Component Reliability Extensions for Fractal component model

Architecture/Design manual and User manual

Final deliverable (T0+18)

Jiri Adamek
Tomas Bures
Pavel Jezek
Jan Kofron
Vladimir Mencl
Pavel Parizek
Frantisek Plasil

Component Reliability Extensions for Fractal component model: Architecture/Design manual and User manual: Final deliverable (T0+18)

by Jiri Adamek, Tomas Bures, Pavel Jezek, Jan Kofron, Vladimir Mencl, Pavel Parizek, and Frantisek Plasil

Table of Contents

Chapter 1. Introduction

1.1. Fractal

Fractal is a component model developed initially by France Telecom and INRIA and later as an open source project in the ObjectWeb consortium. The component model is defined by the Fractal Component Model specification [BCS]. The specification defines a hierarchical component model, where a component is specified in terms of its server and client (provided and required) interfaces and configurable attributes. The model supports advanced features such as component sharing, mandatory/optional interfaces, collection interfaces. The Fractal API is defined for three languages: Java, C, and CORBA IDL. The reference implementation of Fractal, Julia, is developed in Java and supports Java Fractal components.

The Fractal specification allows to use an Architecture Description Language (ADL), however, it does not directly specify one. In the Fractal ADL project, an XML-based ADL for the Fractal component model is defined to specify the initial architecture of an application. The features of this ADL language include inheritance among component specifications, and also a mechanism to specify values of components' attributes.

1.1.1. Basic assumptions

The Fractal component model specification is very flexible (and structured in several conformance levels), consequently, many concrete component systems comply with it. To make the integration of behavior protocols into Fractal possible, we take the following additional assumptions:

(1) In Fractal, every component has internal and external interfaces. We suppose that for every external interface there exists an internal interface of the same type (and vice versa). In addition, an event on an external interface immediately causes the complementary event on the corresponding internal interface, and these two events happen atomically. In a similar way, an event on an internal interface immediately causes the complementary event on the corresponding external interface (and the two events happen atomically).

(2) Interfaces in Fractal are connected by bindings. We suppose that an event occurring on an interface `I` causes immediately the complementary event on the interface `I` is bound to, and the two events happen atomically, assuming `I` is bound to exactly one interface. If `I` is bound to more interfaces, the events on those interfaces do not have to happen atomically.

1.2. Behavior protocols

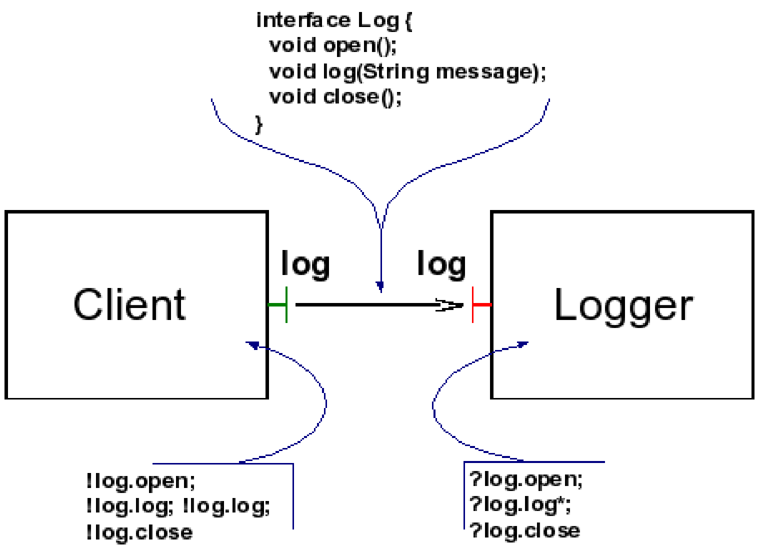
The purpose of *behavior protocols* is to specify the *behavior* of software components, so that interesting properties of their behavior can be verified.

The problem of behavior verification is undecidable in general. There are two ways to face it: (1) To use behavior description languages which describe behavior of the software precisely and to put up with the fact that the tools will never stop for some inputs (behavior descriptions). (2) To use behavior description languages, which are not expressive enough to describe behavior of software precisely, but the verification of the specifications is decidable. We have chosen the second approach. Therefore, a behavior protocol should be seen rather as an approximation of a component's behavior. The most important benefit of this approach is the existence of a fully automatic behavior verification procedure (implemented in our behavior protocol checker).

