An expressive semantics of mocking

Josef Svenningsson[1], Hans Svensson[2], Nicholas Smallbone[1],
Thomas Arts[2], Ulf Norell[1,2], and John Hughes[1,2]

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] Quviq, Gothenburg, Sweden

Abstract. We present a semantics of mocking, based on a process calculus-like formalism, and an associated mocking framework. We can build expressive mocking specifications from a small, orthogonal set of operators. Our framework detects and rejects ambiguous specifications as a validation measure. We report our experience testing software components for the car industry, which needed the full power of our framework.

1 Introduction

Software components rarely exist in isolation; most components not only provide an API, but depend on the APIs of other components. When a component is tested in isolation, then these other APIs must be replaced by a suitable simulation. Nowadays "mocks" are often used for this purpose, which not only simulate the other components, but also help to check that they are used correctly.

There are many mocking frameworks available to support mocking, such as Google Mock [9] for C++, or jMock [11] or Mockito [12] for Java. Yet we developed a new framework of our own—why?

We recently designed conformance tests for parts of the AUTOSAR automotive software standard [3]. The goal was to test different vendors' implementations of AUTOSAR components for compliance with the standard. We needed mocks in order to test each component in isolation. We had three main requirements which ruled out existing mocking frameworks.

Expressive AUTOSAR does not completely specify how a compliant component must behave, and different vendors interpret the standard differently. Therefore, the system under test might invoke the mocks in a variety of very different ways. As we cannot tailor our tests to the vendor's implementation, our mocks must handle this diversity instead. To allow diverse behaviour without making the mocks too permissive, we need an expressive mocking framework.

Orthogonal Many mocking frameworks have a non-orthogonal feature set. For example, mocking frameworks support optional calls, which the system under test may call or ignore, but it is often *not* possible to mark a sequence of calls as optional, so that either the whole sequence must be called or none at all.

In the AUTOSAR project we used QuickCheck [6, 2] to model the software components. From the model we can generate test cases and corresponding

mocks; generating the mocks is extremely painful if the mocking framework imposes arbitrary restrictions on what we can write. We want the freedom to combine the features of the mocking framework however we like.

Clear and unambiguous specifications In most mocking frameworks, the meaning of a specification can be quite subtle, a point we illustrate in Section 2. For example, these frameworks have rules for resolving ambiguity, and the user can exploit these rules in writing specifications. This is convenient but makes it hard to say what a given specification means.

Our AUTOSAR mocking specifications are, by necessity, sometimes long and complex. They are tricky to get right. The last thing we want from our framework is a subtle semantics! We want each mocking specification to have a simple, declarative meaning. Likewise, we want the mocking framework to reject ambiguous specifications, rather than make arbitrary choices: this reduces the number of potential pitfalls.

One might expect that we could use an ambiguous specification to mock a *nondeterministic* component, if the framework resolves ambiguity randomly. We believe this is the wrong approach, because it makes tests unrepeatable. Instead, the test suite itself should choose a particular deterministic interpretation.

This paper presents a new mocking framework which is expressive, is built from a small core of orthogonal features, has a simple, compositional semantics where every specification has a clear meaning, and which avoids making arbitrary choices during test execution by rejecting ambiguous specifications. Although our requirements came from the AUTOSAR testing project, we believe these features are compelling in their own right, and are especially important when testing large components. The contributions of the paper are as follows:

- We present a new framework for mocking (Sections 3–4). The framework is given two semantics, a simple, compositional denotational semantics and a small-step operational semantics. The two semantics have been proved equivalent (see the accompanying technical report [13]).
- We avoid making arbitrary choices during test execution by ruling out ambiguous mocking specifications. Specifically, we provide a procedure to validate specifications (Section 5) which rules out specifications which are ambiguous. The validation is sound with respect to the semantics. Perhaps surprisingly, it is also complete, which means that if we reject a specification, it must be ambiguous, and we can moreover find a trace that demonstrates the ambiguity. The soundness proof, a sketch of the completeness proof, and a link to the full formalization are found in the tech. report [13]
- We extend our basic framework to make it practical and describe how to implement it in a memory-efficient way (Section 6).
- We report on our experience using an earlier version of this framework in a large industrial case study writing specifications for, and then testing implementations of, automotive software (Section 7).

2 Why a Mocking Semantics?

Before going into the details of our new mocking framework, we will explain why we are dissatisfied with the non-compositional semantics of conventional mocking frameworks. We use Google Mock [9] (and Google Test [10]) purely as a representative for existing mocking frameworks.

