CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Bidirectional Testing of Communicating Systems

Master's thesis in Computer Science

MAXIMILIAN ALGEHED

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Master's thesis 2017

Bidirectional Testing of Communicating Systems

Maximilian Algehed

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Bidirectional Testing of Communicating Systems
Maximilian Algehed

Supervisor: Koen Claessen, Computer Science and Engineering
Examiner: Mary Sheeran, Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Bidirectional Testing of Communicating Systems
Maximilian Algehed
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This report presents a new tool called SessionCheck. This tool helps programmers write distributed applications that work correctly. SessionCheck is designed to help rid programmers of the tedium of maintaining more than one specification and test suite for multiple application components. SessionCheck does this by borrowing ideas from session types [14] and domain specific languages in order to provide a simple yet expressive and compositional specification language for communication protocols. Specifications written in the SessionCheck specification language can be used to test both client and server implementations of the same protocol in a completely language-agnostic manner.

Keywords: SessionCheck, Testing, Communication Protocols

Acknowledgements

First and foremost I would like to thank my supervisor, Koen Claessen, for his insightful comments on this work and most of all for his timely advice on when to stop hacking and start writing. I would also like to thank Andreas Ekeroot, for inspiring this work after an especially intense lemonade drinking session at the student pub, and Sólrún Halla Einarsdóttir, for always being great fun around the office.

Maximilian Algehed, Gothenburg, November 2017

Contents

Chapter 1

Introduction

Imagine you wanted to build a client-server application for storing files in the cloud. The functionality, at least initially, of the application is simple. The client, running on the user's computer either as a standalone application or a web-page, should allow the user to upload files to the server where the files will be stored for later retrieval. In this scenario your client and server parts of the application would most likely be two different programs, perhaps written in two different programming languages. In order for the two programs to work together, they need to co-ordinate their operation using some form of communication protocol. The first protocol you design allows the application to only provide functionality for file retrieval, allowing the client to retrieve files stored on the server. A simple sketch of the communication protocol looks something like this:

```
client: Send login information
server: Send ok/not ok
client: Send filename
server: Send file contents
```

Having established how the client and server are supposed to interact, the next question is how to ensure that your two programs are correct, and, more importantly, that the entire application works as expected when they are put together. Maybe you would write a few test cases, perhaps using something like the mocking combinators of Svenningsson et al. [29], which let you simultaneously specify the accepted behaviour of one component while giving an example of the other's. In this notation a test case for the client would look similar to the example below.

$$\mathit{sendLogin}\ (\mathit{username}, \mathit{password}) \mapsto \mathit{ok}\ .\ \mathit{sendFilename}\ (\mathit{filename}) \mapsto \texttt{"Hello, test."}\ .\ \epsilon$$

If you carefully construct your test cases they should guarantee that if the server and the client both pass their respective tests in isolation, then they will work flawlessly together. Or rather, any errors in the application will not be down to errors in the communication protocol implementation.

Having implemented and thoroughly tested your application you might feel satisfied with the process you have gone through to get here. The protocol was carefully specified and test cases were designed to exercise your application components to their fullest. But what happens when, later on, you want to add new functionality to your application? Say you wanted to add functionality which lets the client retrieve the files she has stored in the could. The first thing you need to do is to update the communication protocol. The new version of the specification reads something like this:

```
client: Send login information
server: Send ok/not ok
client: Either send a filname to download, or a file to upload
server: Either send some file contents or send a status message
```

Next you have to change some, if not all, of your test cases to work with the new version of your application. It is not unlikely that you end up getting one or more of the new test cases wrong, or at least that the labour required is on par with the labour required to change the actual application code you are testing. Writing test cases by hand is hardly the ideal solution to the problem of specifying and verifying communicating code. The approach is susceptible to error as any inconsistency between the client and server test cases means that code which is correct with respect to its tests may still not work correctly with the rest of your application. Furthermore, as argued above, it can be a labour intensive process to keep specifications up to date.

Technology like QuickCheck [9] exists to reduce the labour involved in specifying and testing code by prompting the programmer to write code which generates thousands or sometimes millions of random test cases to test specific functionality. QuickCheck may be used together with the type of mocking library mentioned previously to significantly reduce the labour involved in testing communication protocols. However, this approach traditionally means we end up with two different QuickCheck specifications, one for the server and one for the client. As a consequence it is still not possible to guarantee that if the client and the server pass their tests in isolation, they will work correctly when put together. What's more, a specification might be inconsistent with itself, there is nothing guaranteeing that your two specifications, even when correct with respect to each other, can not deadlock, making it impossible for one party to satisfy the protocol under all circumstances.

In this report we present our tool SessionCheck. In essence, SessionCheck extends QuickCheck to solve both of these problems. With SessionCheck, maintaining multiple facets of the same specification is a thing of the past! In SessionCheck each specification is written only once, either from the point of view of the client or the server. From the point of view of the client the specification above would look something like the code below.

```
protocol :: Spec MessageType ()
protocol = do
  send anyLoginData
  ok ← get anyBool
  when (¬ ok) stop
  choice ← choose ["upload", "download"]
  case choice of
    "upload" → do
      send anyFile
    "download" → do
      send fileName
      get fileContents
```

The specification above can be used to derive multiple test cases for the client and the server implementation in our example. Furthermore, SessionCheck specifications can be automatically tested to ensure they are coherent, reducing the possibility for the problem alluded to above. Concretely, this report makes the following contributions.

- We present the SessionCheck specification language and show examples of how it may be used to specify a variety of protocols.
- We present a method for making SessionCheck specification bi-directional. The same specification in effect specifies both the client and server behaviour in a protocol.
- We show how to use SessionCheck to find faults in communication protocols before writing a single line of implementation code.
- We show how to use the SessionCheck tools to derive test-suites for both a client and a server implementation of a protocol using a single SessionCheck specification.
- We use SessionCheck to specify the SMTP [23] protocol and test example code taken from popular SMTP libraries for the Python programming language.

Chapter 2

SessionCheck in Action

A SessionCheck specification can be used in two different ways. The specification may be used to test implementations of the protocol end-points. It is possible to use the same SessionCheck specification to test both a client and a server implementation of a protocol. This mode of operation is illustrated in Figure 2.1 As seen in the figure, both client and server code may be individually tested, and should they both be deemed correct (O.K.) by SessionCheck, we expect them to work correctly together. SessionCheck can also test if a specification is coherent, that locally correct choices made by protocol end-points do not make it impossible for either party to satisfy the obligations of the protocol. This mode of operation is illustrated in Figure 2.2. In effect, the specification **S** is made to test itself. This chapter explains the use of SessionCheck as a language and a tool by giving examples of both modes of operation mentioned above.

2.1 Checking that protocols are coherent

To understand the idea of incoherent protocols consider the following protocol specification

```
protocol :: Int ⊑ t ⇒ Spec t Int
protocol = do
  a ← send anything
  b ← send anything
  get (inRange a b)
```

The first line is a type signature, telling us that *protocol* is a specification over a channel where messages of type *Int* can be transmitted ($Int \sqsubseteq t$). The protocol requires one end-point to transmit two *Int*s, a and b, and the other end-point to respond with an *Int* which is in the range $[a, b]$. The problem with the protocol is that local choices at the first two *send anything* actions may

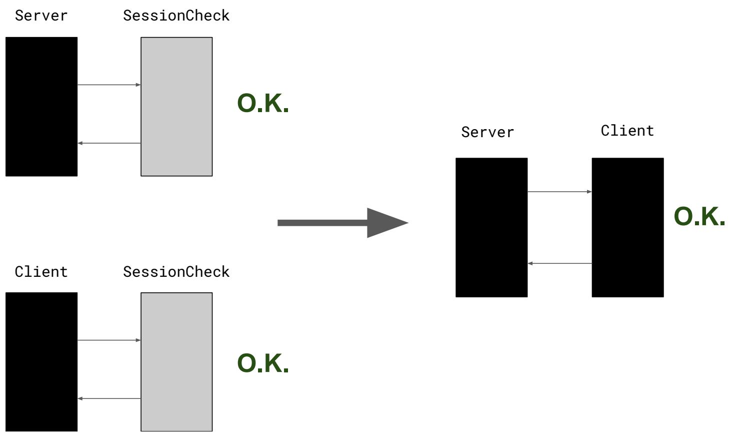

Figure 2.1: Using SessionCheck to test both client and server implementations of the same protocol.