The main difference between "full" description of a component behavior and a corresponding behavior protocol is that the protocol describes only sequences of method calls on the component's interfaces, abstracting from the values passed as method parameters and return values. Such a level of abstraction is very suitable for verification tasks specific to software components.

1.2.1. Behavior protocol basics

Figure 1.1. Example of a component application with behavior protocols



On Figure 1.1, there is an example of a component application consisting of two simple components. The component `Logger` provides basic logging functionality, which is used by the component `Client`. Therefore, `Client`'s (required) `log` interface is bound to `Logger`'s (provided) `log` interface.

`Logger`'s `log` interface consists of three methods: `open`, which has to be called at the beginning of the "logging session", `log`, which can be called several times after the `open` method was called (every call of the `log` method causes writing of the string passed as the `message` parameter into a persistent store), and the `close` method, which has to be called at the end of the "logging session".

In a classic software development process, description of a component's functionality (such as `Logger`) has typically the form of a plain English text, which is not suitable for an automatized behavior verification. To fill this "semantic gap", we add a behavior protocol to the "classic" component interface specification.

For example, the behavior protocol of `Logger`, consistent with the plain English specification above, reads as follows:

```
?log.open;
?log.log*;
?log.close
```

This protocol consists of tokens denoting method calls (`?log.open`, `?log.log`, and `?log.close`) and operators specifying the ordering of the method calls (`;` and `*`). Every of these tokens consists of the question mark, denoting that the method call is *absorbed* by `Logger`, and the qualification of the method within the component, consisting of the interface name and the method name (separated by the dot sign). Finally, the `;` binary operator stands for *sequencing* of method calls, while the `*` postfix unary operator denotes zero or more *repetitions* of `?log.log`. Therefore, this protocol indeed specifies what was written informally above: the call of `log.open` is absorbed, then zero or more calls of `log.log` are absorbed, and finally `log.close` is absorbed.

Now, let us focus on the behavior protocol of `Client`:

```
!log.open;
!log.log;
```

```
!log.log;
!log.close
```

It differs from `Logger` in two ways: First, the method qualifications are preceded with the exclamation mark, which stands for *emitting* a method call. Second, it specifies that `Client` calls `log.log` exactly twice. It is correct, because `Logger` is ready to accept an arbitrary number of `log.log` calls, if they occur after `log.open` and before `log.close` (which is the case).

In the rest of this chapter, we show an overview of verification tasks, which can be done with behavior protocols. Detailed description of behavior protocols can be found in Chapter 2.

1.2.2. Static checking

Using the (static) behavior protocol checker, two important properties of component behavior can be analyzed statically (i.e., at the development time): *composition errors* and *behavior compliance*.

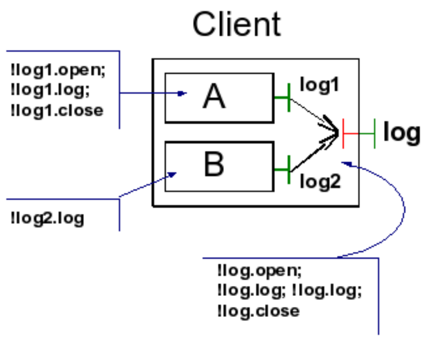
To explain what a composition error is, let us assume that the behavior protocol of the `Client` component from Figure 1.1 changed in the following way:

```
!log.log;
!log.log;
!log.close
```

I.e., `Client` does not call `log.open` at the beginning. However, `Logger` expects `open` to be called as the first one. In general, an attempt of a component A to call a method of another component B in a situation where the call is not expected by B (i.e., such a behavior of B is not specified in B's behavior protocol) is called *bad activity* (of A). Bad activity is one of composition errors, as it results from composition of components with incompatible behavior protocols. Other types of composition errors are described in Section 2.4.2.

The basic idea of behavior compliance is that for a composite component C, the behavior of its "internals" (determined by joint behavior of its subcomponents S1, ..., Sn) should be compliant with the behavior specified by C's protocol.

