GRACeFUL

Global systems Rapid Assessment tools
through Constraint FUnctional Languages

FETPROACT-1-2014 Grant N^{o} 640954

D4.4: Testing and verification framework

Testing and verification framework for RATs with applications to the CRUD case study

Lead Participant:	Chalmers (WP leader: P. Jansson)
Dissemination Level:	PU
Document Version:	Final
Date of Submission:	2018-01-25
Due Date of Delivery:	2018-01-31

Authors: Maximilian Algehed, Sólrún Einarsdóttir, Alex Gerdes, and Patrik Jansson.

Abstract

This fourth deliverable (D4.4) of work package 4 presents a framework for testing and verifying communicating systems. The work leading up to this deliverable is within Task 4.5 “Build a testing and verification framework for Rapid Assessment Tools (RATs)” and the full source code of the implementation is available on GitHub.

1 Introduction

The GRACeFUL project can be described as a stack of four layers, starting with stakeholder interaction (group model building, WP2) at the top, then the graphical user interface (GRACeFUL editor, WP3), the domain specific language for models and constraints (GRACe DSL, WP4) and finally the constraint programming (CP) backend providing efficient solution algorithms (WP5). In addition to the GRACe DSL, WP4 has developed a RESTful server around our DSL, called GRACeServer, for handling webservice requests. The server communicates both "upwards" to the GRACeFUL editor and "downwards" to the CP layer. This deliverable describes our methods for verifying correctness of this communication.

The division into frontend, server and backend is very common in software systems and this means that our results should be applicable for a wide range of other systems. Our approach to software verification is based on three main parts: declarative programming with strong types, property-based testing in general, and the SessionCheck tool for testing communicating systems in particular.

Scope and limitations This deliverable (D4.4) covers the software technology side of testing and verification of RATs. The CRUD RAT evaluation based on stakeholder sessions is reported elsewhere [D2.6 Evaluation Report CRUD RATs: m36] as is the interactivity and usability of the visual front-end [T3.4, D3.3 VA EDA Tool Prototype (RAT components): m34].

Rapid Assessment Tools RATs are used in large organisations like the World Bank and the United Nations to assess risks and needs and to make plans for improvement. In GRACeFUL the main focus is on Climate Resilient Urban Design, but the software developed in the project could be used for almost any Global Systems Science problem.

2 GRACeFUL software architecture

In this section we briefly describe the different layers of the GRACeFUL software stack and how testing and verification is performed on each layer.

Visual Editor

The top layer of the software stack is the visual editor. It provides a graphical user interface where the user can build GCMs as a graphical map from available components. The visual editor is implemented in the programming language JavaScript, using the Data Driven Documents library (D3.js). The code for the visual editor can be found in the GRACeFUL-project GitHub repository GRACeFULEditor.

The visual editor is described in deliverabe D3.3 including how it can be accessed[1], how it can be used, and more details on how it is implemented. The editor was evaluated by stakeholders in November 2017, details can be found in deliverable D2.6.

[1] http://vocol.iais.fraunhofer.de/graceful-rat/static/

GRACe

GRACe is a domain specific language embedded in Haskell. It is used to express GRACeFUL concept maps (GCMs) and GCM library components. GRACe programs representing GCMs are compiled to haskelzinc constraint programs, and the resulting solutions are passed back to GRACe. The DSL is described in detail in deliverable D4.2. We use different test approaches to validate the correctness of our DSL implementation. In Section 3 we explain how we use strong types and property-based testing. In addition to these two approaches, we also have a set of unit test cases, which test the entire software stack.

GCM component libraries

The visual editor allows users to create a GRACeFUL Concept Map with components of a chosen library. Components are written in GRACe and are grouped in a library, for example a library with CRUD components. The visual editor can query the components available in a library using its identifier. A JSON representation of the components are sent to the visual editor.