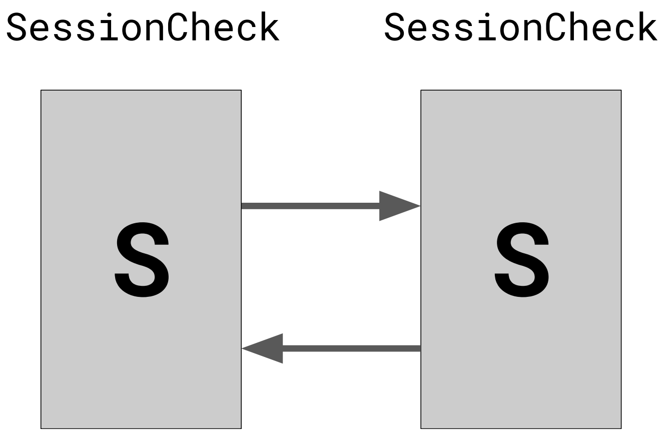

Figure 2.2: Checking that a specification is coherent.

result in the protocol end-point performing the *send*s making it impossible for the other end-point to be compliant with the protocol. If we run *checkCoherent protocol* SessionCheck will print the following:

```
Failed with:
Timeout: Timeout on {get (inRange 22 -30)} and channel is
         dead with reason:
         "Timeout: Failed to satisfy {inRange 22 -30}"

With trace:
Output {22}
Output {-30}
```

Telling us that if the messages 22 and -30, both valid messages according to the specification, are sent then the protocol obligation *get* (*inRange* $22 - 30$) times out. We are also provided with an explanation of the timeout, SessionCheck failed to generate a value satisfying the predicate.

2.2 Testing protocol end-points

SessionCheck features a modular back end system which allows the user to test different implementations of the same protocol written using different communication substrates in different languages. At the time of writing SessionCheck supports two different back ends, one for testing Erlang programs by incorporating SessionCheck into the existing BEAM message passing system, and one for testing protocols like SMTP [23] and POP3 [20] which communicate using `<CR><LF>` terminated string messages transmitted over TCP.

As an example of testing protocol end-points we consider the following protocol, called *echo*:

```
echo :: String ⊑ t ⇒ Spec t String
echo = do
  msg ← get anything
  send (is msg)
```

To test the Python [31] implementation in Figure 2.3 of the *echo* protocol we will use SessionCheck's TCP backend. The function *tcpMain* :: *Mode* → *String* → *PortNumber* → *Spec TCPMessage a* → *IO* () takes as arguments the role which SessionCheck is to take (*Client* or *Server*), the shell command which launches the program being tested (`"python EchoServer.py"` in our example), the IP port number which will be used for communication, and the specification of the protocol implemented by the system under test.

```
main :: IO ()
main = tcpMain Client "python EchoServer.py" 10000 echo
```

```
import socket

HOST = 'localhost'
PORT = 10000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
while True:
    conn, addr = s.accept()
    data = conn.recv(1024)
    conn.sendall(data)
```

Figure 2.3: The `EchoServer.py` implementation of the (*dual echo*) protocol

As the implementation in Figure 2.3 is correct running the piece of code above will make SessionCheck print `Ok, passed 100 test`, telling us that the protocol was tested 100 times without protocol violation. If we alter the implementation of `EchoServer.py` to make it buggy, by inserting the statements

```
if data == "":
  data = "BUG!"
```

before the line `conn.sendall(data)`, and run SessionCheck again we get the following output:

```
Failed with:
Bad: get {is ""}

With trace:
Output {""}
InputViolates {is ""} "BUG"
```

SessionCheck now tells us it has found a bug in the implementation of the server. It tells us that after producing the output (sending) the message containing the empty string, it received back the string `"BUG"`, rather than the expected empty string.

To see another example of testing server code we will briefly explore using SessionCheck to test an Erlang [8] implementation of a small protocol describing an online book shopping service. The protocol we will be working with specifies the interaction between the client and server side of a service where the user, who is communicating with the server using the client program, maintains a shopping basket of books to be purchased. The SessionCheck specification can be seen below:

```
bookShop :: (Int ⊑ t, String ⊑ t, [Int] ⊑ t) ⇒ Spec t ()
bookShop = bookShopLoop []
bookShopLoop :: (Int ⊑ t, String ⊑ t, [Int] ⊑ t) ⇒ [Int] → Spec t ()
bookShopLoop books = do
  choice ← choose ["buy", "request", "stop"]
  case choice of
    "buy" → do
      book ← send anyInt
      bookShopLoop (book : books)
    "request" → do
      get (permutationOf books)
      bookShopLoop books
    "stop" → do
      stop
```

The client begins by choosing one of three actions (*choice* ← *choose*...), to "buy" a book, to "request" to see their basket, or to "stop", terminating the communication. The behavioural specification is simple, if the client buys a new book that book is added to the basket and if the client wishes to see their basket they should receive a list of books which is a permutation of their current basket. The **case** *choice* **of** expression gives the specification for each choice the client has. If the client has chosen to "buy" a new book, the client must transmit the book identifier (an *Int*), and the protocol resumes with the book added to the basket (*bookShopLoop* (*book* : *books*)). If the client has chosen to "request" to see their current basket, the client should receive a permutation of the current basket from the server (*get* (*permutationOf books*)). Finally, if the client has decided to terminate the communication "stop" the session ends with *stop*.

The Erlang implementation of the server can be seen in Figure 2.4. Using SessionCheck we can test the implementation against the specification using *erlangMain* "bookShop:main" *bookShop*. Running this program SessionCheck finds the following trace demonstrating a bug in the implementation of the Erlang server.

```
Failed with:
Bad: {get permutationOf [3, -25]}

With trace:
Output {ErlString "buy"}
Output {ErlInt 3}
Output {ErlString "buy"}
Output {ErlInt (-25)}
Output {ErlString "request"}
InputViolates {permutationOf [-25]} {ErlList [ErlInt 3]}
```

What we have is an inconsistency between the specification and the implementation of the protocol. There is good reason to consider the ”bug” to be in the specification rather than in the implementation, as there is no good reason why a book identifier should be a negative integer.

```
main() -> loop([]).

loop(Books) ->
  receive
    {Hs, "buy"} ->
      receive
        {Hs, B} -> if B >= 0 -> loop([B | Books]);
                      true   -> loop(Books)
                   end
      end;
    {Hs, "request"} -> Hs ! Books, loop(Books);
    {Hs, "stop"} -> exit(done)
  end.
```

Figure 2.4: An Erlang implementation shell for the *bookShop* server

Fixing the bug in the specification is a simple case of swapping out the *anyInt* predicate in the line *book* ← *send anyInt* for the *posInt* predicate.