Figure 1.2. Example of a composite component with behavior protocols



On Figure 1.2, there is an example of an implementation of the `Client` component. `Client` consists of two subcomponents - `A` and `B`, whose behavior protocols are also in Figure 1.2.

To check the compliance, the first task is to figure out the behavior of `A` and `B` being run simultaneously. In this case, it is simple, as `A` and `B` do not communicate with each other, therefore the behavior of `Client`'s internals is

```
(
  !<A:log1-log>.open;
  !<A:log1-log>.log;
  !<A:log1-log>.close
) | (
  !<B:log2-log>.log
)
```

There are several new constructs in this protocol. First, it does not specify behavior of a component, but the behavior of a *group* of components. Therefore, to fully qualify a method, it is not more suitable to use the local name of the interface. Instead, name of the *binding* is employed. For example, `<A:log1-log>` identifies binding between the `log1` interface of the `A` subcomponent and the `log` interface of the supercomponent (from the context, `Client` is known to be the supercomponent here; this is why its name is not used to prefix the `log` interface name).

Second, the `|` binary operator stands for *parallel* execution of two subprotocols. This is what is needed to be expressed - the subcomponents `A` and `B` run independently on each other.

The behavior of internals of the `Client` component is not compliant with its protocol. There are two reasons: (1) `B` can emit its call (`!<B:log2-log>.log`) before `open` is called by `A` or after `close` is called by `A`, and (2) the calls of the `log` method emitted by `A` and `B` can occur in parallel. None of these situations is permitted by the behavior protocol of `Client`.

The construction of the protocol describing behavior of `Client`'s internals was shown only for illustration. In practice, this is done automatically by the behavior protocol checker. Also, in the situation when components in a group communicate with each other, it is not possible to use the parallel operator to specify the resulting behavior - more advanced operators have to be used instead (see Chapter 2).

1.2.3. Verifying behavior of primitive components

For a *primitive component* (i.e., a component that is directly implemented in Java instead of being composed of several subcomponents), the behavior compliance (Section 1.2.2) cannot be verified by the static checker (Section 1.2.2). The reason is that the static checker requires the behavior of both the component and its internals (subcomponents/implementation) to be specified by behavior protocols, and it can therefore verify the compliance for composite components only.

There are two ways to verify whether the implementation of a primitive component is compliant with the protocol (or, as we say in this context, whether the component's behavior is *bounded* by the protocol): *run-time checking* (Section 1.2.3.1) and *code analysis* (Section 1.2.3.2).

1.2.3.1. Run-time checking

The main motivation for developing the *run-time checker* is the run-time verification of a primitive component. However, in principle, the run-time checker can be used also for a composite component.

The run-time checker keeps track of the method calls on external interfaces of a component at run-time and checks whether the behavior of the component is bounded by its protocol.

The main disadvantage of this approach is that (unlike the static checking) it is not exhaustive: even if the behavior of a primitive component is not bounded by the protocol, it may not become evident for many runs of the component application monitored by the run-time checker.

More details on run-time checking and the differences between static and run-time checking are presented in Section 2.5.

1.2.3.2. Code analysis

The *checker for code analysis* verifies whether the behavior of a primitive component is bounded by the behavior protocol of the component. The verification is done statically (i.e., at the development time) and is based on the analysis of the component code. It is exhaustive, i.e., if the behavior of the component is not bounded by the protocols, it is always detected, if the analysis completes. On the other hand, the code analysis is an undecidable problem in general, i.e., the analysis may not stop for some inputs. Even if it stops, it is a very time- and memory-consuming process: therefore, we provide the developers with both run-time checker and the checker for code analysis, and they should be seen as complementary to each other.

More details on code analysis of primitive components are presented in Chapter 3, Section 4.7.1 and Section 4.7.2

Chapter 2. Behavior protocols overview

2.1. Events and traces

Events are the keystone of behavior protocol semantics. Every event is atomic. We define two types of events: requests and responses. Let `m` be the (fully qualified) name of a method. Then, `m^` stands for a request/call of `m` and `m$` stands for a response/return from `m`.

