For the second form, $\Delta(U,T)$ is again 4, in which case we count again B from $T|_4$. The same holds for the last form. Therefore, $D_4 = \{B\}$. Thus, the failure is an output failure: the SUT incorrectly outputs B before receiving stimulus A.

Example 2.27. At this point you almost certainly wonder on the complexity of the definitions which are used here to obtain the clearly obvious result in the previous example. You may have noticed that there is a hint of a limit construction. But where is the limit construction needed? The answer is: when the set K is open.

Let S be now a specification such that

$$\mathbf{Tr}(S) = \bigcup \{\mathbf{Pfx}(\langle\{\langle B,t\rangle\},t'\rangle) \mid t \in [0,1), t' > 1\}. \quad (2.43)$$

Intuitively, this specifies that the SUT must send the message B out *before* the time 1. Assume that we have at hand the trace

$$T = \langle\emptyset, 2\rangle. \quad (2.44)$$

Clearly $T \notin \mathbf{Tr}(S)$. Now $T[1]$ is not a valid trace, but for any $\epsilon > 0$, $T[1-\epsilon]$ is. Hence $K = [0,1)$.

We can choose δ from K e.g. as $\delta = 0.5$ (the choice does not matter). The valid extension for $T[\delta]$ all differ from T strictly before time stamp 1 by introducing the output B. Hence $D_{0.5} = \{B\}$, there has been an output failure. However, attempting to choose $\delta = 1$ (incorrectly, because $1 \notin K$), we would find out that there are no valid extensions of $T[1]$, because $T[1]$ is invalid!

Example 2.28. Yet, let us consider a specification S such that

$$\mathbf{Tr}(S) = \bigcup \{\mathbf{Pfx}(\langle\{\langle B,t\rangle\},t'\rangle) \mid t \in [0,1), t' > 1\} \cup \bigcup \{\mathbf{Pfx}(\langle\{\langle A,t\rangle\},t'\rangle) \mid t \in [0,1), t' > 1\}. \quad (2.45)$$

We assume, as before, $A \in \Sigma_{\text{in}}$ and $B \in \Sigma_{\text{out}}$. Consider again the trace

$$T = \langle\emptyset, 2\rangle. \quad (2.46)$$

for which it holds that $T \notin \mathbf{Tr}(S)$. Again, $K = [0,1)$.

Regardless of our choice of $\delta < 1$, there always exists two forms of valid extensions for $T[\delta]$: those, where the first event is A, and those, where the first event is B. This yields (you could verify this)

$$D_\delta = \{A, B\}. \quad (2.47)$$

Hence, we have a shared failure.

Intuitively, S specifies that either the tester or the SUT must send out a message before time 1. When neither of the entities sends a message, it is impossible to put blame solely on one of the two.

2.4.5 Calculating Verdicts

Now that we have been able to classify "low-level" failures into (1) input, (2) output, and (3) shared failures, we will link these failure types to the different testing verdicts.

DEFINITION 2.29. Let T be a trace and S a specification. The correct verdict for T is calculated as follows.

1. If $T \in \mathbf{Tr}(S)$, then the verdict is PASS.
2. Otherwise, if T is an output failure with respect to S, the verdict is FAIL.
3. Otherwise, if T is an input failure with respect to S, the verdict is ERROR.
4. The remaining case is that T is a shared failure with respect to S. The verdict is CONF.

The correct verdict for T with respect to S is denoted by $\mathbf{verdict}(T,S)$.

Note that the **verdict** function never returns INCONC. The reason is that no *test purposes* have been introduced yet. The verdict INCONC can be given only when there is a test purpose involved.

The different verdicts have been summarized in Table 2.1.

© Antti Huima 2004. All rights reserved.