2.3 Shrinking

The counterexample demonstrating the fault in the book shop server implementation above was needlessly complicated. While it did demonstrate that the number −25 was not included in the list of books the client received after sending a `"request"` message, it also included the irrelevant messages `"buy"` and 3. In this example the bug is easily spotted, but generally counterexamples containing redundant information, like the two messages `"buy"` and 3 above, make it more difficult to find the cause of the failure. Following QuickCheck, SessionCheck produces minimal counterexamples by means of shrinking existing ones. When asked to shrink the counterexample above SessionCheck will produce the following:

```
Failed with:
Bad: get {permutationOf [-1]}

With trace:
Output {ErlString "buy"}
Output {ErlInt -1}
Output {ErlString "request"}
InputViolates {permutationOf [-1]} {[]}
```

While the example above demonstrates that SessionCheck is indeed capable of finding minimal counterexamples, the algorithm is far from perfect. If we alter the bug in the implementation of the server to instead of discarding negative book identifiers discarding all book identifiers if the basket is not empty we may get the following counterexample:

```
Failed with:
Bad: get {permutationOf [-1, 22, 13]}

With trace:
Output {ErlString "buy"}
Output {ErlInt -1}
Output {ErlString "buy"}
Output {ErlInt 22}
Output {ErlString "buy"}
Output {ErlInt 13}
Output {ErlString "request"}
InputViolates {permutationOf [-1, 22, 13, -10]} {[-1]}
```

In this case SessionCheck does not always manage to shrink it down to the minimal failing test case, which would be the following:

```
Failed with:
Bad: get {permutationOf [0, 0]}

With trace:
Output {ErlString "buy"}
Output {ErlInt 0}
Output {ErlString "buy"}
Output {ErlInt 0}
Output {ErlString "request"}
InputViolates {permutationOf [0, 0]} {[0]}
```

Instead, we sometimes get counterexamples containing more than two `"buy"` messages as well as not all book identifiers being 0. However, in our experience SessionCheck does produce what one might call "small-ish" counterexamples after shrinking.

In essence, there are two reasons why SessionCheck's shrinking does not always produce the minimal counterexample. The first reason is that local changes, sending a `"buy"` instead of a `"request"`, change the rest of the protocol and any information that can be obtained from a previous failing counterexample may no longer be relevant to the current run. The second reason is that the higher-order nature of SessionCheck means that while a specification might have implicit states and two different *send*s can be said to be "the same" *send*, there is no way to observe this in the implementation of SessionCheck (see Chapter 4 for details). Improvements to the shrinking facilities in SessionCheck are noted as future work.

Chapter 3

The SessionCheck specification language

We begin with an example. Consider a simple protocol between a server and a client which requires the client to transmit two positive integers and receive back from the server their sum. In SessionCheck, this specification would be written as follows:

```
protocol :: Spec Int Int
protocol = do
  a ← send posInt
  b ← send posInt
  get $ is (a + b)
```

SessionCheck specifications are written from the point of view of a particular party, in this case the

```
send   :: a ⊑ t ⇒ Predicate a → Spec t a
get    :: a ⊑ t ⇒ Predicate a → Spec t a
stop   :: Spec t a
fail   :: String → Spec t a
return :: a → Spec t a
(≫=)   :: Spec t a → (a → Spec t b) → Spec t b
  -- Derived combinators
choose :: (Eq a, a ⊑ t) ⇒ [a] → Spec t a
branch :: (Eq a, a ⊑ t) ⇒ [a] → Spec t a
```

Figure 3.1: The SessionCheck specification language

client. We may as well have written the same specification from the point of view of the server, in which case it would look like this:

$$\begin{aligned}&\mathit{protocol}' :: \mathit{Spec}\ \mathit{Int}\ \mathit{Int}\\&\mathit{protocol}' = \textbf{do}\\&\quad a \leftarrow \mathit{get}\ \mathit{posInt}\\&\quad b \leftarrow \mathit{get}\ \mathit{posInt}\\&\quad \mathit{send}\ \$\ \mathit{is}\ (a+b)\end{aligned}$$

It is no coincidence that the two specifications are very similar, and SessionCheck can work equally well with both, as we will see shortly.

The SessionCheck specification language is a domain specific language embedded in the Haskell [16] programming language. Being an embedded language means that the language primitives in SessionCheck are implemented as Haskell data types and functions. The language primitives in SessionCheck can be seen in Figure 3.1. The type argument t in the type *Spec* t a denotes the type of messages being delivered on the channel on which the system under test is communicating with SessionCheck. The *send* and *get* primitives represent obligations for the respective party to send a message which is compliant with the given *Predicate*, more on this soon. The constraint $a \sqsubseteq t$ denotes a subtyping relation between a and t. That is, any value of type a is also a value of type t, and a value of type t may be a value of type a. Included in the interface are also the *fail* and *stop* functions. The *stop* primitive specifies that the protocol session is terminated. Unlike *send*, *get*, and *stop*, *fail* does not directly correspond to an action in the protocol. Rather it allows the user to specify conditions for when the system being tested by SessionCheck fails which are not directly coupled to direct constraints on messages. The *choose* and *branch* primitives are not actually primitive operations, rather they are derived from the rest of the interface. The specification *choose xs* reads ”send one of the values in *xs*”, while *branch xs* reads ”get one of the values in xs”.

The primitives described above do not permit small specifications, like *send anyInt* and *get anyBool*, to be composed to form more complex specifications. For this purpose SessionCheck also supports the standard *Monad* interface [32], which contains the two primitive operations *return* and $(\gg=)$ (pronounced ”bind”). The bind operator allows specifications to be composed by taking a specification, s, and a function which creates a specification from a value, f, and composing them to form the specification $s \gg= f$ which means ”first the protocol behaves like s, then whatever value is produced at the end of s is fed to f to produce a new specification to follow”. As an example, consider the case where $s = \mathit{send}\ \mathit{anyInt}$ and $f\ x = \mathit{send}\ (\mathit{greaterThan}\ x)$, here $s \gg= f$ is a specification which first requires the end-point to send an *Int* and then to send another *Int* which is greater than the first one. The **do**... notation in the above examples are syntactic sugar for successive uses of $(\gg=)$ and lambda abstraction $\lambda x \to e$, where the expression $\lambda x \to e$ denotes a function where the variable x is used to bind the input of the function in the output e. When written using explicit $(\gg=)$ the specification of *protocol* would look like this:

```
data Predicate a = Predicate { apply :: a → Bool
                             , gen   :: Gen a
                             , name  :: String }
```

Figure 3.2: The *Predicate* type

```
anything  :: Arbitrary a ⇒ Predicate a
is        :: (Show a, Eq a) ⇒ a → Predicate a
anyOf     :: [Predicate a] → Predicate a
pairOf    :: Predicate a → Predicate b → Predicate (a, b)
unfailing :: (Arbitrary t, a) ⊑ t ⇒ Predicate a → Predicate (Either t a)
bimap     :: (a → b) → (b → a) → Predicate a → Predicate b
```

Figure 3.3: Some *Predicate*s in SessionCheck

```
protocol :: Spec Int Int
protocol =
  send posInt ≫= λa →
  send posInt ≫= λb →
  get $ is (a + b)
```

Supporting the generic *Monad* interface means that we get several useful combinators "for free", like $(\gg) :: \mathit{Spec}\ t\ a \to \mathit{Spec}\ t\ b \to \mathit{Spec}\ t\ b$ which sequences two independent specifications, and $\mathit{forever} :: \mathit{Spec}\ t\ a \to \mathit{Spec}\ t\ a$ which repeats a specification indefinitely. Finally, the *return* function simply wraps a value in a specification. It represents no obligation on either part of the communication protocol, but rather is part of the standard monad interface.