Always, two components cooperate on an event: one component *emits* the event and another component *absorbs* the event. To distinguish between those two roles, we use the prefix `!` for emitting and `?` for absorbing. If `m` is a method name, the symbols `?m^`, `?m$`, `!m^`, `!m$` are called *event tokens*. Recall Figure 1.1 from Section 1.2.1. In the protocol of `Client`, `!log.log^` would stand for emitting the call of `log.log`, while `?log.log$` would stand for absorbing the return from `log.log`.

To specify that an event occurs as an *internal event* of a component `C` (i.e., it results from a communication of `C`'s subcomponents), we use the # prefix.

To provide a way to specify a request and the corresponding response at once, we define *abbreviations*: if `m` is a (fully qualified) name of a method, `?m` is an abbreviation for the protocol `(?m^ ; !m$)` (the whole method call from the point of view of the callee) and `!m` stands for `(!m^ ; ?m$)` (the whole method call from the point of view of the caller). In fact, in the examples in Chapter 1, only these abbreviations were used to specify the behavior, and usage of explicit requests and responses was not necessary.

We also define two more complex abbreviations: if `P` is an arbitrary protocol, `?m{P}` means that the call request of `m` is absorbed, and while `m` is processed, the component behaves as specified by `P`; afterwards, the call response of `m` is emitted. In a similar way, `!m{P}` means that P specifies the behavior of the caller between issuing the call of `m` and receiving the response of `m`.

The abbreviations not only serve as syntactic sugar, allowing to write readable behavior protocols, but they also explicitly denote pairing of events (requests and corresponding responses). In certain situations, such information is essential for the behavior protocol checker. This is why for certain types of interfaces, only an abbreviation can be used to specify the method call, and the use of explicit event specification is forbidden (see Section 4.5.1).

A computation of a component application is formally described by a *trace* - a finite sequence of event tokens. Every protocol specifies a set of traces. Recall the protocol of `Client` from Figure 1.1:

```
!log.open;
!log.log;
!log.log;
!log.close
```

It specifies a single trace:

```
<!log.open^, ?log.open$,
!log.log^, ?log.log$,
!log.log^, ?log.log$,
!log.close^, ?log.close$>
```

For `Logger`, the situation is more complex:

```
?log.open;
?log.log*;
?log.close
```

This protocol specifies an infinite number of traces, as it accepts arbitrary number of calls to `log.log`. We show the first three shortest traces specified by the protocol:

```
<?log.open^, !log.open$, ?log.close^, !log.close$>

<?log.open^, !log.open$, ?log.log^, !log.log$,
?log.close^, !log.close$>

<?log.open^, !log.open$, ?log.log^, !log.log$,
?log.log^, !log.log$, ?log.close^, !log.close$>
...
```

The set of all traces specified by a protocol `P` is denoted as `L(P)`.

2.2. Behavior protocol basic operators

For behavior protocols, the following basic operators are defined: *sequencing* (denoted by `;`), *repetition* (denoted by `*`), *alternative* (denoted by `+`), *and-parallel* (denoted by `|`), and *or-parallel* (denoted by `||`). We illustrate the meaning of the operators (except the last one) on the following protocol of the `Client` component from Figure 1.1:

```
!log.open;
(
   (!log.log | !log.log)
   +
   !log.log*
)
!log.close
```

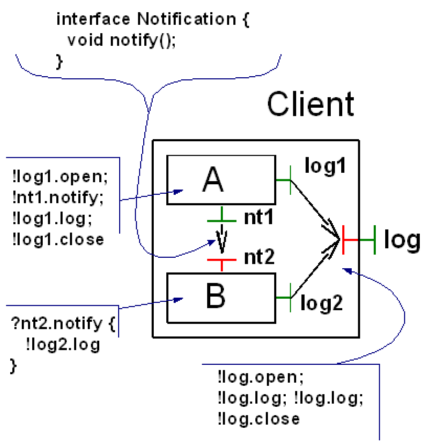
`Client`, whose behavior is specified by this protocol, first calls `log.open`. Then, it either calls `log.log` twice in parallel, or it calls `log.log` several times sequentially (or it does not call `log.log` at all, as `*` stands for zero or more repetitions). At the end, it calls `log.close`.

Or-parallel is defined as follows: if `P` and `Q` are protocols, `(P || Q)` stands for `(P + (P | Q) + Q)`.

