Chapter 3

Scheme in Everything

The aim of this chapter is to study more carefully how Scheme programs can be used to represent

1. implementations (members of the set $\mathbb{I}$),
2. specifications (members of the set $\mathbb{S}$),
3. testing strategies (members of the set $\mathbb{T}$),
4. and full testers (no formal set defined for these).

The case of implementations seems to be the most straightforward. We are familiar with the idea that computer programs can be employed to describe desired computational processes.

It should be also easy to imagine how a testing strategy can be described by a Scheme program. A testing strategy can be a Scheme program that interacts with an implementation, but that does not yield a verdict. In contrast, a full tester is a testing program that yields verdicts also.

But how can a Scheme program function as a specification? The idea is to see a Scheme program as a *reference implementation* of a desired system. Thus, a specification is a "blessed" or "golden" program that is *assumed* to function correctly *a priori*. Any other system works correctly if and only if it mimics the behavior of the golden program—but we need to define what does the word "mimic" mean here.

3.1 Programs as Implementations

We start by considering the case that we are intuitively most familiar with, namely Scheme programs as implementations. In other words, we will consider Scheme programs as the black boxes, systems, that we are testing.

Assume that the set $\mathbb{I}$ is the set of syntactically correct Scheme programs (as understood here). To "know" how the implementations "work", we need to define the function ξ, i.e. define how the implementations execute and generate behavior, when they are run against testing strategies.

Before assuming that testing strategies are Scxheme programs, let us consider the general case. Namely, suppose that the structure of the set of testing strategies of $\mathbb{T}$ is unknown to us. What can we state about the properties of ξ, given all the common, intuitive understanding we have of the execution of our Scheme programs?

Assume i is an implementation, i.e. now a Scheme program, and that s is a testing strategy. Furthermore, assume that T is a trace, and that for the given ξ function—regardless of how it is defined—it holds that $P[i, s, T] > 0$. In other words, there is a non-zero probability that the strategy s will produce the given trace T against the implementation i.

This implies that there must exist an execution of the program i that produces the output messages of the trace T, assuming that the input events of trace T take place. This does not mean that the sequence if input messages should imply, with probability one, the whole trace T against system i; rather, that there must be a possibility. For example, consider the following Echo Program, reproduced from the first chapter:

```
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)
```

Regardless of how the set of testing strategies is structured, the trace

$$T = \langle \{ \langle \text{“hello”}_{\text{in}}, 0 \rangle, \langle \text{“world”}_{\text{out}}, 0.01 \rangle \}, 2 \rangle \tag{3.1}$$

should not be ever producible by running *any* testing strategy against the Echo Program (as implementation). Of course, we could define the function ξ so as to allow this trace (nothing would prevent us from doing that in principle). But this would break those execution semantics for Scheme that we are accepting. Hence, we have the informal rule: *if for any strategy s it holds that $P[i, s, T] > 0$, then the program i must be capable of producing the outbound message sequence of T, assuming an environment that produces the inbound message sequence of the trace T.*

This rule mentions implicitly a set of traces: the set of those traces a Scheme program can produce against at least one suitably functioning environment. We reserve a notation for this set, even though we can not define its contents formally. Namely, a formal definition would require us to give full operational semantics for Scheme, which is outside our scope. But this lack of formalized semantics does not cause havoc on our aims here.

DEFINITION 3.1. For a Scheme program p, the set $\mathbf{ETr}(p)$ is the set of those traces that the program could produce, assuming suitably functioning environments. If, during assumed execution, an execution error occurs (e.g. division by zero), or the system running p receives a message when the program is not ready to accept one, the program is assumed to output the special output $\mathsf{ERR} \in \Sigma_{\text{out}}$ in “a short time” (it turns out that the actual time from computation error to the ERR output does not matter; what is important is that ERR is produced in a bounded time), and then to halt and to produce no output whatsoever after the error condition. The termination of the last thread of a program execution moves the system into a state where all further inputs are erroneous (the first input causes the ERR output).