3.1 The *Predicate* type

The *Predicate a* type in Figure 3.2 represents predicates which may be used both to test a condition on a value of type a and to generate a random value satisfying that property. Note that the type variable a appears in both positive and contrapositive position, as witnessed by the *bimap* function. This dual functionality of the *Predicate a* type is crucial in order to make it possible to use the same specification for both mocking and monitoring of protocol end-points at the same time. SessionCheck features multiple combinators for constructing predicates, some of which can be seen in Figure 3.3. The *anything* predicate will accept any value, representing the predicate $p\ x = \mathit{True}$. The *is* predicate combinator is more restrictive, accepting only precisely the value provided to it, *is a* represents the predicate $p\ x = x \equiv a$. Another important predicate is *anyOf*, it takes a list of predicates and accepts any value accepted by at least one of the input predicates.

$$\begin{aligned}
&\mathit{dual}\ (\mathit{send}\ p) &&= \mathit{get}\ p\\
&\mathit{dual}\ (\mathit{get}\ p) &&= \mathit{send}\ p\\
&\mathit{dual}\ (\mathit{fail}\ s) &&= \mathit{fail}\ s\\
&\mathit{dual}\ \mathit{stop} &&= \mathit{stop}\\
&\mathit{dual}\ (\mathit{return}\ a) &&= \mathit{return}\ a\\
&\mathit{dual}\ (s \ggg\!= \lambda x.\ e) &&= \mathit{dual}\ s \ggg\!= \lambda x.\ \mathit{dual}\ e
\end{aligned}$$

Figure 3.4: Duality of specifications

3.2 The *dual* of a specification

One important property of SessionCheck specifications is that each specification admits a *dual*. An equational specification of the *dual* operation can be seen in Figure 3.4. If the role of a specification s is to specify one party in a two party protocol, *dual* s is the symmetric specification of the other party in the same protocol. The cases for *send*, *get*, *fail*, *stop*, and *return* are self-explanatory, however the case for $\ggg\!=$ is interesting. It says that in order to be symmetric to a specification which first requires the behaviour s, the result of which creates a new specification according to g, it is sufficient to first be symmetric to s and to then be symmetric to whatever specification is produced by g. The reason for this is that $\ggg\!=$ encodes sequencing of protocols, therefore if two protocols are sequenced using $\ggg\!=$ the dual should be the dual of each protocol in sequence. If the client does s then t, the server should do *dual* s then *dual* t. This is perhaps simpler to see when considering the $(\gg) :: \mathit{Spec}\ t\ a \rightarrow \mathit{Spec}\ t\ b \rightarrow \mathit{Spec}\ t\ b$ operator, which is defined as $s \gg t = s \ggg\!= \lambda\_ \rightarrow t$. By the definition of *dual* we obtain $\mathit{dual}\ (s \gg t) = \mathit{dual}\ s \gg \mathit{dual}\ t$, which encodes precisely the reasoning above. The definition for $\ggg\!=$ is simply a generalisation of this reasoning to deal with dependency.

One important property to note about *dual* is that it is an involution. That is to say that $\forall\ x.\ \mathit{dual}\ (\mathit{dual}\ x) \equiv x$. The proof of this property is straightforward and we leave it as an exercise for the reader. As simple as this statement is, it informs us that the behaviour of *dual* is what we expect, it describes the inverse of the input protocol and nothing else.

We will now see *dual* in action, recall the *echo* protocol from Chapter 2, given below:

```
echo :: String ⊑ t ⇒ Spec t String
echo = do
  msg ← send anything
  get (is msg)
```

Consider now the dual specification:

$$\begin{aligned}&\mathit{dual\_echo} :: \mathit{String} \sqsubseteq t \Rightarrow \mathit{Spec}\ t\ \mathit{String}\\&\mathit{dual\_echo} = \mathbf{do}\\&\quad \mathit{msg} \leftarrow \mathit{get}\ \mathit{anything}\\&\quad \mathit{send}\ (\mathit{is}\ \mathit{msg})\end{aligned}$$

The structure is preserved but the *send* has been exchanged for a *get* and vice versa. To see why this makes sense we consider *echo* written using explicit $\ggg$ notation:

$$\mathit{echo} = \mathit{send}\ \mathit{anything} \ggg \lambda \mathit{msg} \rightarrow \mathit{get}\ (\mathit{is}\ \mathit{msg})$$

We can compute the *dual* of *echo* to $\mathit{get}\ \mathit{anything} \ggg \lambda \mathit{msg} \rightarrow \mathit{send}\ (\mathit{is}\ \mathit{msg})$ (see below) using the definition of *dual* in Figure 3.4. When written using **do** notation $\mathit{get}\ \mathit{anything} \ggg \lambda \mathit{msg} \rightarrow \mathit{send}\ (\mathit{is}\ \mathit{msg})$ is precisely the definition of *dual_echo* above.

$$\begin{aligned}&\ \ \mathit{dual}\ (\mathit{send}\ \mathit{anything} \ggg \lambda \mathit{msg} \rightarrow \mathit{get}\ (\mathit{is}\ \mathit{msg}))\\&\equiv \mathit{dual}\ (\mathit{send}\ \mathit{anything}) \ggg \mathit{dual} \circ \lambda \mathit{msg} \rightarrow \mathit{get}\ (\mathit{is}\ \mathit{msg})\\&\equiv \mathit{get}\ \mathit{anything} \ggg \mathit{dual} \circ \lambda \mathit{msg} \rightarrow \mathit{get}\ (\mathit{is}\ \mathit{msg})\\&\equiv \mathit{get}\ \mathit{anything} \ggg \lambda \mathit{msg} \rightarrow \mathit{dual}\ (\mathit{get}\ (\mathit{is}\ \mathit{msg}))\\&\equiv \mathit{get}\ \mathit{anything} \ggg \lambda \mathit{msg} \rightarrow \mathit{send}\ (\mathit{is}\ \mathit{msg})\end{aligned}$$

Chapter 4

The implementation of SessionCheck

This chapter presents an overview of the implementation of SessionCheck, focusing on some key design decisions that enable large portions of the software infrastructure to be reused for testing, shrinking, and coherence checking.

4.1 The *Spec* type

The *Spec* type, see Figure 4.1, is implemented as a Generalised Algebraic Data Type (GADT) [28]. The monadic structure of *Spec* is explicitly represented in the type using the constructors *Return* and *Bind*. The technique is originally due to Svenningsson and Svensson [30]. One benefit of this representation of specifications is that the *dual* operation is simple to implement, as can be seen in Figure 4.2. It is trivial to see that this implementation of *dual* is true to the specification in Figure 3.4. In the interest of completeness we also give the definition of the $\sqsubseteq$ type class in Figure 4.3.

4.2 Modularity

The implementation of SessionCheck consists of two primary parts, the evaluation engine and the back end driver. The interaction between the two can be seen in Figure 4.5. In order to implement a new back end for SessionCheck one essentially only needs to provide an instance of the *Interface* data structure in Figure 4.4. The *outputChan* and *inputChan* channels are used to communicate the

```
data Spec t a where
  Get   :: (a ⊑ t, NFData a) ⇒ Predicate a → Spec t a
  Send :: (a ⊑ t, NFData a) ⇒ Predicate a → Spec t a
  Stop  :: Spec t a
  Fail  :: String → Spec t a
  Return :: a → Spec t a
  Bind :: Spec t a → (a → Spec t b) → Spec t b
```

Figure 4.1: The *Spec* type

```
dual :: Spec t a → Spec t a
dual (Get p)      = Send p
dual (Send p)    = Get p
dual Stop          = Stop
dual (Fail s)      = Fail s
dual (Return a) = Return a
dual (Bind s f)  = Bind (dual s) (dual ∘ f)
```

Figure 4.2: The implementation of *dual*

```
class a ⊑ t where
  inj :: a → t
  prj :: t → Maybe a
```

Figure 4.3: The implementation of ⊑

```
data Interface t = IFace { outputChan :: TChan t
                         , inputChan  :: TChan t
                         , isDead     :: MVar ()
                         , isDone     :: MVar ()
                         , run        :: IO () }
```