Consider a small test for a `Dashboard` component. The dashboard is connected to a speed sensor and a display, and is supposed to read the speed and update the display appropriately. In this example, the dashboard has a correct C implementation which we want to test. In order to test it, we mock the sensor and display component.

```
TEST(Dashboard, Test1) {
 MockSensor mSensor;
 MockDisplay mDisplay;
 EXPECT_CALL(mSensor, readSpeed()).WillOnce(Return(10));
 EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, _));

 Dashboard dashboard(&mSensor, &mDisplay);
 dashboard.main(); // Actual test
}
```

The test creates a mock sensor and a mock display, and a concrete `Dashboard` object containing the mock objects. Thereafter, the mock objects are prepared to expect a call of `readSpeed()` (returning 10) and `updateDisplay(Display::SPEED, _)` respectively (where `_` matches any argument). The test finally calls the main function of `Dashboard`. When a mocked object is destroyed, the framework checks that all and only the expected calls have been made. This test will pass provided that `dashboard.main()` calls the mocked functions exactly as specified.

Let us enrich the example test by adding two more calls of `dashboard.main()` and having the mocked function `readSpeed` return a different value each time. We tell `updateDisplay` that it will be called three times (by adding `Times(3)` to the specification), and call the main function three times:

```
MockSensor mSensor;
MockDisplay mDisplay;
EXPECT_CALL(mSensor, readSpeed())
  .WillOnce(Return(10))
  .WillOnce(Return(6.7))
  .WillOnce(Return(12.5));

EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, _)).Times(3);

Dashboard dashboard(&mSensor, &mDisplay);
dashboard.main();
dashboard.main();
dashboard.main(); // main x3
```

This test will pass for correct implementations of the dashboard. Next, suppose we want to be a little bit more precise. It so happens that the `Dashboard` should convert the sensor speed, given in *m/s*, to *km/h*; i.e. if `readSpeed` returns 10, `updateDisplay` should be called with 36 as its second argument. We change the expected calls to:

```
EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, 36)).Times(1);
EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, _)) .Times(2);
```

Surprisingly, this test fails even if the implementation does the correct thing. It turns out that expectations are put on a stack, so are tested in the reverse order that they are defined. Thus, the correct way to specify this would be

```
EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, _)) .Times(2);
EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, 36)).Times(1);
```

even though the call returning 36 happens first. And indeed, this test passes.

Now suppose that we change the specification so that the final call to `readSpeed` returns 10 instead of 12.5:

```
EXPECT_CALL(mSensor, readSpeed())
  .WillOnce(Return(10))
  .WillOnce(Return(6.7))
  .WillOnce(Return(10));

EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, _)) .Times(2);
EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, 36)).Times(1);
```

We might expect this test to pass, but it does not! The reason is that (by default) expectations are not removed from the stack once they are fulfilled. Thus as soon as the function `updateDisplay` is called with argument `36` it remains on the stack as being called once. The second time it is called it increases the call count of the `updateDisplay` with argument `36` instead of increasing the call count of `updateDisplay` with arbitrary argument.

The above mocking specification looks ambiguous, since a second call with argument `36` can be handled in two ways: it can be accepted by the first clause or rejected by the second. The mocking framework has arbitrarily chosen the second way.

The way to fix this test in Google Mock is either to expect `36` twice, or to tell the second expectation to retire once it is fulfilled with the feature `RetiresOnSaturation()`. We choose the second option and the test now passes:

```
EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, _)) .Times(2);
EXPECT_CALL(mDisplay, updateDisplay(Display::SPEED, 36)).Times(1)
  .RetiresOnSaturation();
```

While conventional mocking frameworks have a precise semantics, it is quite complicated. There are subtle interactions between features because the semantics is not compositional and ambiguous specifications are given an arbitrary, though documented, semantics.

In this case, the test failed and therefore we found out that the mocking specification was ambiguous. More worrying is that, by resolving an ambiguity the wrong way, the framework might allow a test to pass that should fail. We would like to be alerted to such problems.

A complex semantics also places a mental burden on the user. We believe that this burden becomes worse as the mocking specifications become bigger. We will now present our approach to mocking, which brings a simple, surprise-free semantics, and ambiguity checking to avoid having to make arbitrary choices.

3 Introduction to the Mocking Language

Our running example will be a dashboard similar to that of Section 2. The dashboard executes a main loop 25 times per second. Each time round, it reads a number of signals, such as the speed, battery status, outside temperature, etc., and updates a display accordingly. The display is a different software component and is mocked by an `update_display` function.

We start with a simple main loop that only reads the speed and then updates the display. It will first call the mocked function `read_speed`, which returns a value in *m/s*; let's say that it should return 5.833 m/s. The mocking specification $\mathit{read\_speed} \mapsto 5.833$ says that the software under test must call `read_speed`, and the mocked function will return 5.833. We refer to a single call and return as an *event*; by combining events we can build more complex mocking specifications.