2.3. Frame and architecture protocols

From the point of view of behavior, every component can be divided into two parts: *frame* and *architecture*. The frame of a component `C` consists of all interfaces which are provided or required by `C` to "outside world" (the components which are external to `C`). The architecture of `C` consist of frames of `C`'s direct subcomponents and bindings between those frames (and also bindings between interfaces of `C`'s subcomponents and interfaces of `C` itself).

Figure 2.1. Example of a composite component with bindings among subcomponents



On Figure 2.1, the frame of `Client` consists of the (only) `log` interface, while its architecture is formed by the frames of `A`, `B` and the bindings `<A:log1-log>`, `<B:log2-log>`, `<A:nt1-B:nt2>` (as explained in Section 1.2.2, in the context of the `Client` component `<A:log1-log>` stands for the binding of the `log1` interface of the `A` subcomponent to the `log` interface of `Client` itself; here, we introduce `<A:nt1-B:nt2>` - the binding between interfaces of `A` and `B` subcomponents).

An architecture is always associated with a concrete frame (we also say that the architecture *implements* this frame).

Following the definition of frame and architecture, we also distinguish between *frame protocols* and *architecture protocols*. Frame protocol of a component `C` describes requests and responses on the frame of `C`. The frame protocol is specified by the developer. The architecture protocol of `C` is automatically constructed from the frame protocols of `C`'s direct subcomponents by the behavior protocol checker. It describes what is happening "inside" `C`.

In the architecture protocol of a component `C`, two types of events appear: events on the frame of `C`, and events resulting from the communication of `C`'s direct subcomponents (internal events). The first type of events is denoted in the same way as in frame protocols. The # prefix is used (in both event tokens and abbreviations) to denote internal events (Section 2.1).

For example, the architecture protocol of the `Client` component from Figure 2.1 reads as follows:

```
!<A:log1-log>.open;
#<A:nt1-B:nt2>.notify {
   !<B:log2-log>.log
}
!<A:log1-log>.log
!<A:log1-log>.close
```

Formally, the composition of subcomponent frame protocols resulting in the architecture protocol is defined by the *consent operator* [AP05]. This operator is never used by the designer specifying the frame protocols, it is only a formalization of the behavior composition which is done automatically by the behavior protocol checker.

2.4. Static checking

2.4.1. Protocol compliance

One of the behavior properties, which can be statically verified with our behavior protocol checker, is compliance of an architecture protocol `PA` (of a component `C`) with the frame protocol `PF` (of `C`). Informally, there are two conditions which have to be satisfied in order for `PA` to be compliant with `PF` (let `F` be the frame of `C`): (1)`PA` specifies acceptance of any sequence of calls of the methods provided by `F` that are dictated by `PF`. (2) For such sequences, `PA` specifies only such calls of the methods required by `F` that are anticipated by `PF`.

Example of an architecture protocol not compliant with the frame protocol was already described in Section 1.2.2. As a more elaborate example of compliant behavior, recall the architecture protocol of `Client` from Figure 2.1 in Section 2.3...

```
!<A:log1-log>.open;
#<A:nt1-B:nt2>.notify {
   !<B:log2-log>.log
}
!<A:log1-log>.log
!<A:log1-log>.close
```

... and the corresponding frame protocol:

```
!log.open;
!log.log;
!log.log;
!log.close
```

The architecture protocol is compliant with the frame protocol, because if we abstract from the internal events of `Client` (which are not important from the point of view of compliance), and from different naming conventions (the architecture protocol uses binding names, the frame protocol uses interface names), both the protocols specify the same set of traces (or, in this particular case, the same trace).

We have developed two different formal definitions of behavior compliance: *pragmatic compliance*, published in [PV02], and *consensual compliance*, which uses the consent operator [AP05] and is implemented in the current version of the behavior protocol checker.

2.4.2. Composition errors

Composition errors are communication errors, which result from composition of components with incompatible behavior. If the definition of the components is enhanced by behavior protocols, those composition errors can be checked statically.

The first type of composition error is *bad activity*, which was demonstrated in Section 1.2.2. It occurs when a component `A` tries to call a method of a component `B` in such a way which in not specified in `B`'s behavior protocol.

No activity (or deadlock) occurs when computation in a component application can not progress (none of the components is able to emit an event), and at least one of the components has not finished its computation (the application thus cannot stop correctly).