Verdict	Description
PASS	SUT was tested correctly and it behaved correctly.
FAIL	SUT was tested correctly but it failed to produce correct behavior. There is a problem with the SUT.
ERROR	The SUT worked correctly until the tester began to test it in an incorrect fashion. There is a problem with the tester, because it puts the SUT into an environment that is outside the specification.
CONF	There is something strange going on, but the specification is ambiguous and it is impossible to pinpoint whether the tester or the SUT is working badly. The specification must be corrected.
INCONC	In principle PASS, but a test purpose has not been met.

Table 2.1: A summary of the different testing verdicts.

.

Exercises

Exercise 2.1. Consider the following set of traces:

$$T_1 = \langle \{\langle A, 1\rangle, \langle B, 2\rangle\}, 3\rangle \quad (2.48)$$
$$T_2 = \langle \{\langle A, 1\rangle, \langle B, 1.5\rangle\}, 2.5\rangle \quad (2.49)$$
$$T_3 = \langle \{\langle A, 1\rangle, \langle B, 2\rangle, \langle C, 3.5\rangle\}, 4\rangle \quad (2.50)$$
$$T_4 = \langle \{\langle A, 1\rangle, \langle B, 2\rangle\}, 4\rangle \quad (2.51)$$
$$T_5 = \langle \emptyset, 0\rangle \quad (2.52)$$
$$T_6 = \langle \emptyset, 1\rangle \quad (2.53)$$

Enumerate all pairs (i, k) such that $T_i \prec T_k$.

Exercise 2.2. Prove that the relation $\preceq$ is reflexive, transitive, and antisymmetric.

Exercise 2.3. Show that $T \preceq T'$ implies $\mathbf{Pfx}(T) \subseteq \mathbf{Pfx}(T)'$.

Exercise 2.4. Compute the correct verdict for the following cases. S is a specification, T is a trace. Calculate $\mathbf{verdict}(T, S)$. Assume $\Sigma_{\text{in}} = \{A\}$ and $\Sigma_{\text{out}} = \{B\}$.

Case 1:

$$\mathbf{Tr}(S) = \{\langle \emptyset, t\rangle \mid t \geq 0\} \quad (2.54)$$
$$T = \langle \{\langle A, 2\rangle\}, 3\rangle \quad (2.55)$$

Case 2:

$$\mathbf{Tr}(S) = \bigcup \{\mathbf{Pfx}(T) \mid T \in \{\langle \{\langle A, 1\rangle, \langle B, 2\rangle\}, t\rangle \mid t > 2\}\} \quad (2.56)$$
$$T = \langle \{\langle A, 1\rangle, \langle B, 1.5\rangle\}, 2\rangle \quad (2.57)$$

Case 3:

$$\mathbf{Tr}(S) = \bigcup \{\mathbf{Pfx}(T) \mid T \in \{\langle \{\langle A, 1\rangle, \langle B, 2\rangle\}, t\rangle \mid t > 2\}\} \quad (2.58)$$
$$T = \langle \{\langle A, 1.2\rangle, \langle B, 2\rangle\}, 2\rangle \quad (2.59)$$

Case 4:

$$\mathbf{Tr}(S) = \bigcup \{\mathbf{Pfx}(T) \mid T \in \{\langle \{\langle A, 1\rangle, \langle B, 2\rangle\}, t\rangle \mid t > 2\}\} \quad (2.60)$$
$$T = \langle \emptyset, 0.5\rangle \quad (2.61)$$

Exercise 2.5. In this exercise we consider the progressivity of specifications.

Let S be a specification. Prove that there does not exist a constant K and an infinite sequence of traces $\langle E_1, t_1\rangle \preceq \langle E_2, t_2\rangle \preceq \cdots$ such that (1) for all i, $\langle E_i, t_i\rangle \in \mathbf{Tr}(S)$; (2) for all i, $t_i < K$, and (3) for any N, there exists i such that $|E|_i > N$.

Exercise 2.6. Prove that if T is an input failure with respect to S, then it cannot be an output failure, and *vice versa*.

© Antti Huima 2004. All rights reserved.