Next, the dashboard must make the display show 21 km/h. It does this by calling `update_display(speed,21)`; as before, we model this call with an event $\mathit{update\_display}(\mathit{speed}, 21) \mapsto ()$.[3] To say that the dashboard must call `read_speed` and `update_display` in that order, we combine the two events with the sequential composition operator "$\cdot$". The resulting specification is:

$$\mathit{read\_speed} \mapsto 5.833 \cdot \mathit{update\_display}(\mathit{speed}, 21) \mapsto ()$$

Now we turn to the battery level. This display only needs to be updated once a second, though it may be updated more often. Since our mocking specification only captures 1/25 of a second, we cannot check this directly; instead, we allow the dashboard *optionally* to update the display, and will check in the test suite itself that the display is updated often enough by counting the calls to `update_display`. To express optional behaviour we add two new constructs to the mocking language. The $+$ operator allows the software under test to behave according to either of two specifications, while the empty specification ε forbids any calls. We may then express an optional behaviour by giving the software under test the option of having that behaviour or not doing anything:

$$(\mathit{read\_battery} \mapsto 234 \cdot \mathit{update\_display}(\mathit{battery}, 70) \mapsto ()) + \varepsilon$$

Another feature of the dashboard is that when driving in bright sunlight, the display may light up. Not all cars have this feature. Moreover, some dashboards

[3] If a function's return type is `void`, we use () for the return value.

read the light sensor each time they update a part of the display, while others only read it once per loop. But whenever the dashboard reads the light sensor, it must then update the display brightness. The dashboard may read the light sensor any number of times per loop, which we can model using the $*$ operator:

$$(\mathit{read\_light} \mapsto 6 \cdot \mathit{light\_display} \mapsto ())^*$$

The three specifications above capture three aspects of the dashboard. To mock the dashboard as a whole, we combine the three specifications with the parallel composition operator "$||$". This says that the dashboard may interleave the execution of the three specifications, but must respect the order of events within each single specification. For example, the dashboard may read the speed, then the light sensor, then set the display brightness, then update the display:

$$\begin{aligned}&(\mathit{read\_speed} \mapsto 5.833 \cdot \mathit{update\_display}(\mathit{speed}, 21) \mapsto ()) \\ ||\ &(\mathit{read\_light} \mapsto 6 \cdot \mathit{light\_display} \mapsto ())^* \\ ||\ &((\mathit{read\_battery} \mapsto 234 \cdot \mathit{update\_display}(\mathit{battery}, 70) \mapsto ()) + \varepsilon)\end{aligned}$$

From this specification we can automatically generate mocks. Our mocks check that the calls made by the dashboard precisely match the calls in the specification: no extra calls, no missing calls, and all calls in the right order.

4 A Process Calculus for Mocking

We have now seen all of the features of our mocking language, and begin a formal treatment of its semantics. Mocking specifications resemble terms in a process calculus, and their syntax is summarised below. An event $a \mapsto z$ denotes calling the function a to get result z. For now we treat a and z abstractly; in Section 6 we will breathe life into the calculus by allowing events to be real function calls.

$$p \quad ::= \quad \varepsilon \quad | \quad a \mapsto z \quad | \quad p \cdot q \quad | \quad p \,||\, q \quad | \quad p + q \quad | \quad p^*$$

We want to assign meaning to mocking specifications. We therefore define a denotational semantics in terms of traces; a trace is a sequence of events. The language $\mathcal{L}(p)$ of a process is the set of traces that the process accepts, i.e. that satisfy the mocking specification, and is defined as follows:

$$\begin{aligned}\mathcal{L}(p \cdot q) &= \{st \mid s \in \mathcal{L}(p) \wedge t \in \mathcal{L}(q)\} \\ \mathcal{L}(p + q) &= \mathcal{L}(p) \cup \mathcal{L}(q) \\ \mathcal{L}(p \,||\, q) &= \{u \mid s \in \mathcal{L}(p) \wedge t \in \mathcal{L}(q) \wedge u \text{ is an interleaving of } s \text{ and } t\} \\ \mathcal{L}(p^*) &= \{s_1 s_2 \cdots s_n \mid n \in \mathbb{N} \text{ and for all } i,\ s_i \in \mathcal{L}(p)\} \\ \mathcal{L}(a \mapsto z) &= \{a \mapsto z\} \\ \mathcal{L}(\varepsilon) &= \{\varepsilon\}\end{aligned}$$

This semantics is compact and easy to understand, and ideal for understanding the behaviour of a mocking specification. However, it is of little use for implementing the mocking framework. It accepts or rejects whole execution traces,

but during test execution we are given a single call at a time and have to return a single result. Therefore, we also provide a small-step semantics. The small-step semantics is more complicated than the denotational one. In order to make sure that we have not made a mistake, we have proved that both semantics are equivalent: see the accompanying tech. report [13].

The small-step semantics is based on two judgements: reduction $p \to_{a,z} q$ means that on a call to a, the process p will return z and behave as q thereafter, while "p is accepting" means that p accepts the empty trace: the test case may finish without calling any mocked functions. We design both judgements so that they coincide with the denotational semantics.

A process p should be accepting if $\varepsilon \in \mathcal{L}(p)$. Looking at the denotational semantics, we get the following rules: $p \cdot q$ is accepting if both p and q are accepting (likewise $p \mid\mid q$), $p + q$ is accepting if either p or q are accepting, p^* is accepting, ε is accepting and $a \mapsto z$ isn't.