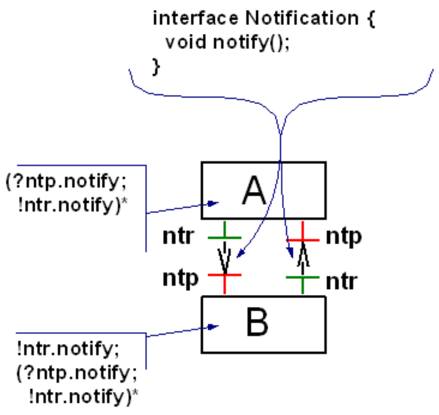
To show an example of no activity, let us modify the frame protocol of `Client`'s A subcomponent from Figure 2.1 in Section 2.3:

```
!log1.open;
!log1.log;
!log1.close
```

Here, the `B` component will be "blocked", as it expects a call of the `notify` method on its `nt2` interface - this call is never emitted by `A`. After `A` makes all the calls specified in its behavior protocol, a no activity error occurs.

An *infinite activity* (divergence) occurs when computation of a component application never stops, but components are never blocked, i.e. always there is an event which can be both emitted and absorbed.

Figure 2.2. Example of infinite activity



An example of a component composition resulting in infinite activity can be found on Figure 2.2. Here, the `A` and `B` components call forever the `notify` method on each other's `ntp` interface in turns.

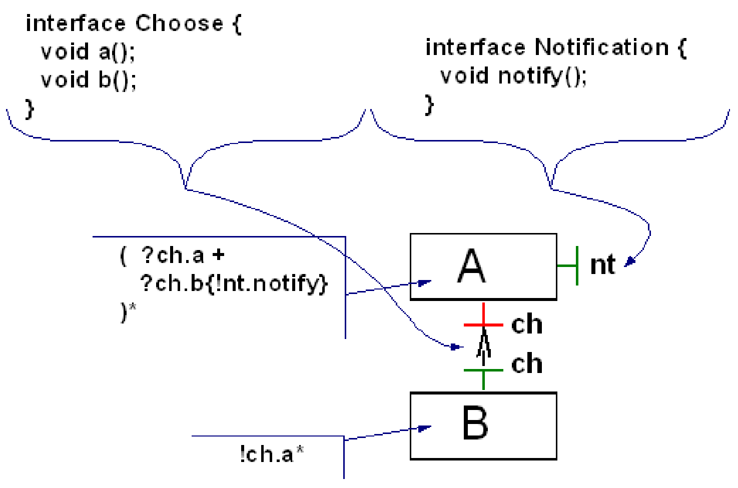
More information on composition errors can be found in [AP05].

2.4.3. Incomplete bindings

We say that a component architecture has *incomplete bindings*, if there exists an interface (either provided or required), which does not participate in any binding (we call such an interface an *unbound interface*). The existence of an unbound interface is not necessarily a design error: this typically happens when the designer reuses a component developed originally for different application and decides to utilize only a part of the component's functionality. If the behavior of the components in the architecture is specified using behavior protocols, it is possible to statically check whether the incomplete bindings cause a problem.

An unbound provided interface can cause bad activity or no activity (Section 2.4.2). On the other hand, an unbound required interface can cause a new type of composition error: *unbound requires error*. Unbound requires error occurs when a component tries to call a method on its required interface, which is unbound.

Figure 2.3. Example of incomplete bindings



An example of a component application with one unbound required interface (the `nt` interface of the `A` component) is shown on Figure 2.3. On the `ch` interface of `A`, the `a` or the `b` method can be called. If `b` is called, `A` reacts by calling `nt.notify`. As the `B` component calls only `ch.a`, the `A:nt.notify` method is never called and the fact that `A:nt` is unbound does not cause any problem. On the other hand, if the behavior protocol of `B` was `(!ch.b*)`, it would result in an unbound requires error.

More information on incomplete bindings can be found in [AP04].

2.5. Run-time checking

The run-time checker monitors the events on the external interfaces of a component (the trace) and checks whether this trace is one of those specified by the frame protocol of the component. If not, it is considered to be an error.