Figure 4.4: The interface between the SessionCheck evaluation engine and a back end

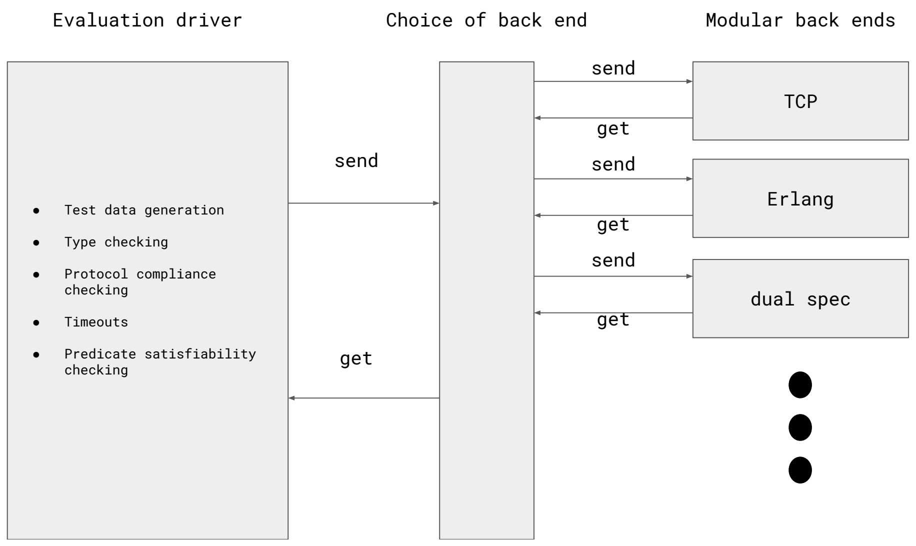

Figure 4.5: The SessionCheck software architecture

values which will be transmitted back and forth between the system under test and SessionCheck. The *isDead* and *isDone* fields are used to communicate that the channel has either been broken or that the specific run of the protocol has been completed. The *run* io-action is the most interesting as it is what is used to start a new session and to manage the communication. When testing simple TCP clients like in the `Echo` example above, this action consists of starting the program under test and connecting to it via a TCP socket.

4.3 Checking coherence

Checking that a protocol is coherent is a straightforward instance of the modularity described above. It works by running the specification against its own dual. In essence, the *Interface* consists of standard values for the channels and communication *MVar*s as well as the *run* action $run = sessionCheck\ (dual\ spec)$. Duality of specifications does not only give a conceptual method by

```
data DynamicShow = DynShow String Dynamic
instance Show DynamicShow where
  show (DynShow s _) = s
instance (Typeable a, Show a) => a ⊑ DynamicShow where
  inj a = DynShow (show a) (toDynamic a)
  prj (DynShow _ dyn) = fromDynamic dyn
```

Figure 4.6: Dynamic values for testing a specification against itself

which it is possible to check for coherence, it also results in a simple procedure by which it can be achieved.

However, one issue is left to resolve. What should the type of messages be? For this we use a variant of the *Dynamic* type [21]. The type *Dynamic* has a single constructor, $Dyn :: \forall\ a \circ a \rightarrow$ $TypeRep\ a \rightarrow Dynamic$, where a is an existentially quanitifed type variable (written $\forall\ a...$) and $TypeRep\ a$ is a concrete representation of the type of a. While there is an instance of *Show* for *Dynamic*, it simply prints the type representation, rather than the value. As a consequence, $show\ (Dyn\ (5 :: Int)\ (typeOf\ 5 :: TypeRep\ Int))$ is `"<<Int>>"`, rather than `"5"`. If we were to simply use *Dynamic* as the type of messages the resulting counterexamples presented by SessionCheck would be on the form:

```
Failed With: ...
Output: {<<Int>>}
Output: {<<String>>}
```

rather than the more instructive:

```
Failed With: ...
Output: {5}
Output: {"Hello"}
```

Our variant of *Dynamic*, called *DynamicShow*, can be found in Figure 4.6. It resolves the issue above by representing messages as both a *Dynamic* value and a *String*. The *String* component of a value of type *DynamicShow* is obtained by using *show* when a value is injected into the type, which happens when the message is sent.

4.4 Shrinking

Shrinking in SessionCheck is significantly more difficult to implement than in traditional property-based testing of pure functions. In essence, the combination of dependency and external choice in

```
shrink fuel spec trace = do
  when (fuel ≡ 0) (abortWithTrace trace)
  trace' ← runTest spec trace
  if length trace' ⩽ length trace then
    shrink (fuel − 1) spec trace'
  else
    shrink (fuel − 1) spec trace
runTest spec trace = do
  if spec `matches` head trace then
    do
      spec' ← step spec (head trace)
      runTest spec' (tail trace)
  else
    do
      spec' ← step spec (lookAhead spec trace)
      runTest spec' (dropLookAhead spec trace)
```

Figure 4.7: The algorithm for shrinking in SessionCheck

SessionCheck specifications makes shrinking more difficult than in the usual setting. Dependency, introduced by the monadic $\gg\!=$ operator, and external choice, introduced by the *get* primitive, mean that the structure of the communication required by the protocol may change and make it impossible to follow the trace which exhibits the bug. Figure 4.7 outlines the algorithm for shrinking in SessionCheck.

The primary function for shrinking is the *shrink* function. It takes three arguments, *fuel*, *spec*, and *trace*. The *fuel* parameter is the maximum number of shrinking attempts to be made, *spec* is the specification of the system under test, and *trace* is the trace witnessing the current smallest counterexample. The interesting function is *runTest*, which runs one complete test of the system, using the *trace* as a guide for the values to *send*. It works by attempting to match the obligation of the specification, a *send* or a *get*, with the first element in the trace. If the first event in the *trace* matches the obligation in the spec, a "sent" event matches *send p* if the sent value is accepted by *p* and likewise for "got" and *get*, the test will take one step attempting to shrink that value (if the obligation is a *send*). Otherwise, the algorithm looks ahead in the trace to find the first, if any, event which matches the specification and proceed from that point in the trace.

Chapter 5

Case study: The SMTP protocol

In this chapter we briefly present our attempt at formalising the minimum required subset of the SMTP protocol, specified in RFC821 [23], in SessionCheck. The SMTP protocol is a client server protocol where the client sends commands like `MAIL FROM:<algehed@chalmers.se>` and `QUIT` and the server replies with status codes like `250 OK` or `500 Syntax Error`. In this case study we focus on specifying the minimal required subset of the protocol. Apart from the `MAIL FROM:` command a minimal implementation also needs to support the following commands

- `HELO` for establishing a handshake between the client and the SMTP server.
- `RCPT TO:` for adding mail recipients.
- `DATA` for starting a mail text transfer.
- `RSET` for aborting a session.
- `QUIT` for exiting a session with success.
- `NOOP` for doing nothing.

The first thing that happens when an SMTP client connects to a server is that the server sends a `220 Service Ready` status code message. The client then replies with a `HELO` message, after which the server is supposed to send the status code `250 OK`. After the handshake between the client and the server has been completed the client begins by issuing commands. In the minimal SMTP protocol this consists of either sending `RSET`, `QUIT`, `NOOP`, or `MAIL FROM:` commands. If the `MAIL FROM:` command is sent the client proceeds by giving a number of recipients by successively issuing the `RCPT TO:` command, finally the client issues the `DATA` command and transmits the content of

```
data SMTPCommand = HELO Domain
                 | MAIL_FROM ReversePath
                 | RCPT_TO ForwardPath
                 | DATA
                 | RSET
                 | NOOP
                 | QUIT
                 deriving (Ord, Eq, Generic, NFData)
```

Figure 5.1: Minimal grammar of messages in the SMTP protocol

```
data SMTPReply = R500
               | R501
               | R502
               | R503
               | R504
               | R211
               | R214
               | R220 Domain
               | R221 Domain
               ...
```

Figure 5.2: Some of the replies to commands in the SMTP protocol

the email line by line, finishing with a line containing a single full stop. After this `DATA` transaction is completed, the communication returns to the state where the server can accept commands from the client.