Example 3.2. Consider the following simple program p:

```
1 (sync)
```

The set $\mathbf{ETr}(p)$ is the set of all traces void of events, and all traces that contain only input messages and at most one ERR output message. For the program q below,

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

3 (run)
```

the set $\mathbf{ETr}(q)$ contains, however, all traces that contain only input messages. The difference is that when p is “executed”, the first received input message triggers an error (the system is not ready to receive a message). This causes the system to answer with ERR and to be silent from now on. The program q never “crashes”, but silently consumes all input.

Example 3.3. Consider the following variant of the Echo Program, which receives a number, computes it reciprocal, and echoes that back:

```
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 (/ 1 x) (echo))))))

7 (echo)
```

A particular trace generated by this program (denote it by p) is the following:

$$T = \langle\{\langle 2_{\text{in}}, 0.1\rangle, \langle 1/2_{\text{out}}, 0.101\rangle, \langle 3_{\text{in}}, 1\rangle, \langle 1/3_{\text{out}}, 3.005\rangle\}, 4\rangle. \tag{3.2}$$

The following traces belong also to the set $\mathbf{ETr}(p)$, but they cause the execution to go through an error:

$$T = \langle\{\langle 2_{\text{in}}, 0.1\rangle, \langle 1/2_{\text{out}}, 0.101\rangle, \langle 0_{\text{in}}, 1\rangle, \langle \mathsf{ERR}, 1.1\rangle, \langle 4_{\text{in}}, 2\rangle\}, 10\rangle \tag{3.3}$$

and

$$T' = \langle\{\langle 2_{\text{in}}, 0.1\rangle, \langle 1/2_{\text{out}}, 0.101\rangle, \langle \text{“hello”}_{\text{in}}, 1\rangle, \langle \mathsf{ERR}, 1.03\rangle, \langle 4_{\text{in}}, 2\rangle\}, 10\rangle. \tag{3.4}$$

Note that T and T' cause both errors during execution: T involves division by zero, T' involves division by a value that is not a number (the string “hello”).

Example 3.4. Consider still another variant of the Echo Program. We saw this in the introductory chapter. Call this program p:

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

5 (echo)
```

Recall that there is a problem with this program: after receiving a message, the system is not ready to accept a message during a time window that varies from 1 to 100 milliseconds.

The following trace belongs clearly to $\mathbf{ETr}(p)$:

$$T = \langle\{\langle \text{“hello”}_{\text{in}}, 0\rangle, \langle \text{“hello”}_{\text{out}}, 0.01\rangle\}, 2\rangle. \tag{3.5}$$

So does the following:

$$T' = \langle\{\langle \text{“hello”}_{\text{in}}, 0\rangle, \langle \text{“hello”}_{\text{in}}, 0.01\rangle, \langle \mathsf{ERR}, 0.02\rangle\}, 2\rangle. \tag{3.6}$$

The trace T' is generated by the program p as follows: upon receiving the first message, the system calls the random function to obtain eventually a time delay greater than 10 milliseconds. The second message is then received while the execution waits for the timeout. Therefore an error condition occurs (there is no thread reading the external interface).

However, the following trace is also a member of $\mathbf{ETr}(p)$:

$$T'' = \langle\{\langle \text{“hello”}_{\text{in}}, 0\rangle, \langle \text{“hello”}_{\text{out}}, 0.005\rangle, \langle \text{“hello”}_{\text{in}}, 0.01\rangle\}, 2\rangle. \tag{3.7}$$

Note that the set of input events is exactly the same for the traces T'' and T'. The difference is that in the execution that produces T'', the chosen timeout happens to be only five milliseconds.

3.2 Programs as Testing Strategies

Let us now consider the case of a Scheme program as a testing strategy. Thus, assume that the set $\mathbb{T}$ is the set of syntactically correct Scheme programs.

We can see these testing strategy programs basically as implementations that just happen to reside on the other side of the testing interface. Thus, if the execution of a testing strategy program triggers an error, the tester part sends ERR and silences down. The implementation is not assumed to be capable of handling the ERR input; hence, a tester error occurs.

Assuming this, we can intuitively figure mostly the structure of the function ξ, as giving the semantics for executing two Scheme programs whose inputs and outputs have been linked together. The particular structure of test steps we will leave open for now.