The most interesting case for reduction is sequential composition. To reduce $p \cdot q$, we can either reduce p or, if p is accepting, remove it and reduce q. This gives the following rules:

$$\frac{p \ \to_{a,z} \ q}{p \cdot r \ \to_{a,z} \ q \cdot r} \ (\textsc{ThenL}) \qquad \frac{p \text{ is accepting} \qquad q \to_{a,z} r}{p \cdot q \ \to_{a,z} \ r} \ (\textsc{ThenR})$$

We can also derive these rules from the denotational semantics. Suppose we have a trace $st \in \mathcal{L}(p \cdot q)$, where $s \in \mathcal{L}(p)$ and $t \in \mathcal{L}(q)$. ThenL: If s is non-empty, the first event in st is from $\mathcal{L}(p)$, hence we should reduce p to, say, p'. The remainder of st is a trace from $p' \cdot q$, so we should reduce to that. ThenR: If s is empty, which can only occur if $\varepsilon \in \mathcal{L}(p)$, the trace is simply $t \in \mathcal{L}(q)$, hence we should reduce q to, say, q'. The remainder of st is a trace from q', so we should reduce to that.

Reasoning either informally or from the denotational semantics, we find the other reduction rules. To reduce a parallel composition $p \mid\mid q$, reduce either p or q; to reduce a choice $p + q$, remove one of the choices and reduce the one that's left. To reduce p^*, expand it to $p \cdot p^*$ and then reduce p; finally, an event $a \mapsto z$ reduces to ε. This is captured in the rules below.

$$\frac{p \to_{a,z} q}{p \mid\mid r \ \to_{a,z} \ q \mid\mid r} \ (\mid\mid\text{L}) \qquad \frac{q \to_{a,z} r}{p \mid\mid q \ \to_{a,z} \ p \mid\mid r} \ (\mid\mid\text{R}) \qquad \frac{p \to_{a,z} q}{p^* \ \to_{a,z} \ q \cdot p^*} \ (*)$$

$$\frac{p \to_{a,z} q}{p + r \ \to_{a,z} \ q} \ (+\text{L}) \qquad \frac{q \to_{a,z} r}{p + q \ \to_{a,z} \ r} \ (+\text{R}) \qquad \frac{}{a \mapsto z \ \to_{a,z} \ \varepsilon} \ (\textsc{Event})$$

If we are not interested in the result of the call, we write $p \to_a q$, and say that p a-reduces (or just reduces) to q; if we are not interested in the resulting process q either, we just write $p \to_a$, and say that p can consume a. We lift the terminology from single events to whole traces in the natural way.

5 Ambiguity detection

As argued in the Introduction and Section 2, we want to forbid ambiguous specifications, because they lead to complex semantics, or to unrepeatable tests if resolved at random. An example of an ambiguous specification in our language is $a \mapsto z_1 + a \mapsto z_2$: if the program calls a, we do not know whether to return z_1 or z_2. We will see in Section 6.1 that the user does not decide what value an event will return until that event is called, so we must also reject $a \mapsto z + a \mapsto z$—we have no way of knowing that both events will always return the same value.

This suggests the following definition of ambiguity: p is ambiguous if for some call a, there are two applicable reduction rules for $p \to_a$. A process is also ambiguous if it reduces to an ambiguous process. Our process $a \mapsto z + a \mapsto z$ is ambiguous because, for the call a, the rules +L and +R both apply.

Here are some examples of ambiguous processes:

- $a \mapsto z_1 + a \mapsto z_2$ is ambiguous, as above. In general, if $p \to_a$ and $q \to_a$, then $p + q$ is ambiguous.
- $(a \mapsto z_1 \cdot b \mapsto z_2) \mid\mid b \mapsto z_3$ is ambiguous: after a call to a, it reduces to $b \mapsto z_2 \mid\mid b \mapsto z_3$, in which there are two b-reductions. In general, if p and q have overlapping alphabets, then $p \mid\mid q$ is ambiguous.
- $(a \mapsto z_1 + \varepsilon) \cdot a \mapsto z_2$ is ambiguous: calling a, we could return either z_1 or z_2.
- Along the same lines, $a \mapsto z_1 \cdot (a \mapsto z_2 + \varepsilon) \cdot a \mapsto z_3$ is ambiguous: after a call to a, we are left with $(a \mapsto z_2 + \varepsilon) \cdot a \mapsto z_3$, essentially the previous example.

The examples above tell us how to detect ambiguity. We will start with $+$ and $\mid\mid$. Note that the two constructs need different rules: the second example is ambiguous, but replacing $\mid\mid$ by $+$ it becomes unambiguous. With $+$, the first call needs to tell us which alternative to choose, but with $\mid\mid$ every call needs to have this property.

- If $p \to_a$ and $q \to_a$, then $p + q$ is ambiguous because rules +L and +R both apply.
- If $a \in \text{alphabet}(p) \cap \text{alphabet}(q)$ then $p \mid\mid q$ is ambiguous because we can reach a process $p' \mid\mid q'$ where $p' \to_a$ and $q' \to_a$; rules ||L and ||R then both apply. (The alphabet of a process is simply the set of events that appear syntactically in it.)