The main reason for using the run-time checker is verification of the composite components with dynamic architectures (which cannot be verified statically). Also, run-time checking is an alternative to static checking in the situations when the architecture of a (composite) component is so complex that the static checker cannot be used. Last but not least, run-time checker can be used to check the compliance of a primitive component behavior with the frame protocol (this cannot be done using the static protocol checker in principle, because there are no subcomponent frame protocols).

We show the functionality of the run-time checker on the example from Figure 1.1 in Section 1.2.1. The frame protocol of `Client` specifies that the `log.log` method has to be called after `log.open` has been called. If `log.close` were called instead at that moment, the run-time checker would detect an error.

What exactly happens when such an error is detected depends on the configuration of the run-time checker. Typically, the error is reported and logged within the runtime-checking framework. The run-time checking framework may throw an exception in the calling thread to notify the application about the erroneous call, or the application may continue without being affected. In either case, the run-time checking of the component (whose frame protocol was violated) is stopped. It is not possible to continue run-time checking of the component in this case, as the behavior protocols formal model does not support "error recovery".

The run-time checker also detects the violation of the frame protocol caused by the component's environment (the "outer world"). For example, if the frame protocol of `Client` were

```
!log.open;
```

```
!log.open;
!log.close
```

(i.e., to start by calling `log.open` twice) and `Client` behaved in compliance with this protocol (so that no protocol violation would be detected by the run-time checker for `Client`), error would be reported for `Logger`, as its protocol does not allow to accept a call of the `log.open` method twice.

2.6. Code analysis

The purpose of code analysis of a primitive component is to check whether the component's behavior is *bounded* by its frame protocol, that means checking whether the component can accept and emit method calls on its frame interfaces only in sequences that are determined by its frame protocol. Main advantage of code analysis over runtime checking is that all techniques of code analysis are exhaustive, i.e. they check all the possible runs of the verified code. We decided to employ model checking, which is one of the more popular techniques of software code analysis (see Chapter 3).

We show the idea of code analysis on the example from Figure 1.1 in Section 1.2.1. Assume that the frame protocol of `Client` is defined in the same way as in Section 1.2.1, i.e. it is

```
!log.open;
!log.log;
!log.log;
!log.close
```

and that the implementation of `Client` in Java is (with only fragments presented)

```
public class Client
{
 private Logger log;

 public void run()
 {
  log.open();
  ...

  log.log("message 1");
  log.log("message 2");
  ...
  if (/*some condition*/) log.log("message 3");

  ...
  log.close();
 }

 public static void main(String[] args)
 {
  Logger log = new LoggerImpl();

  Client client = new Client();
  client.setLog(log);

  client.run();
 }
}
```

It is clear from the example above that the implementation of `Client` is not bounded by the protocol, as the implementation allows three invocations of `log.log` in some cases whereas the protocol allows only two invocations.

Since the code analysis (via model checking) is an exhaustive verification technique, it will find the error when it checks the run in which the condition in the `if` statement evaluates to `true` and (consequently) `log.log` is called for the third time.

Chapter 3. Model checking of software components

Model checking [CGP] is a formal method of verification of finite state systems. The basic idea is that a model checker checks whether the model of a target system satisfies the property expressed in some property specification language. The checking is done by traversal of the state space that is generated from the model.

Some model checkers accept as input the model manually created by the user, while others are able to automatically extract the model from the source code. However, both approaches have severe drawbacks. Manual construction of the model is a tedious and error-prone process. On the other hand, automated extraction of the model faces the problem that the model is an abstraction and, therefore, it may represent behavior not possible in the original program. Consequently, a model checker may then find errors that are not present in the program (i.e., false negatives). Fortunately, there exist model checkers that work directly with the implementation of a target system - Java Pathfinder is an example of such a model checker.

Properties to be checked are usually expressed via temporal logic (CTL, LTL), or in the form of assertions. Some model checkers are also able to check for a fixed set of special properties (deadlocks, uncaught exceptions, etc).

The biggest problem of model checking with respect to practical use of this technique is the size of the state space typical for software systems (the problem of *state explosion*). However, decomposition of a software system into components helps to mitigate the problem. A component usually generates smaller state space than the entire system and, therefore, can be checked with fewer requirements on space and time.