The following code shows an example CRUD library, where `rain` and `pump` are `GCM` components:

```
library :: Library
library = Library "crud"
    [ item "rain" $
        rain ::: "amount" #
          tFloat .-> tGCM          (tPort $ "rainfall" # tInt)
    , item "pump" $
        pump ::: "capacity" #
          tFloat .-> tGCM (tPair   (tPort $ "inflow"   # tInt)
                                   (tPort $ "outflow"  # tInt))
    ]
```

The example library makes use of Typed Values, which are explained in Section 3 (the syntax `"text" #` is used to tag ports with their names). We have implemented tools for property-based testing of GRACe components, described in Section 5.

Communication with visual editor

The GRACeServer provides a RESTful API and uses JSON[2] as the exchange format. The server offers the following webservices:

`libraries` returns a list with the available component libraries

`library (id)` has a library identifier parameter and returns a list with a description (in JSON) of the all library components

[2] JavaScript Object Notation

submit (gcm) takes a GRACeFUL Concept Map description as argument, which is translated to a GRACe DSL program, and returns the result of the constraint solver

The visual editor, developed in work package 3, communicates with the GRACeServer by making service calls. The GRACeServer is available as the `RestAPI` executable in the GRACe repository.

We have developed the tool SessionCheck, described in Section 4, to test the communication of systems such as this.

Haskelzinc

Haskelzinc is a Haskell interface to the MiniZinc constraint programming language. It provides a Haskell DSL for building MiniZinc model representations and a parser that returns a representation of the solutions obtained by running the MiniZinc model.

The latest version of haskelzinc (currently 0.3) is available from https://hackage.haskell.org/package/haskelzinc.

3 Verification through types and tests

Rapid Assessment Tools are used to evaluate plans for problems in Global System Science, such as Climate Resilient Urban Design. The evaluation output that is produced by the GRACeFUL tools should be reliable. This implies that the software behind the tools that we have developed should ideally be free of errors. It is well known that this is an utopia and not realistic. We can, however, do our best to keep the number of errors as low as possible. In this section we describe the two approaches to software verification that we have used in this project: types and property-based testing.

3.1 Strong types

Types are used to help the programmer to write code which doesn't "go wrong" and helps the compiler generate efficient code. But types can also be seen as *specifications* of parts of a program, such as functions and expressions. A type checker, normally part of a compiler, checks if the declared types match with the expected ones. A programmer can use a type checker to (semi-)formally verify the given specification, in the form of type declarations and annotations. In this sense, type systems can be regarded as the most used formal verification. It is of course a light-weight kind of formal verification, but nevertheless very useful.

We have used types extensively in the GRACeFUL project, in particular in our domain specific language GRACe, the GRACeServer, and in Haskelzinc, the interface to the constraint programming layer. Our DSL for GRACeFUL concept maps is embedded in Haskell, which is a strongly typed functional programming language. Using Haskell's type system, we can prevent a DSL user from making mistakes.

An end user of the GRACeFUL tool chain is most likely not programming directly in Haskell using our GRACe DSL, but using the Visual Editor from Work Package 3 instead. This poses a problem since the communication with the Visual Editor goes via an untyped channel. An additional problem is that we want to group `GCM` components, which we mentioned in Section 1, in a library, which is a challenge if you want to do this in a type safe manner. `GCM` components can have many different types, which makes it hard to collect them in a data type, such as a list.

We have addressed the above issues with *Typed Values*, which are related to the `Typeable` and `Dynamic` Haskell libraries [11]. We give a short description of Typed Values, leaving out many implementation details (which can be found in the GRACeFUL code repository on github). The key idea behind Typed Values is to combine a value with its *type representation*. A type representation can be implemented as follows in Haskell:

```
data Type t where
  -- Base types
  TInt   :: Type Int
  TBool  :: Type Bool
  -- Annotations
  Tag   :: String -> Type t -> Type t
  -- Lists
  List  :: Type t -> Type [t]
  -- Functions
  (:->) :: Type a -> Type b -> Type (a -> b)
```

We use the above Generalised Algebraic DataType (GADT) to encode a type representation. The constructor `TInt`, for example, represents the `Int` type. The constructor for the list type representation takes another type representation as argument. So, the `List TBool` value represents the type “list of boolean”. The `Tag` constructor is used to annotate a type’s representation and leaves the represented type untouched. We can use such annotations to add meta-data (visual appearance in the editor, like port names and icons). Finally, the `(:->)` constructor can be used to represent function types.