We will define a function $p\checkmark$ that checks that p is unambiguous. For now we only define the easy cases:

$$
\begin{aligned}
p + q \checkmark &= p \checkmark \wedge q \checkmark \wedge \neg\exists a\, (p \to_a \wedge q \to_a) \\
p \mid\mid q \checkmark &= p \checkmark \wedge q \checkmark \wedge \text{alphabet}(p) \cap \text{alphabet}(q) = \emptyset \\
a \mapsto z \checkmark &= \text{true} \\
\varepsilon \checkmark &= \text{true}
\end{aligned}
$$

Sequential composition is trickier. Looking at $(a \mapsto z_1 + \varepsilon) \cdot a \mapsto z_2$, we see that the reduction rules ThenL and ThenR both apply, the first because

$a \mapsto z_1 + \varepsilon$ can consume a and the second because $a \mapsto z_1 + \varepsilon$ is accepting and $a \mapsto z_2$ can consume a. Generalising to an arbitrary sequential composition $p \cdot q$:

- If $p \rightarrow_a$, then rule ThenL applies.
- If p is accepting and $q \rightarrow_a$, then rule ThenR applies.

If both conditions are true, $p \cdot q$ is ambiguous. The final example above, $a \mapsto z_1 \cdot (a \mapsto z_2 + \varepsilon) \cdot a \mapsto z_3$, does not satisfy the above conditions, but is still ambiguous because it a-reduces to a process that does. Let us say that a overlaps p, or $p \,?\, a$, if there is a trace under which p reduces to a process p', such that $p' \rightarrow_a$ and p' is accepting. Then we may generalise our remarks above: if $p \,?\, a$ and $q \rightarrow_a$, then by our argument above, $p' \cdot q$ is ambiguous; hence $p \cdot q$ is too.

$$p \cdot q \checkmark = p \checkmark \wedge q \checkmark \wedge \neg \exists a \, (p \,?\, a \wedge q \rightarrow_a)$$

Finally, we take replication p^*. Informally, p^* is a sequence $p \cdot p \cdot \cdots \cdot p$ of ps, so it should be enough to check that $p \cdot p$ is unambiguous. This, though, is slightly too restrictive: the process $(a \mapsto z + \varepsilon)^*$ is unambiguous (only rule $*$ can ever apply) but we would reject it. The first reduction of p^* must be rule $*$, so it cannot be ambiguous unless p is ambiguous. Therefore, we find all one-step reductions $q \cdot p^*$ of p^*, and check that for all of *those*, $q \cdot p$ is unambiguous:

$$p^* \checkmark = p \checkmark \wedge \neg \exists a \exists b \exists q \, (p \rightarrow_a q \wedge q \,?\, b \wedge p \rightarrow_b)$$

We must also be able to say whether a overlaps p, according to the definition of overlapping that we gave earlier. We have a number of simple structural rules:

$$\frac{p \,?\, a}{p \mid\mid q \;?\; a} \;(\mid\mid\text{L}) \qquad \frac{q \,?\, a}{p \mid\mid q \;?\; a} \;(\mid\mid\text{R}) \qquad \frac{p \,?\, a}{p^* \;?\; a} \;(*\text{-INNER})$$

$$\frac{p \,?\, a}{p + q \;?\; a} \;(+\text{L}) \qquad \frac{q \,?\, a}{p + q \;?\; a} \;(+\text{R}) \qquad \frac{q \,?\, a}{p \cdot q \;?\; a} \;(\text{ThenR})$$

We also have a couple of "nearly" structural rules. Since p^* is always accepting, if $p \rightarrow_a$ then $p^* \;?\; a$. And if q is accepting, then $\mathcal{L}(p) \subseteq L(p \cdot q)$, so if $p \,?\, a$ then $p \cdot q \;?\; a$:

$$\frac{p \,?\, a \qquad q \text{ is accepting}}{p \cdot q \;?\; a} \;(\text{ThenL}) \qquad \frac{p \rightarrow_a}{p^* \;?\; a} \;(*\text{-OUTER})$$

Finally, $p + q$ can introduce an overlap, if p is accepting and $q \rightarrow_a$ or vice versa:

$$\frac{p \rightarrow_a \qquad q \text{ is accepting}}{p + q \;?\; a} \;(+\text{LR}) \qquad \frac{p \text{ is accepting} \qquad q \rightarrow_a}{p + q \;?\; a} \;(+\text{RL})$$

Our ambiguity detection is both sound and complete. Because of soundness, we never accept an ambiguous specification; because of completeness, when we reject a specification we can give a trace showing that it is ambiguous. The proof of soundness and a sketch of completeness are found in the tech. report [13].