3.3 Programs as Testers

A Scheme program can work as a full tester if it can yield verdicts. For this end, we can assume the existence of the following Scheme functions: `pass`, and `fail`. These are two functions that, when called, somehow magically report a testing verdict to the outside world, and then cause the execution to terminate.

Here is an example of one (deterministic) tester for the Echo Program:

```
1 (begin
2   (sync (output "hello"
3                 (sync (input x
4                               (if (equal? x "hello")
5                                   (pass)))
6                       (wait 1 #f))))
7   (fail))
```

3.4 Programs as Specifications

We come now to the most interesting part: how can Scheme programs function as specifications?

Recall that the set $\mathbf{ETr}(p)$ is the set of traces that could be obtained by somehow interacting with a program p. If we would like to see p as a specification, i.e. as a reference implementation, could we use $\mathbf{ETr}(p)$ as the set of valid traces? In other words, could we postulate $\mathbf{ETr}(p) = \mathbf{Tr}(p)$?

The answer is resolute *no*. The problem is that the set $\mathbf{ETr}(p)$ contains traces that involve an execution error, implying the output of ERR and the silencing of the executing system due to the error afterwards. We do not want to see these traces as valid traces. More rigorously, a valid trace of a reference implementation must be a trace that can be produced by executing against the reference implementation without an error condition.

Therefore, let $\mathbf{VTr}(p)$ denote the set of execution traces that p can produce, without outputting ERR. Thus,

$$\mathbf{VTr}(p) = \{\langle E, t\rangle \mid \langle E, t\rangle \in \mathbf{ETr}(p) \wedge \neg\exists t' : \langle \mathsf{ERR}, t'\rangle \in E\}. \tag{3.8}$$

Could it now be that $\mathbf{VTr}(p) = \mathbf{Tr}(p)$?

Again, the answer is *no*. The remaining problem can be illustrated by the following program. Denote it by p:

```
1 (define (buggy-void)
2   (sync (input x
3               (sync (wait 1
4                           (/ 1 0)
5                           (buggy-void))))))

6 (buggy-void)
```

The following trace belongs to $\mathbf{VTr}(p)$:

$$T = \langle\{\langle \text{“hello”}_{\text{in}}, 1\rangle\}, 1.5\rangle. \tag{3.9}$$

However, all continuations for T that are long enough must execute through a division by zero at $t = 2$. But after this division by zero, the ERR output occurs within a short time frame. Therefore, the requirement that every valid trace of a specification can be extended to an arbitrarily long one can not be fulfilled if $\mathbf{VTr}(p) = \mathbf{Tr}(p)$. This discussion also suggest a solution: we let $\mathbf{Tr}(p)$ be the greatest subset of p that *has* this desired property of *seriality*.

The term *serial* comes from the world of modal logic.

DEFINITION 3.5. Let $\mathcal{T}$ be a set of traces. Let a trace $T = \langle E, t\rangle$ be *serial* in $\mathcal{T}$ if for every $t' > t$ it holds that there exists for some E' a trace $\langle E', t'\rangle$ in $\mathcal{T}$ such that $T \preceq \langle E', t'\rangle$.

Call a set $\mathcal{T}$ of traces *serial* if every trace $T \in \mathcal{T}$ is serial in $\mathcal{T}$.

Let p be a Scheme program. The set $\mathbf{Tr}(p)$ is the greatest subset of $\mathbf{VTr}(p)$ such that $\mathbf{Tr}(p)$ is serial. (The existence of this set is guaranteed because being serial is a closure property.)

Example 3.6. Let us continue to denote by p the “buggy-void” program above. Note that the set $\mathbf{Tr}(p)$ contains only traces that are completely empty of events! Namely, if an event is received by this program, the program crashes in one second. This means that there is no arbitrarily long continuation after *any* input event. However, the program does not crash if it never receives anything. Hence,

$$\mathbf{Tr}(p) = \{\langle\emptyset, t\rangle \mid t \geq 0\}. \tag{3.10}$$

3.4.1 The 'require' procedure

This all ground work might seem a bit heavy, but we have now a foundation on which we can build a very useful construct: the `require` compound procedure. Namely, define:

```
1 (define (require x)
2   (if (not x)
3       (/ 1 0)))
```

Consider now the following variant of the Echo Program, which "wants" only to echo integers:

```
1 (define (echo)
2   (sync (input x
3                (require (integer? x))
4                (wait-and-send x))))

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

9 (echo)
```

Let this program be called p and consider it as a specification. Let i be an implementation that should implement this specification. Suppose that the following trace has been produced against i by some testing strategy:

$$T = \langle \{ \langle \text{“hello”}_{\text{in}}, 1 \rangle \}, 3 \rangle. \tag{3.11}$$

Clearly, something has gone astray: there is no echo response, but neither is the input message an integer. What does the specification p say of this trace?