The SessionCheck formalisation of the protocol is centered around two data types, *SMTPCommand* for describing the possible commands the client may call, and *SMTPReply* for describing the possible reply codes. Figure 5.1 gives the implementation of the *SMTPCommand*. In the interest of simplicity the *Domain*, *ReversePath*, and *ForwardPath* types are all equal to *String*. Figure 5.2 outlines the implementation of *STMPReply*. Every reply code is given its own constructor, note that this type could also have been implemented as a simple tuple (*Int*, *Maybe Domain*).

Figure 5.6 shows our specification of the SMTP command loop from the point of view of the server. The protocol begins with the RFC 821 handshake specified in Figure 5.3. The server starts by sending a `220 Service Ready` greeting, the client and server then exchange `HELO` and `250 OK` messages. After the handshake is complete, the server proceeds to accept `MAIL TO`, `QUIT`, and `RSET` commands. In the case of the latter two the session is terminated, in the case of the former a mail transaction begins. The mail transaction is specified in Figure 5.4 and consists of the client specifying multiple recipients and finally transitioning to the `DATA` phase of the transaction. In the `DATA` phase, specified in Figure 5.5, the client proceeds by sending a number of lines of text, finishing

```
handshakeRFC821 :: (SMTPCommand ⊑ t
                   , SMTPReply    ⊑ t) ⇒ Spec t ()
handshakeRFC821 = void $ do
    -- Handshake
  send r220Message
  get heloMessage
  send (is R250)
```

Figure 5.3: Specification of the RFC821 handshake

```
mail :: (String ⊑ t, SMTPReply ⊑ t, SMTPCommand ⊑ t) ⇒ Spec t ()
mail = do
  msg ← get $ anyOf [rcptMessage, dataMessage, is RSET]
  case msg of
    RSET → stop
    RCPT_TO _ → do
      send $ anyOf [is R250, is R550]
      mail
    DATA → do
      send $ is R354
      dataRecv
```

Figure 5.4: The command loop in the SMTP protocol

with a single line containing only a full stop. Note that the predicate constructed using *anyOf* to specify which messages are legal explicitly contains the predicate *is* ".". This redundancy shows a mismatch between the pure specification, in which *anything* would suffice to specify the behaviour of the client, and a specification which can be used to effectively test real implementations, in which case being explicit about the introduction of the case for "." helps SessionCheck to generate the special case message which will terminate the transmission.

5.1 Testing Implementations

The SessionCheck specification described above has been used to test both a client and a server implementation of the SMTP protocol. The client implementation, see Figure 5.8, was taken from the documentation of the python library ”smtplib” [6]. The server implementation, see Figure 5.7, was taken from the documentation of the python library ”smtpd” [5].

```
dataRecv :: (String ⊑ t, SMTPReply ⊑ t) ⇒ Spec t ()
dataRecv = do
  line ← get $ anyOf [anything, is "."]
  case line of
    "." → void ∘ send $ is R250
    _ → dataRecv
```

Figure 5.5: Specification of the data transmission phase of the SMTP protocol

```
smtp :: (String ⊑ t, SMTPReply ⊑ t, SMTPCommand ⊑ t) ⇒ Spec t ()
smtp = do
    -- Perform the handshake
  handshakeRFC821
  forever $ do
      -- Choice of operations
    op ← get $ anyOf [mailMessage, is QUIT, is RSET]
    case op of
      RSET → stop
      MAIL_FROM _ → do
        send (is R250)   -- Approximation
        mail
      QUIT → stop
```

Figure 5.6: Specification of the SMTP protocol in SessionCheck

```
import smtpd
import asyncore

class CustomSMTPServer(smtpd.SMTPServer):

    def process_message(self, peer, mailfrom, rcpttos, data):
        print 'Receiving message from:', peer
        print 'Message addressed from:', mailfrom
        print 'Message addressed to  :', rcpttos
        print 'Message length        :', len(data)
        return

server = CustomSMTPServer(('127.0.0.1', 1025), None)

asyncore.loop()
```