6 From Process Calculus to Mocking Framework

The goal of this section is to turn the process calculus into a fully-fledged mocking framework. A basic implementation is simple. We first check that the mocking specification p is unambiguous. To execute p, we wait for the system under test to make a call a. We check if $p \rightarrow_{a,z} q$ for some q; if not, the call is erroneous. Otherwise, we return the result z to the caller, and continue by executing q. Finally, when the test finishes, we check that the final process is accepting.

6.1 Matching

In our examples so far, an event specifies a single concrete call such as $update\_display(speed, 21)$ and a concrete result like 5.833. In reality, we do not always know the function arguments so precisely, and need a richer event language. In our framework, an event specifies a *pattern* of function calls. For example, we may write $update\_display(speed, \_)$, where the "_" is a wildcard; this matches any call to *update_display* where the first argument is *speed*. A pattern simply stands for any of the concrete calls which it matches.

We also allow the event's return value to depend on the call arguments. The user can associate an *evaluation function* with each event, which is given the call's concrete arguments and computes the return value.[4] Note that each occurrence of an event in the mocking specification can have a different evaluation function: the same call need not always return the same result. An event that returns a constant result is a degenerate case where the evaluation function ignores its arguments.

We need to be careful that we can still execute mocking specifications that use pattern matching, and check them for ambiguity. Executing the specification is not a problem: we only need to be able to check if a concrete call matches a particular event. Given a process p and a call c, we check if there is an event that p can consume and which matches c. Finally, we use the evaluation function associated with the event to calculate the return value, and reduce p.

We can also check the specification for ambiguity, as long as we can tell whether any two events intersect. (Two events intersect if there is a single concrete call that matches both of them.) It will help to write out the existing rules, using equality explicitly whenever we compare the events of two processes:

$$
\begin{aligned}
p + q \checkmark &= p \checkmark \wedge q \checkmark \wedge \neg\exists a \exists b\, (p \rightarrow_a \wedge\, q \rightarrow_b \wedge\, a = b) \\
p \mid\mid q \checkmark &= p \checkmark \wedge q \checkmark \wedge \neg\exists a \exists b\, (a \in \mathrm{alphabet}(p) \wedge b \in \mathrm{alphabet}(q) \wedge a = b) \\
a \mapsto z \checkmark &= \mathrm{true} \\
\varepsilon \checkmark &= \mathrm{true} \\
p \cdot q \checkmark &= p \checkmark \wedge q \checkmark \wedge \neg\exists a \exists b\, (p\,?\,a \wedge q \rightarrow_b \wedge\, a = b) \\
p^* \checkmark &= p \checkmark \wedge \neg\exists a \exists b \exists c \exists q\, (p \rightarrow_a q \wedge q\,?\,b \wedge p \rightarrow_c \wedge\, b = c)
\end{aligned}
$$

[4] This is why we could not tell if two events have the same return value in Section 5.

Now, instead of checking if two events are equal, we need to check if they intersect. All we have to do is replace each occurrence of "$a = b$" above with "a and b intersect"! This gives a sound and complete ambiguity detection algorithm for our mocking language with patterns. We will, for example, consider $update\_display(speed, 36) \mapsto () + update\_display(speed, \_) \mapsto ()$ to be ambiguous.

For now we have only implemented quite basic matching. In particular, we can match each argument against either a constant or the wildcard "_"; these were all that we needed for the AUTOSAR testing. However, it is easy to add more powerful patterns, provided they meet the two requirements above. For example, we could easily add value ranges ("x must be between 0 and 200").

6.2 Efficient Implementation

For our AUTOSAR testing we implemented the mocking framework in C. We could simply have implemented the reduction rules of the process algebra, but then reduction would need to allocate memory. We wanted to allocate all memory before running the test, and to avoid heavy term manipulation while testing.

An obvious choice is to translate the mocking specification to a finite-state automaton. Unfortunately, the $||$ operator suffers from exponential blowup: an automaton that implements $p \,||\, q$ needs to remember "how far" it has got in both p and q, so the number of states it needs is the *product* of the number of states in p's automaton and q's automaton.

Instead, we keep the terms of the process calculus but *augment* them with flags that record how far execution has got. During test execution we need only update the flags and not modify the structure of the terms.

For example, we annotate the sequential composition $p \cdot q$ with the flag "left". This indicates that we are reducing p. When we apply rule ThenR to start reducing q, we change the flag to "right", and from then on we ignore p and treat the composition as if it were just q. This gives us the following rules for the augmented "$\cdot$" operator:

$$\frac{p \;\to_{a,z}\; q}{(p \cdot r)_{\text{left}} \;\to_{a,z}\; (q \cdot r)_{\text{left}}}\;(\textsc{ThenL}) \qquad \frac{p \text{ is accepting} \qquad q \to_{a,z} r}{(p \cdot q)_{\text{left}} \;\to_{a,z}\; (p \cdot r)_{\text{right}}}\;(\textsc{ThenR})$$

$$\frac{q \;\to_{a,z}\; r}{(p \cdot q)_{\text{right}} \;\to_{a,z}\; (p \cdot r)_{\text{right}}}\;(\textsc{ThenR2})$$

Notice that we no longer change the structure of the term, we only change the flag. The first two rules correspond exactly to the rules we had before; the third one is an extra structural rule that arises because we can no longer get rid of p once we have finished reducing it.