We use the type representation `Type t` to create a Typed Value, implemented with the following GADT:

```
data TypedValue where
  (:::) :: a -> Type a -> TypedValue
```

Note that type variable `a` does not appear in the result type. We combine a value, of type `a`, with its type representation, of type `Type a`. Using this data type we can hide the type of a value, and turn it into an untyped value. We can, for instance, put values of different type into a list of Typed Values. This addresses one of the issues we identified above. It is important the type representation is part of a Typed Value; it allows us to extract the (typed) value, of type `a`, from a `TypedValue`, that is we can go back to the typed realm in a type safe manner!

This is, unfortunately, not as easy as it may seem. We need to have a way to convince the Haskell compiler that two type representations encode the *same* type. The following data type and type equality function does this:

```
data a :~: b where
  Refl :: a :~: a

(?=) :: Type a -> Type b -> Maybe (a :~: b)
TInt         ?= TInt       = return Refl
TBool        ?= TBool      = return Refl
Tag t        ?= Tag t'     = t ?= t'
(List t)     ?= (List t')  = do
  Refl <- t  ?= t'
  return Refl
(t0 :-> t1)  ?= (t0' :-> t1') = do
  Refl <- t0 ?= t0'
  Refl <- t1 ?= t1'
  return Refl
```

The actual implementation of Typed Values is much more involved, but in essence the same as the code showed above. We use these Typed Values to create libraries of GCM components, and to communicate with the untyped outside world. Using these Typed Values we can enjoy the benefits of types as long as possible.

Using a strongly typed programming language, such as Haskell, and making good use of the types allows programmers to be more confident about their code and catches many mistakes. Alas, it does not prevent all mistakes and it is necessary to test our implementation, to even further increase our confidence. The next section explains the type of testing we have used in this project.

3.2 Property-based testing

A traditional way of testing software is using so-called unit tests. An unit test case specifies some input data along with the expected output. We use these test cases to assure that the software is behaving as expected. For real-life sized software projects, however, there are many test cases needed to get a decent test coverage, which makes maintenance more difficult. Moreover, it is hard to think of all the corner cases that may be present in the software.

An complementing approach is *property-based testing*, which verifies software against particular properties that the software should have. A property can be regarded as an abstraction of many unit tests. These properties should hold for *all* inputs, which would be impossible to do by hand, but the property-based testing framework that we use (QuickCheck [4]) can actually *generate* input data automatically. QuickCheck uses generated random input data to test such properties. Properties are powerful: a good property gives a strong specification for a large set of test data.

QuickCheck is a well known implementation of property-based testing, and is available in many programming languages. QuickCheck verifies properties by generating a large number of random test cases, and reports test cases for which the property fails.

For example, a property of the `reverse` function from the Haskell standard library (the `Prelude`) is that if we reverse a list twice, we end up with the same list. This can be expressed in Haskell QuickCheck as follows:

```
propTwice :: [Int] -> Bool
propTwice = \ xs -> xs == reverse (reverse xs)
```

We can validate such a property by calling `quickCheck`:

```
> quickCheck propTwice
+++ OK, passed 100 tests.
```

The above evaluation shows that the `reverse` function most likely fulfills the `propTwice` property. When QuickCheck finds a failing test case (a counterexample to the property), it shrinks the failing test case automatically, by searching for similar, but smaller test cases that also fail. The result of shrinking is a minimal failing test case, and this simplifies the debugging process.

Our tool SessionCheck, described in Section 4, uses ideas from property-based testing to test communicating systems. And in Section 5 we describe our property-based testing of programs written in GRACe.

4 Testing Communicating Systems

SessionCheck [3] is a tool developed in WP4 to test distributed systems like the GRACeFUL editor. The tool is focused on the communication protocols used between the components. In the case of the GRACeFUL editor the protocol would be the REST API described in deliverable D4.3. But, as the tool can do more than that, we here give a mini-tutorial of how to use SessionCheck in general.