First, note that T belongs to $\mathbf{ETr}(p)$. However, if the program p would be executed, it would crash at division by zero in the call to `require`, because the integrality check for `x` would not be passed. Hence, T does not belong to $\mathbf{VTr}(p)$, because any execution that produces T involves division by zero. Therefore, $T \notin \mathbf{Tr}(p)$. Thus the trace T is invalid with respect to the specification. What remains is to find out if T is an input or an output failure.

The set of valid prefixes of T is the set

$$\{ \langle \emptyset, t \rangle \mid 0 \leq t \leq 1 \}. \tag{3.12}$$

Hence, the set K of the time stamps of the valid prefixes is the closed interval $[0, 1]$. We can choose any value from K, say, $t = 1$. Any extension of the trace $\langle \emptyset, 1 \rangle$ that belongs to $\mathbf{Tr}(p)$ is either empty of events, or begins with the input of an integer message. Therefore, the failure type is an input failure. This means that the corresponding verdict is ERROR. Thus, there has been a *tester error*.

3.4.2 On intensional style

Observe how the careful definitions that we have laid down now enable us to describe a very powerful specification construct. Namely, `require` is a tool for both restricting acceptable inputs in a specification as well as a tool for describing *intensional specifications* in general. As an example of the latter use, consider the following variant of the Echo Program that echoes square roots when they exist (denote it by p):

```
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               (let ((value (any-rational-number)))
7                 (require (= (* value value) x))
8                 (sync (output value (echo)))))))
```

```
9 (echo)
```

If this program would be really executed it would crash all the time: the probability that the random number generator behind `any-rational-number` returned the exact square root of `x` could be very low, even when the square root would be rational. But, if we assume that this probability is non-zero anyway, then every trace where the correct square root value is returned to the environment belongs to $\mathbf{VTr}(p)$, and thus also here to $\mathbf{Tr}(p)$. But note that the actual computation of $\sqrt{x}$ is not written in the program; the program guesses a number, and then verifies its correctness. This "intensional style" can be used in principle as a very powerful specification construct.

This intensional style can be also very confusing. Consider the following program:

```
1 (define (guess)
2   (let ((v (any-integer)))
3     (sync (input x
4                  (require (= v x))
5                  (sync (wait 0.1
6                              (sync (output "ok" (wait-for-ever)))))))))

7 (define (wait-for-ever)
8   (sync (input x (wait-for-ever))))

9 (guess)
```

Consider now the following trace, executed against an implementation of this specification (denote the specification program by p):

$$T = \langle \{\langle 3, 0\rangle\}, 10\rangle. \tag{3.13}$$

Intuitively, this trace corresponds to sending 3 to the program, after which the program finds out that 3 is not equal to the value it has silently chosen in a call to `any-integer`, after which the program refuses to send "ok" back. Now the specification p *prohibits* this kind of a behavior (be sure you understand why). But is the failure that T represents an input failure or an output failure?

Can you derive the correct conclusion intuitively?

I would guess that our common style of thinking, conditioned to understand causal relationships, could lead you to assume that this is an input failure: the tester guessed wrongly the number that the system had chosen, so the tester messed up.

But this is the wrong conclusion. Namely, the trace

$$T' = \langle \{\langle 3, 0\rangle, \langle \text{"ok"}, 3.1\rangle\}, 10\rangle \tag{3.14}$$

is a valid trace, obtained from T by changing an output event at $t = 3.1$, and hence there is an output failure. The point is that the order of execution within a Scheme program that is a specification does not matter *per se*; even though we think that the value of `v` was chosen "before" the value of `x`, from the point of view of external behavior it is the SUT that has responsibility here. After all, the SUT could have guessed the forthcoming value of `x` anyway...