Here is how we augment the other constructs:

- For alternation, $p + q$, we add a flag that records which alternative, p or q, we have chosen. It is initially "neither". If we make a p-transition it becomes "left", and we ignore q from then on, and vice versa.

– We do not need to augment $p \mid\mid q$, though p and q themselves are augmented. The reduction rules are the same as before.
– We augment a single event, $a \mapsto z$, with a flag that indicates whether we have performed the event. If the flag shows that we have already performed the event, we may no longer perform it.

Replication is the trickiest case, because in executing p^* we may execute p an unlimited number of times. To handle this we need to be able to *reset* a term, which sets its flags back to their initial state. Whenever p in p^* does not accept an event $a \mapsto z$, but does accept the empty trace, we reset p and feed $a \mapsto z$ to it; this corresponds to unrolling p^* in the original semantics. We also augment p^* with a flag that records whether we have performed any reductions on it; this flag is set after the very first reduction, and allows us to model the fact that p^* always accepts the empty trace.

6.3 Extensions

The mocking language we have presented so far is quite minimal. When writing mocking specifications in practice we use a larger repertoire of constructs. Constructs we've found useful include permutations, optional behaviours and finite repetition. The permutation construct operates on a list of behaviours and is similar to parallel composition but doesn't allow interleaving of behaviours: the behaviours must execute one after another, but in an arbitrary order.

Constructs like these are definable in the language we've already presented. For example, an optional p is simply $p + \varepsilon$. However, in our implementation we've added them as primitives for reasons of efficiency. It is particularly important to have permutations be a primitive in the implementation since its encoding into our calculus causes an exponential blow-up in the size of the process.

As an example of using permutations consider the example with parallel composition from Section 3:

$$\begin{aligned}&(\mathit{read\_speed} \mapsto 5.833 \cdot \mathit{update\_display}(\mathit{speed}, 21) \mapsto ())\\ \mid\mid\ &(\mathit{read\_light} \mapsto 6 \cdot \mathit{light\_display} \mapsto ())^*\\ \mid\mid\ &((\mathit{read\_battery} \mapsto 234 \cdot \mathit{update\_display}(\mathit{battery}, 70) \mapsto ()) + \varepsilon)\end{aligned}$$

This specification allows all values to be read before any updates are performed. This might be exactly the freedom one wishes to express. However, suppose that we wish to ensure that the calls to *read_speed* and *update_display* should happen in immediate sequence without being interrupted by any of the other calls, and likewise with the calls to *read_battery* and *update_display*. We can achieve this by using permutations instead of parallel composition, writing the permutation of p, q and r as $\mathit{perm}[p, q, r]$, as follows:

$$\begin{aligned}&\mathit{perm}[\mathit{read\_speed} \mapsto 5.833 \cdot \mathit{update\_display}(\mathit{speed}, 21) \mapsto (),\\ &\quad(\mathit{read\_light} \mapsto 6 \cdot \mathit{light\_display} \mapsto ())^*,\\ &\quad(\mathit{read\_battery} \mapsto 234 \cdot \mathit{update\_display}(\mathit{battery}, 70) \mapsto ()) + \varepsilon]\end{aligned}$$

The components we are testing are currently single-threaded, but provided our implementation of mocking is thread-safe then there is no reason not to use it with multi-threaded code—we would synchronise on each mocked call, thus establishing a sequential order of calls. It is likely that multi-threaded code would require mocking specifications to use the || operator to handle the inevitable non-determinism in the order of mocked calls, but our framework supports this.

7 Mocking in the AUTOSAR Testing Project

Our mocking semantics arose out of a recent project testing AUTOSAR Basic Software components [3] for Volvo Cars [14]. We modelled around twenty AUTOSAR components using an earlier version of the mocking framework. These included the layers of protocol stacks for CAN, LIN, and FlexRay, a router and some diagnostic components. Each component corresponded to an approximately 150-page written specification. Our testsuite has been used to check a handful of implementations from Volvo Cars' subcontractors.

We modelled the AUTOSAR software components in QuickCheck [2], a model-based testing tool that can automatically create random test cases from state machine specifications written in a domain-specific functional language. During the project we developed our mocking framework and integrated it with QuickCheck so that for each generated test case appropriate mocks were also generated. The complexity of the mocking generators varied wildly: from a single line of code to several hundred for the most complicated function we tested. To be able to write these complex mocking generators it was absolutely essential to have a compositional mocking framework where specifications can easily be combined. The simple and clear semantics is also crucial to be able to understand complex mocking generators.

One of the particular challenges with modelling AUTOSAR is that it does not always completely specify the behaviour of the software. Not only may the mocked functions behave in a number of ways, components also have some freedom in which mocked functions to call and how often they are called. And sure enough, whenever the specification allowed for some leeway we found that implementations typically differed in behaviour. The expressiveness of our mocking framework proved invaluable for developing mocking specifications which could handle all legal behaviours mandated by the standard.