Typical testing of distributed systems is done by maintaining a database consisting of hundreds or thousands of test-cases, each specifying the behaviour of one party in the protocol with respect to a specific exchange of messages, a trace. This approach is not ideal, as each test case is an artifact on its own which needs to be kept in sync with the rest of the software. SessionCheck relieves developers of the duty of maintaining hundreds of software artifacts, replacing them with only three: the client, the server, and the SessionCheck specification.

To see the SessionCheck specfication language in action, 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 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:

```
data Spec t a    -- SessionCheck spec. for channel type t

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 1: The SessionCheck language

```
protocol' :: Spec Int Int
protocol' = do
  a <- get posInt
  b <- get posInt
  send (is (a + b))
```

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 [10] 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 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 type constraint a :< t (for send and get) denotes a subtyping relation between a and t. That is, any value of type a can be transformed into value of type t, and it may be possible to transform a value of type t into value of type a.

Included in the interface are also the stop and fail functions. The stop primitive specifies that the protocol session is successfully terminated. The behaviour of stop is specified by the equation stop >>= f = stop. Unlike send, get, and stop, the fail primitive 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 constraints on messages. The choose and branch primitives are not actually primitive operations, rather they are derived from (implemented using) 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 explained so far 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 [13], which contains the two primitive operations `return` and `(>>=)` (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 >>= 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 = send anyInt` and `f x = send (greaterThan x)`, here `s >>= 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 is syntactic sugar for successive uses of `(>>=)` and lambda abstraction `\ x -> e`, where the expression `\ x -> e` denotes a function where the variable `x` is used to bind the input of the function in the (output) expression `e`. When written using explicit `(>>=)` the specification of `protocol` could look like this:

```
protocol :: Spec Int Int
protocol = send posInt >>= \ a -> send posInt >>= \ b -> get (is (a + b))
```

Supporting the generic `Monad` interface means that we get several useful combinators "for free", like `(>>) :: Spec t a -> Spec t b -> Spec t b` which sequences two independent specifications, and `forever :: Spec t a -> Spec t b` 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.

As a third example of a SessionCheck specification we can take the `echo` protocol:

```
  echo = do
    s <- send anyString
    get (is s)
```

In this protocol the client sends a single string to the server which replies with the same string. We can use this specification to verify that either a client or a server implementation of the protocol is correct. For example, when testing a faulty server implementation SessionCheck will print the following:

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

With trace:
Output {""}
InputViolates {is ""} "\n\r"
```

To see the SessionCheck language in action we will briefly explore using SessionCheck to specify a small book ordering service. The idea of this protocol is that the client transmits a number of books it wants to order to the server, followed by a `checkout` message. The server then transmits back to the client the clients current basket. We hope that this small example is sufficient to see how SessionCheck might be used to specify larger protocols. We start by specifying the type of messages which may be transmitted between the client and the server. The client will transmit Request messages and the server will reply with Reply messages. In this simple example we get the ordinary Haskell types below.

```
data Request  =  Order String | Checkout

data Reply    =  Basket [String]
```

The protocol contains a loop where the client orders books.

```
loop :: Request :< t =>  [String] -> Spec t [String]
loop books = do
  r <- anyRequest
  case r of
    Order book -> loop (book:books)
    Checkout   -> return books
```

Having specified the loop we can put everything together into the final protocol.

```
protocol :: (Request :< t, Reply :< t) =>  Spec t Reply
protocol = do
  books <- loop []
  get (supersetOf books)
```

So far we have only seen how we can specify and test one part of a protocol. However, thanks to something called duality, a client specification gives rise to a server specification without any effort. The key idea of duality is that if a specification specifies one part of a two-party protocol, then the same specfication with all instances of `send` replaced by `get` and vice-versa specifies the other. In SessionCheck, duality is implemented as a function `dual :: Spec t a -> Spec t a`. The specification (not implementation) of `dual` can be seen below.

```
dual :: Spec t a   ->  Spec t a
dual    (send p)    =  get p
dual    (get p)     =  send p
dual    stop        =  stop
dual    (fail s)    =  fail s
dual    (return a)  =  return a
dual    (s >>= f)   =  dual s >>= dual . f
```