To conclude this discussion we could make the point that maybe a specification like this is no likely to pop out in the real world. Regardless, the fact that even strange cases like this are handled meaningfully adds credibility on the basic framework.

3.5 Conformance is Trace Inclusion

We are now on the verge of a very important observation.

3.5.1 Correct Testing Strategies

We could call a testing strategy s *correct* with respect to a specification S, if for any implementation i,

$$P[i, s, T] > 0 \implies \mathbf{verdict}(T, S) \neq \text{ERROR}. \tag{3.15}$$

In other words, a correct testing strategy never causes input failures; it is well-behaving. The following definition formalizes this:

DEFINITION 3.7. Let S be a specification. We denote by $\mathbf{CT}(S)$ the set of *correct testing strategies* for the specification S, defined by

$$\mathbf{CT}(S) = \{s \mid s \in \mathbb{T} \wedge \forall i, T : P[i, s, T] > 0 \implies \mathbf{verdict}(T, S) \neq \text{ERROR}\}. \tag{3.16}$$

3.5.2 Universal Existence of Testing Strategies

Now consider an implementation i that is a Scheme program. The set $\mathbf{ETr}(i)$ is the set of all possible traces that "could be produced against an environment". We need to ask: is every trace in this set also producible by a testing strategy?

We assume here that this is true. Furthermore, we assume that if $T \in \mathbf{ETr}(i)$ and that $\mathbf{verdict}(T, S) \in \{\text{PASS}, \text{FAIL}\}$ for a specificiation S, then there exists a correct testing strategy (element of $\mathbf{CT}(S)$) that is capable of producing the trace T against i with non-zero probability. It will be shown later that this assumption is not groundless. It is possible to construct a general family of correct testers such that every execution traces that results in one of these two verdicts can be constructed by an element of the family. But we need to assume that this general family is part of the set $\mathbb{T}$.

Hence, we get

$$\mathbf{ETr}(i) = \{T \mid \exists s \in \mathbb{T} : P[i, s, T] > 0\}. \tag{3.17}$$

Furthermore, denote by $\mathbf{ETr}(i, S)$ the set of all potential traces when we make the assumption that the testing strategies are correct with respect to a specification S:

$$\mathbf{ETr}(i, S) = \{T \mid \exists s \in \mathbf{CT}(S) : P[i, s, T] > 0\}. \tag{3.18}$$

3.5.3 Conclusion

Assume now that

$$\forall T \in \mathbf{ETr}(i, S) : \mathbf{verdict}(T, S) = \text{PASS}. \tag{3.19}$$

By definition of the **verdict** function, it then holds that

$$\mathbf{ETr}(i, S) \subseteq \mathbf{Tr}(S). \tag{3.20}$$

On the other hand, assuming (3.20), (3.19) is implied. Hence, these two conditions are *equivalent*. In other words, the implementation i is completely correct if and only if those traces generated by i against a correctly working tester form a subset of the traces of the specification S.

Note carefully, that the condition (3.20) does not mention a testing strategy directly. The condition can be understood as stating directly something about two trace sets: the set of traces an implementation can produce against a "correctly working" environment, and the set of traces of a specification. This conformance relation, which we have derived through to concept of a testing strategy, could thus in principle be checked for without every actually "executing" any particular testing strategy all. For example, model checking could be used for this aim.

The identification of conformance with trace inclusion as in (3.20) is *the* underlying notion in the different variants of formal conformance testing theories. Notably, it is the foundation for the "ioco" family of testing theories by Jan Tretmans *et al*. We will return to this subject later when we consider these theories on a more detailed level.

One note is due. Namely, trace inclusion as above is a form of "unconditional" correctness. The set $\mathbf{ETr}(i)$ is the set of all those traces that the implementation i can produce with a probability greater than zero against a suitable testing strategy. Now there could exists a trace $T \in \mathbf{ETr}(i)$ such that $\mathbf{verdict}(T, S) = \text{FAIL}$ (i.e. T is invalid), but so that for any testing strategy s, say, $P[i, s, T] < 10^{-10}$. If T would be the only failing trace that could be produced by i, would you say i is conforming to S or not? Would you expect to be able to demonstrate that i is incorrect in a testing session? These questions point toward a more general notion of probabilistic correctness, which is outside the basic, yet very important, view that eventually conformance *is* trace inclusion.