Figure 5.7: The implementation of `SMTPServer.py`

```
import smtplib
import sys

def prompt(prompt):
    return raw_input(prompt).strip()

fromaddr = prompt("")
toaddrs  = prompt("").split()

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n" %
        (fromaddr, ", ".join(toaddrs)))
while 1:
    try:
        line = raw_input()
    except EOFError:
        break
    msg = msg + line

server = smtplib.SMTP()
server.connect('localhost', 252525)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Figure 5.8: The implementation of `SMTPClient.py`

5.2 Bugs found

Several inconsistencies between the client implementation and our initial specification were discovered. It is important to note that some of the inconsistencies found were due to errors in our formalisation of the SMTP protocol from RFC821 [23], while others were due to what we would describe as strange, but not necessarily incorrect with respect to the library documentation, behaviour on the part of the client.

- Our first specification of the handshake procedure incorrectly required the client to send the first `HELO` message, according to the RFC document this is the task of the server.
- The documentation for the example claims that the implementation is consistent with RFC821, however a bug was discovered where the client used the `EHLO` message from RFC1869 instead of the `HELO` message required by RFC821 during handshake.
- The document RFC821 and its successors specify that "Command codes are four alphabetic characters. Upper and lower case alphabetic characters are to be treated identically." Our initial specification did not take this in to account and assumed all commands were upper case only.
- Our specification incorrectly looped back to the start of the protocol upon a successful email transmission attempt.
- The document RFC821 and its successors specify that an `RSET` message can be sent at any point in the communication. This was not captured in our specification.

The testing of the server implementation went more smoothly, due in no small part to the fact that it occurred *after* ironing out the bugs in the specification while testing the client implementation.

- The RFC821 specification requires that the server send a `220 Service Ready` message. Our specification did not account for this and SessionCheck reported an error when receiving it as the specification called for a `HELO` message.

5.3 Lessons Learned

The first, and most important, thing to take away from this case study is that SessionCheck works. We developed a formalisation of the SMTP protocol with which we were able to test both a client and a server implementation of the protocol.

The second lesson learned is that translating specifications from prose to SessionCheck is an error prone process. This is hardly surprising. English is not a formal language and it is easy to make a mistake in the details. Interestingly, one of the bugs in the formalisation was not discovered when testing the client implementation of the protocol but only at the time of testing the server. We put this down to correct implementations being *defensively* written. That is to say that the code will do its best to work well even when communicating with a faulty implementation of the protocol.

Chapter 6

Related work

This chapter presents a brief overview of related work and provides accounts of how SessionCheck relates to them as well as how we may incorporate some of the ideas present in the literature in future work.

6.1 Session Types

Using types as a method for formalising and verifying the implementation of protocol end-points has a rich history in the literature on Session Types [14, 33]. These systems generally work by forcing the programmer to implement their protocol end-point in a typed language which features built-in support for typing communication end-points. In these languages the specification of the protocol is generally formulated as a type. In return for implementing their code in a specific language the programmer receives a proof that the implementation adheres to the specification, that is to say that the program is type correct.

SessionCheck on the other hand does not require the implementation to be written in any particular language. Furthermore, SessionCheck specifications may be dependent, something requiring a very sophisticated type system to be able to express. However, SessionCheck does not provide a formal proof of correctness but rather the ability to gain confidence in the correctness of an implementation by means of testing.

While SessionCheck is heavily inspired by session types, we have made some pragmatic decisions motivated by real-world constraints. Specifically, we have omitted the choice $\oplus$ and branch $\&$ constructions from session types in favour of implementing them as the derived operations *choice* and *branch*. This decision is motived by the fact that the way these primitives are used in real-world applications varies between protocols. Protocols are commonly described in terms of explicit message passing rather than implicit integration into the underlying communication substrate (as is common in the session types literature).

6.2 Mocking

Mocking refers to the practice of creating software components specifically in order to exercise the functionality of a system under test. As an example of a mockup consider testing the functionality of a software component, which we will call *dashboard*, in a vehicle computer which is meant to read the speed of a vehicle and update the dashboard display appropriately. The computer has a simple interface consisting of two functions, *readSpeed* and *updateDisplay*. A mockup designed to test the *dashboard* component would consist of a sequence of expected calls to *readSpeed* and *updateDisplay* as well as their respective arguments and return values. In a notation similar to that of Svenningsson et al. [29] a mockup which expects the *readSpeed* and *updateDisplay* functions to be called sequentially may look like the following:

$$\mathit{readSpeed}\ () \mapsto 5.833\ .\ \mathit{updateDisplay}\ (\mathit{speed}, 21) \mapsto ()\ .\ \epsilon$$

This mockup specifies that the call to *readSpeed* will return 5.833 and that the subsequent call to *updateDisplay* will be called with the arguments *speed* and 21. In this case the speed returned by *readSpeed* is in m/s and the speed indicated on the display is meant to be in km/h. Previous work on frameworks for mocking for testing communicating parties in two-party and multi-party protocols by Svenningsson et al. [29] as well as the GoogleMock [2] and EasyMock [1] tools for C++ and Java respectively focus on mocking individual components. SessionCheck improves on the state of the art in mocking by introducing both the possibility of checking consistency of specifications as well as mocking both parties of a two-party protocol using a single specification.

6.3 Contracts, Chaperone Contracts, and Monitors

Contracts [18] are a way of extending functions to provide runtime monitoring of pre- and post-conditions and assigning blame to code which violates these conditions. In their implementation of typed contracts Hinze et al. [13] treat contracts as refinements of ordinary Haskell types. As an example consider the partial *head* function which takes a list and returns the first element:

$$\begin{aligned}&\mathit{head} :: [\,a\,] \to a\\&\mathit{head}\ (x : \mathit{xs}) = x\end{aligned}$$

In the scheme of Hinze et al. a contract of *head* which specifies that *head* may only be called on a non-empty list is specified as:

$$\begin{array}{l} \mathit{headContract} :: \mathit{Contract}\ ([\,a\,] \to a) \\ \mathit{headContract} = \mathit{prop}\ (\lambda \mathit{xs} \to \neg\ (\mathit{null}\ \mathit{xs})) \mathrel{-\!\!\triangleright} \mathit{true} \end{array}$$

Where $\mathit{prop} :: (a \to \mathit{Bool}) \to \mathit{Contract}\ a$ takes a predicate and lifts it to a contract, $(\mathrel{-\!\!\triangleright})$ combines two contracts to form a contract for functions, and *true* is the contract which is always satisfied, equivalent to $\mathit{prop}\ (\mathit{const}\ \mathit{True})$. Finally, associating *head* with its contract *headContract* is done using the function $\mathit{assert} :: \mathit{Contract}\ a \to a \to a$:

$$\begin{array}{l} \mathit{headWithContract} :: [\,a\,] \to [\,a\,] \\ \mathit{headWithContract} = \mathit{assert}\ \mathit{headContract}\ \mathit{head} \end{array}$$

When a programmer uses the new *headWithContract* function the input is dynamically checked and an error is reported in case the contract is violated, that is to say when *headWithContract* is called on the empty list. Crucially, the programmer can also specify location information for each call to *headWithContract*, which will extend contract violations errors with specific information about which call to *headWithContract* failed.

Melgratti and Padovani [17] introduce Chaperone Contracts, as a method for specifying higher-order two-party protocols (protocols which include transmitting protocol endpoints over the network). While this work is similar to SessionCheck it does not address the problem of mocking protocol end points. Furthermore, while the interface for contracts is very similar to ours, providing primitives similar to our *send* and *get*, the interface is not monadic. Rather, specifications need to be explicitly sequenced using the $(@@) :: \mathit{Spec} \to \mathit{Spec} \to \mathit{Spec}$ combinator. As a result of this dependent contracts, where the constraints in *send* and *get* depend on previous sent and received values, are specified using special $\mathit{send\_d} :: (a \to \mathit{Bool}) \to (a \to \mathit{Spec}) \to \mathit{Spec}$ and $\mathit{get\_d} :: (a \to \mathit{Bool}) \to (a \to \mathit{Spec}) \to \mathit{Spec}$ combinators. This introduces additional syntactic noise by making dependency more explicit than it already is.

One important benefit of Melgratti and Padovani's work over ours is the ability to write higher order specifications, that is to say specifications where protocol end-points, themselves having associated specifications, may be transmitted on the communication channels.

We also believe that modest extensions to SessionCheck would allow us to use our specifications as contracts in a way similar to Melgratti and Padovani. Doing this would effectively provide a more convenient (monadic) language for specifying contracts for the subset of protocols which are first order.

6.4 The Scribble Specification Language

The Scribble specification language [34] is a stand alone language which permits specification of multi-party protocols. Scribble specifications can be used to derive monitors which monitor communicating parties to find protocol violations, and to derive skeleton code for implementing the protocol in the Java language [12]. Scribble also features an analog of the *dual* operation, *project*, which turns a global specification into a local one. Both *dual* and *project* have their origins in the literature on session types [14] and the π-calculus [19].

Our work differs significantly from Scribble. SessionCheck is focused on testing and simulating protocols, while the language leverages as much of the Haskell host language as possible, making the implementation simple and succinct. The SessionCheck language being embedded means generating skeleton code from a specification is more difficult. However, we have techniques in mind for a version of SessionCheck which can handle both testing, use as a monitor, and generating skeleton code, bringing SessionCheck up to speed with Scribble.

Chapter 7

Conclusions, Discussion, and Future Work

In this report we have defined and implemented a language for specifying communication protocols called SessionCheck. It combines the idea of typed channels and duality from the literature on session types with mocking and property based testing in a style similar in part to chaperone contracts. As SessionCheck specifications have duals, meaning each specification describes both sides of a protocol, and are language agnostic they can be used to specify and test implementations of protocol end-points written in different languages.

We have shown how SessionCheck is implemented as an embedded domain specific language in Haskell. The implementation allows us to produce multiple interpretations of a single SessionCheck specification, including testing protocol end-points, shrinking examples of protocol violations, checking protocol self-consistency, and generating concrete examples of the protocol in action.

The case study in Chapter 5 demonstrates that SessionCheck can be useful for developing specifications for real world protocols. We were able to use SessionCheck to test both client and server implementations of the SMTP protocol, finding inconsistencies between the specification and the implementations and even something which could be argued to be a minor bug in one of the implementations. While we consider our work on SessionCheck successful, there are some limitations that need to be addressed in future work. Included in these are improvements to the shrinking algorithm as well as more thorough case studies. Ultimately, we believe it would be useful to have a large repository of SessionCheck formalisations of common protocols that programmers could use to verify their own implementations as well as the libraries that they use against.

7.1 Discussion

The decision to implement the SessionCheck specification language as an embedded domain specific language had significant impact on the usability of the tool. On the one hand it allows for a very convenient syntax and expressive semantics, with a simple implementation to match. On the other hand, the higher-order nature of a monadic embedding means that we were severely limited in the implementation of our shrinking algorithm. For example, the monadic implementation makes it impossible to implement a shrinking algorithm which can take advantage of any "automata-like" structure in the protocol to eliminate redundant loops and make intelligent choices at branching points et cetera. Having a first-order representation of the protocol specification could even have made it possible to use techniques from the property based testing literature [25, 11] which eliminates the need for shrinking entirely by enumerating test cases in order of size.

Another important consequence of the design decisions in SessionCheck, and especially the goal of having fully bi-directional specification and testing, is the presence of both intrinsic and extrinsic choice. Intrinsic choices are choices that may influence the run of a test suite which made by the mocking party, in our case SessionCheck. Extrinsic choice on the other hand denotes choices made by the system under test. Svenningsson et al.'s [29] mocking combinators are an example of a system with only extrinsic choice. That is, the combinators allow for the system under test to make different API calls depending on the situation, but the mocked party is always deterministic. As a consequence of the lack of intrinsic choice Svenningsson et al. are forced to use QuickCheck to generate several instances of each specification to exercise the system under test. In SessionCheck, however, the presence of both intrinsic and extrinsic choice complicate the semantics of the language in favour of making the specifications both bi-directional and self-contained. A SessionCheck specification is not a piece of QuickCheck code which generates a mockup, the specification is the mockup. We believe that the choices made in SessionCheck favour the programmer, making specifications both easy to write and to read.

7.2 Future work

Real world distributed systems often involve more than two parties, Honda et al. [15] present *"Multi Party Session Types"*, session types for multiple actors in a distributed system. These are used by the Scribble [34] protocol specification language. In multi-party session types, instead of the notion of duality, there is the more general notion of a projection of a global specification to a local actor $\mathit{project} :: \mathit{GlobalSpec} \to \mathit{Actor} \to \mathit{LocalSpec}$. Unlike two-party session types, multi-party session types are difficult to express in the monadic style of SessionCheck due to the risk of inadvertent sharing of information between actors without explicitly specifying messages. As an example showing why this might be problematic consider the following specification in an imagined multi-party version of SessionCheck where we have $\mathit{send} :: a \sqsubseteq t \Rightarrow \mathit{Predicate}\ a \to \mathit{String} \to \mathit{String} \to \mathit{Spec}\ st\ t\ a$ and $\mathit{send}\ p\ x\ y$ denotes sending a message satisfying p from x to y