To see duality in action consider the `echo` protocol above. When written using explicit (>>=) the protocol looks like this

```
echo = send anyString >>= \ s -> get (is s)
```

the `dual` of which can be computed to

```
dual echo = get anyString >>= \ s -> send (is s)
```

Which when written using the more convenient do notation is simply

```
dual echo = do
  s <- get anyString
  send (is s)
```

Which is clearly a specification of an echo server.

Using SessionCheck to Test GRACe-like Tools We will briefly turn our attention to how SessionCheck could be employed to test tools like the GRACeFUL RAT. There are multiple communicating components in the GRACeFUL software architecture. The two primary components are the visual editor, running as a client in a web browser, and the GRACe server. The GRACe server itself consists of two components, a front end which communicates with the visual editor, and a back end which communicates with the the MiniZinc solver. Both these avenues of communication could possibly be tested by SessionCheck.

To test the communication between the GRACe server and the visual editor we can model the API calls between the client and the server as two data types ClientRequest and ServerResponse, defined as follows:

```
data ClientRequest =
    AskLibraries
  | SubmitProgram LibraryName ProgramRepresentation

data ServerResponse =
    Libraries   [Library]
  | Assignments [(VariableName, Value)]
  | Error503
```

The specification needs to makes use of external knowledge about the system being specified and tested. The specific piece of knowledge required is what libraries the server has access to. The specification is written to be parametric in this information:

```
spec :: (ClientRequest :< t, ServerResponse :< t)
     => [Library] -> Spec t ()
spec allLibraries = do
  req <- get anyClientRequest
  case req of
    AskLibraries  -> send (is $ Libraries allLibraries)
    SubmitProgram libName progRep
                  -> send (validAssignments libName allLibraries progRep)
  spec allLibraries
```

We omit the definition of the predicate

```
validAssignments :: LibraryName -> [Library] -> ProgramRepresentation
                 -> Predicate ServerResponse
```

in the interest of brevity. Writing the predicates in this specifiation is not necessarily a simple task. To describe the client we would need to define what a correct program representation is and come up with a way to generate such a representation.

To sum up, SessionCheck provides a small DSL, implemented in Haskell, for specifying communicating systems. It benefits from the strong type system of Haskell, but also extends the reach of the specification to a large class of communication protocols, including cases when the “other end” of the communication is not written in Haskell (as in the case of the GRACeFUL visual editor).

5 Property-based testing of GRACe programs

We use the `GCMP` (GRACeFUL Concept Map Property) type to define properties of GRACe components and programs to test. With the help of QuickCheck generators we can generate GRACe programs with random parameter values and then test whether a given `GCMP` property holds for all cases. We test this by compiling the program to a MiniZinc model, running the model through a constraint solver, and checking whether the solution found by the solver satisfies the specified property.

We can test whether a GRACe component has been implemented correctly by defining `GCMP` properties that describe its expected behavior and checking whether those properties hold for all generated cases. The user defines `GCMP` properties in a syntax similar to that used to define `GCM` components and programs.

The following example code defines `GCMP` properties for the `pump` component from the CRUD component library shown in section 2.

```
prop_pump :: GCMP ()
prop_pump = do
  -- Generate arbitrary parameter
  k   <- forall (fmap abs QC.arbitrary)
  -- Pump with arbitrary parameter
  (iflow, oflow) <- liftGCM $ pump k

  -- These properties should hold for the pump
  property "Inflow within capacity" $
    portVal iflow .<= lit k
  property "Outflow equal to inflow" $
    portVal oflow === portVal iflow
```

We can test these properties using our `GCMP` testing framework. If they hold for all generated test cases (currently set to 100) we get the following output:

```
Testing Pump...
+++++ Test passed! +++++
```

If one of the properties does not hold, the output gives an overview of all properties and stating if they passed or failed. For example, suppose there is an error in the implementation of the pump such that the outflow is not always equal to the inflow. Then the second property should fail for some test cases, and the following output will appear, showing that in the first failing test case the first property held but the second one did not:

```
Testing Pump...
Failing test case:
("prop_0 Inflow within capacity",True)
("prop_1 Outflow equal to inflow",False)
----- Test failed! -----
```

Using property-based testing for GCM components we can increase the confidence in the implementation of a GCM component.