Exercises

Exercise 3.1. Consider the following Scheme program p:

```
1 (define (main)
2   (sync (input x
3                (sync (wait 1
4                            (sync (output 2 (fall-a-sleep))))))))
5 (define (fall-a-sleep) (sync (input x (fall-a-sleep))))
6 (main)
```

Is the following trace:

$$T = \langle \{ \langle 1_{\text{in}}, 1 \rangle, \langle 2_{\text{out}}, 2 \rangle \}, 3 \rangle \tag{3.21}$$

a valid trace with regards to p as a specification? If not, what is $\textbf{verdict}(T, p)$?

Exercise 3.2. Consider the following Scheme program p:

```
1 (define (main)
2   (sync (input x
3                (sync (wait (random 10) ;; returns number from [0, 10)
4                            (sync (output (random 10) (fall-a-sleep))))))))
5 (define (fall-a-sleep) (sync (input x (fall-a-sleep))))
6 (main)
```

Is the following trace:

$$T = \langle \{ \langle 1_{\text{in}}, 1 \rangle, \langle 2_{\text{out}}, 2 \rangle \}, 3 \rangle \tag{3.22}$$

a valid trace with regards to p as a specification? If not, what is $\textbf{verdict}(T, p)$?

Exercise 3.3. Consider the following Scheme program p:

```
1 (define (main)
2   (sync (input x
3                (sync (wait 1 (sync (output 2 (fall-a-sleep)))))
4         (wait 0.5 (fall-a-sleep)))))
5 (define (fall-a-sleep (sync (input x (fall-a-sleep)))))
6 (main)
```

Is the following trace:

$$T = \langle \{ \langle 1_{\text{in}}, 1 \rangle, \langle 2_{\text{out}}, 2 \rangle \}, 3 \rangle \tag{3.23}$$

a valid trace with regards to p as a specification? If not, what is $\textbf{verdict}(T, p)$?

Exercise 3.4. Consider the following Scheme program p:

```
1 (define (main)
2   (sync (input x
3                (let ((y (+ x 1)))
4                  (sync (wait 1
5                              (sync (output y
6                                            (require (= y 3))
7                                            (fall-a-sleep)))))))))
8 (define (fall-a-sleep (sync (input x (fall-a-sleep)))))
9 (main)
```

Is the following trace:

$$T = \langle \{ \langle 1_{\text{in}}, 1 \rangle, \langle 2_{\text{out}}, 2 \rangle \}, 3 \rangle \tag{3.24}$$

a valid trace with regards to p as a specification? If not, what is $\textbf{verdict}(T, p)$?

Exercise 3.5. Consider the following Scheme program p:

```
1 (define (main)
2   (if (= 2 (random 5))
3       (sync (input x (sync (wait 1 (sync (output 2 (fall-a-sleep)))))))
4       (sync (input x (sync (wait 1 (sync (output 3 (fall-a-sleep)))))))))
5 (define (fall-a-sleep (sync (input x (fall-a-sleep)))))
6 (main)
```

Is the following trace:

$$T = \langle \{ \langle 1_{\text{in}}, 1 \rangle, \langle 2_{\text{out}}, 2 \rangle \}, 3 \rangle \tag{3.25}$$

a valid trace with regards to p as a specification? If not, what is $\textbf{verdict}(T, p)$?

Exercise 3.6. Consider the following Scheme program p:

```
1 (define (main)
2   (sync (wait 1 (sync (output 2 (fall-a-sleep))))))
3 (define (fall-a-sleep (sync (input x (fall-a-sleep)))))
4 (main)
```

Is the following trace:

$$T = \langle \{ \langle 1_{\text{in}}, 1 \rangle, \langle 2_{\text{out}}, 2 \rangle \}, 3 \rangle \tag{3.26}$$

a valid trace with regards to p as a specification? If not, what is $\textbf{verdict}(T, p)$?