```
spec :: Int ⊑ t ⇒ Spec t Int
spec = do
  a ← send anything "A" "B"
  send (greaterThan a) "C" "D"
```

The specification requires C to know about the value exchanged between A and B without any message being sent between B and C. This example illustrates that extending SessionCheck to multi-party protocols is not a simple generalization of the current implementation but rather requires some care to be taken in implementation. The literature on information flow control [10] provides some interesting approaches for handling problems of this kind. Including using monads parameterised by the security level or binding time of values, in both dynamic [27] and static contexts [7, 26]. However, implementing such extensions to SessionCheck is left as future work.

The greedy shrinking algorithm presented in chapter 2 works well for some examples but as demonstrated in the same chapter it is not perfect and works poorly for some bugs. There are multiple possible approaches to solving this problem. One approach is to provide a method for the programmer writing a SessionCheck specification to guide the shrinking algorithm in a way similar to the *shrink* method in the *Arbitrary* type class in QuickCheck [9].

Many communication protocols, including TCP [22], IP [24], and the BitTorrent protocol [4] are asynchronous, meaning that the order of *send* and *get* messages is not necessarily deterministic. Writing specifications of such protocols in SessionCheck is currently not possible and extending SessionCheck to support it poses several challenges which we intend to work on in the future. One such challenge is finding a semantics of interleaving specifications. Our initial tentative experiments suggest that incorporating an *interleave* :: *Spec t a* → *Spec t* () combinator into the specification language provides sufficient syntactic power to write what could intuitively be seen as a specification for the BitTorrent protocol. However, a choice needs to be made when choosing the semantics of *interleave*. One possible semantics of *interleave* simply interleaves the two specifications at random, for example turning:

```
do
  interleave $ do
    get anything
    send anything
  get (is ())
  send (is ())
```

into

```
do
  get (is ())
  get anything
  send anything
  send (is ())
```

effectively implementing a random, non-preemptive scheduler. Another possible semantics is to treat interleaved specifications and threads which may be preempted. Which semantics is the easiest to reason about? Which semantics is more convenient to specify protocols with? These are questions which we are excited to continue working on.

Client side applications like mail clients or graphical user interface applications require some form of user input to perform some form of communication with a server application. In order to test such applications thoroughly this user input also needs to be generated by the testing tool. In the case of the SMTP client case study this was achieved by creating random emails using QuickCheck and providing these as input to the client program. In the case of GUI applications this can be done using a tool like Selenium [3]. We consider the integration of SessionCheck with such tools an interesting avenue for future work.

Bibliography

[1] Easymock. `http://www.easymock.org`.

[2] Google c++ mocking framework. `http://code.google.com/p/googlemock`.

[3] The selenium tool. `http://www.seleniumhq.org/projects/`.

[4] The `BitTorrent` protocol specification. `http://www.bittorrent.org/beps/bep_0003.html`.

[5] The `smtpd` library. `https://docs.python.org/3/library/smtpd.html`.

[6] The `smtplib` library. `https://docs.python.org/2/library/smtplib.html`.

[7] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160. ACM, 1999.

[8] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[9] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

[10] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[11] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: functional enumeration of algebraic types. *ACM SIGPLAN Notices*, 47(12):61–72, 2013.

[12] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[13] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *FLOPS*, volume 6, pages 208–225. Springer, 2006.

[14] Kohei Honda. Types for dyadic interaction. In *CONCUR'93*, pages 509–523. Springer, 1993.

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

[16] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[17] Hernán Melgratti and Luca Padovani. Chaperone contracts for higher-order sessions. *Proceedings of the ACM on Programming Languages*, 1(ICFP):35, 2017.

[18] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[19] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992.

[20] John G. Myers and Marshall T. Rose. Post office protocol - version 3. STD 53, RFC Editor, May 1996. `http://www.rfc-editor.org/rfc/rfc1939.txt`.

[21] Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. *A Reflection on Types*, pages 292–317. Springer International Publishing, Cham, 2016.

[22] Jonathan B. Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc793.txt`.

[23] Jonathan B. Postel. Simple mail transfer protocol. STD 10, RFC Editor, August 1982. `http://www.rfc-editor.org/rfc/rfc821.txt`.

[24] Jonthan B. Postel. Internet protocol. STD 5, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc791.txt`.

[25] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. *SIGPLAN Not.*, 44(2):37–48, September 2008.

[26] Alejandro Russo. Functional pearl: two can keep a secret, if one of them uses haskell. In *ACM SIGPLAN International Conference in Functional Programming (ICFP)*, volume 50, pages 280–288. ACM, 2015.

[27] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *ACM Sigplan Notices*, volume 46, pages 95–106. ACM, 2011.

[28] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

[29] Josef Svenningsson, Hans Svensson, Nicholas Smallbone, Thomas Arts, Ulf Norell, and John Hughes. An expressive semantics of mocking. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411*, pages 385–399, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[30] Josef David Svenningsson and Bo Joel Svensson. Simple and compositional reification of monadic embedded languages. In *ACM SIGPLAN Notices*, volume 48, pages 299–304. ACM, 2013.

[31] Guido Van Rossum and Fred L Drake. *Python language reference manual*. Network Theory, 2003.

[32] Philip Wadler. How to declare an imperative. *ACM Computer Survey*, 29(3):240–263, September 1997.

[33] Philip Wadler. Propositions as sessions. *SIGPLAN Not.*, 47(9):273–286, September 2012.

[34] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *International Symposium on Trustworthy Global Computing*, pages 22–41. Springer, 2013.