6 Related Work

This section presents a brief overview of related work on the main topics in this document: Typed Values and SessionCheck. For related work on property-based testing we refer to the original paper about QuickCheck [4].

6.1 Typed Values

Our implementation of Typed Values are related to the Typeable and Dynamic libraries in Haskell [11], as mentioned before in Section 3. These libraries also offer the possibility to hide the type of a value and to go back and forth from the untyped world. The main difference is that our implementation is able to *annotate* a type representation, using the Tag constructor. We need this functionality to add meta-data to a type representation of GCM parameters, of which we have showed an example in Section 1. The implementation of Typeable and Dynamic is unfortunately closed, which means it is not possible to add a Tag-like construct.

6.2 SessionCheck

This subsection describes research related to SessionCheck and how we may incorporate some of the ideas present in the literature in future work.

Mocking Mocking refers to the practice of simulating the behaviour of the environment 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. [12] a mockup which expects the readSpeed and updateDisplay functions to be called sequentially may look like the following:

```
readSpeed() |-> 5.833 . updateDisplay(speed, 21) |-> () . ε
```

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. [12] 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.

In the context of GRACeFUL, mocking is useful for testing a single layer of the software stack in isolation. Specifically, it can be used to test the GRACe server separately from the visual editor front-end.

Contracts, Chaperone Contracts, and Monitors Contracts [9] 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. [6] 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:

```
head :: [a] -> a
head (x:xs) = x
```

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 written as:

```
headContract :: Contract ([a] -> a)
headContract = prop (\xs -> not (null xs))  ->>  true
```

Here prop :: (a -> Bool) -> Contract a takes a predicate and lifts it to a contract, true is the contract which is always satisfied, equivalent to prop (const True), and the infix combinator (->>) forms a contract for functions.

Finally, associating head with its contract headContract is done using the function assert :: Contract a -> a -> a:

```
headWithContract :: [a] -> [a]
headWithContract = assert headContract head
```

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 [8] introduce "Chaperone Contracts", as a method for specifying higher-order two-party protocols. (Higher-order here means protocols which can transmit 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 `(@@) :: Spec -> Spec -> 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 `send_d :: (a -> Bool) -> (a -> Spec) -> Spec` and `get_d :: (a -> Bool) -> (a -> Spec) -> 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 their ability to write higher order specifications, where protocol end-points (including associated specifications), may be transmitted on the communication channels. We 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.

The Scribble Specification Language The Scribble specification language [15] 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 [5]. 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 [7, 14].

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.

7 Conclusion

The GRACeFUL testing and verification framework is based on three parts: declarative programming with strong types (in Haskell), property-based testing in general (using QuickCheck), and the SessionCheck tool for testing communicating systems in particular. The strong type system of Haskell and the property-based testing tool QuickCheck are off-the-shelf techniques developed by others but adapted for GRACeFUL by WP4. The main new contributions were Typed Values (described in Section 3), SessionCheck (described in Section 4), and property-based testing of GRACe programs (described in Section 5).

References

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

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

[3] Maximilian Alghed. Bidirectional testing of communicating systems. Master's thesis, Chalmers University of Technology, 2017.

[4] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. of the fifth ACM SIGPLAN international conference on Funct. Prog.*, pages 268–279. ACM, 2000.

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

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

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

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

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

[10] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):i–xii,1–255, Jan 2003. http://www.haskell.org/definition/.

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

[12] 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.

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

[14] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA, 2012. ACM.

[15] 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.