8 Related work

It would be natural to compare our work to existing C mocking frameworks. However, there does not seem to be very many, and the few that exist (like *CMock* [7] and *Cgreen* [5]) have very limited functionality. Instead, we compare to *Google Mock*. Google Mock provides mocking functionality for C++ and is feature-wise close to *jMock* [11] and *EasyMock* [8] for Java. Thus it should, to the best of our knowledge, be representative of modern mocking frameworks.

Since C has no objects, we will simply compare the expressiveness of the two approaches.

The main difference is that Google Mock provides lots of default behaviour: expectations are put in parallel by default, there are default return values, etc. The language we define has no default behaviour—everything is explicit. Both approaches have their merits, but hidden defaults require a well-educated user. In terms of expressiveness, we have observed three key differences:

- Google Mock has state, i.e. one action may set a variable that can be read by a later action. This is not included in our language since we have not had the need for it. It would be possible to extend our language with state, but the more interesting question is why we haven't had the need for it. We believe the reason is the compositionality and expressiveness of our mocking language. Compare to writing a regular expression and implementing an equivalent state machine. Regular expressions provide a declarative and compositional interface without the need for state which is much simpler to use than having to maintain the state of the state machine explicitly.
- Google Mock only does *replication* of single events; it is not possible to repeat, for example, a sequence of calls. In our particular use case, L^* is a central ingredient, thus not having it would have presented a problem to us.
- Finally, there does not seem to be a way to express $p + q$ in Google Mock. One could say `atMost(1)` for both p and q, but that would not catch the case when neither or both are called. Again this is central to our use case, but perhaps one often manages without it in ordinary unit testing.

An area closely related to mocking is runtime monitoring. In particular, Jass [4] allows monitoring of "trace assertions" expressed in a CSP-like language; if the monitored code performs an event in the alphabet of the process that is not part of any trace, then an exception is raised. The trace assertion language is described by example and formal properties are not stated or proven. In general, run-time monitors can allow non-determinism in the monitor, because this cannot lead to non-determinism in the test outcome. Because mocking supplies return values to the code under test, then non-deterministic mocking will lead to non-deterministic test outcomes. Similarly, model checkers can allow non-deterministic environments since they can explore branching executions, collect constraints, and use solvers to find interesting cases: since each test execution can follow only one branch then we do not enjoy the same freedom.

Our mocking language shares many similarities with the language PSL [1], used by the hardware community for specifying and verifying circuits. PSL is divided into several layers and one of these layers is a modelling layer, used for specifying parts of the design which are not yet implemented. Although similar in spirit to our language, PSL's mocking language naturally differs on many details as it targets hardware, not software.

9 Conclusions

This paper provides a fresh look at mocking and presents a new expressive and compositional semantics. It is the first such semantics we are aware of; other mocking frameworks have a precise semantics, but only defined by their implementation. The expressiveness is inspired by a large use case of mocking in a model-based testing project in the automotive software domain, but the solution is generally applicable in other domains as well.

Since we have a formal semantics for mocking, we can check mocking specifications for ambiguity. We prove that this verification is sound and complete. Thus, whenever we accept a user-defined mocking specification, the result is unambiguous and if the specification is unambiguous, we accept it. Unambiguous specifications are important because a mocking framework must either make arbitrary choices or random choices in the face of ambiguity; the first leads to surprising behaviour and the second to unrepeatable tests. The formal semantics also makes it clear that our feature set is orthogonal.

Acknowledgements. This research was sponsored by EU FP7 Collaborative project *PROWESS*, grant number 317820.

References

1. Property specification language. IEEE Standard 1850, 2005.
2. T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the ACM SIGPLAN Workshop on Erlang*, New York, NY, USA, 2006. ACM Press.
3. AUTOSAR consortium. AUTomotive Open System ARchitecture specifications. http://www.autosar.org.
4. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.
5. Cgreen. http://www.lastcraft.com/cgreen.php.
6. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.
7. CMock. https://github.com/ThrowTheSwitch/CMock.
8. EasyMock. http://www.easymock.org.
9. Google C++ mocking framework. http://code.google.com/p/googlemock.
10. Google C++ testing framework. http://code.google.com/p/googletest.
11. jMock. http://jmock.org/index.html.
12. Mockito - simpler & better mocking. http://code.google.com/p/mockito.
13. J. Svenningsson, H. Svensson, N. Smallbone, T. Arts, U. Norell, and J. Hughes. An expressive semantics of mocking. Technical Report 2014:01, ISSN 1652-926X, Computer Science and Engineering, Chalmers University of Technology, 2014.
14. R. Svenningsson, R. Johansson, T. Arts, U. Norell, J. Svenningsson, and H. Svensson. Testing AUTOSAR software components with QuickCheck. In *Proceedings of IXe Conf. on AMCTM*. SP, Sweden, 2011.