In our case, we use model checking to check whether a primitive component is bounded by its frame protocol or not. And since most implementation of the Fractal Component Model are Java-based (including the reference implementation Julia), we decided to use the Java PathFinder model checker (JPF) [JPF].

3.1. Environment

Although model checking of individual software components helps to mitigate the problem of state explosion, a component cannot be checked in isolation because it does not form a complete program (with the `main` method) required by JPF. Therefore, it is necessary to create an environment of the target component and then check the whole program composed of the environment and the component.

The environment should be generated in a way that forces the model checker to verify all reasonable control-flow paths in the component's implementation. For that purpose, the environment has to (i) perform all reasonable sequences of method calls on server interfaces of a target component and (ii) invoke each method several times, each time with different values of its parameters.

We employed a tool for automated generation of environment that was developed outside of the scope of this project. As input, the tool accepts (i) the frame protocol of a target component as the behavior specification of the environment and (ii) the name of a Java class that works as a container for sets of values of method parameters. The environment is then generated from the inverted frame protocol [AP05] of the target component, which is constructed from the frame protocol by replacing all the accept events with emit events and vice versa. Our tool also performs several heuristic transformations of the frame protocol - before creating the environment - in order to minimize the size of the state space of the program composed from the component and environment [PP06].

3.2. Java PathFinder

Java PathFinder is a software model checker for Java byte code, which works as a specialized Java Virtual Machine (JPF VM). Unlike standard Java VM, the JPF VM executes the program in all possible ways with respect to threads' instructions interleaving and values of input data. Using this approach, the state space of the target program is generated on-the-fly, as JPF executes the program.

JPF integrates several methods for decreasing the state space size. Like majority of other model checkers, it supports partial order reduction (POR) [CGP]. It is based on the idea that some instructions (or sequences of instructions) are commutative when executed concurrently, i.e., they result in the same state regardless of the order of their execution. Actually, JPF implements POR in a slightly indirect way - it executes instructions of the current thread one after another till the current instruction is scheduling relevant (e.g. it accesses a shared variable, starts/stops a thread, blocks a thread, etc) or a value selection via the methods of the `Verify` class takes place.

Important feature of JPF is its extensibility via the publisher/listener design pattern, which allows to observe the course of the state space traversal and to check for specific properties in each state. This can be done at two abstraction levels: (i) virtual machine listeners provide low level VM information for checking of complex properties, and (ii) state listeners used for basic checks requiring information about visited states. This is especially useful, since by default, JPF checks the target program only for deadlocks, uncaught exceptions and assertions.

JPF also provides the MJI (Model-Java Interface) abstraction, which allows to execute certain methods in the underlying host VM instead of the JPF VM; this can be used to reduce the state space size. Use of the MJI abstraction is especially required in the case of native methods, which cannot be executed in the JPF VM.

Chapter 4. New features developed within this project

4.1. Protocol controller

In order to allow for the static and the runtime checking of a Fractal-based application, it is necessary to have an in-memory representation of the application architecture and the protocols associated with its particular components. For this purpose, we use the runtime representation of an application as it is just before starting. At this point, all components are instantiated in memory (but not running), thus their structure can be queried using content controllers and binding controllers. To associate a protocol with every component, we have created a protocol controller with the following interface.

```
public interface ProtocolController {

  /**
   * Returns the frame protocol associated with a component.
   */
  String getFcProtocol();

  /**
   * Assigns a frame protocol to a component.
   */
  void setFcProtocol(String protocol);
}
```

We implemented this controller for Julia in the form of a mixin-class (`org.objectweb.fractal.behprotocols.julia.ProtocolControllerMixin`).

4.2. Environment controller

For the purpose of automated generation of environment for primitive Fractal components, it is necessary to have an in-memory representation of the application architecture and other environment-related information - namely (i) the name of the Java class which works as a container for sets of values for method parameters, (ii) optionally, Java code for user-defined stubs and drivers, (iii) simplified version of component's frame protocol describing environment's behavior (also optional), and (iv) mapping between names of Fractal interfaces and names of classes that work as stub implementations of the interfaces. As in the case of static and runtime checking, we use the runtime representation of an application just before starting. To associate environment-related information with every primitive component, we have created an environment controller with the following interface.

```
public interface EnvironmentController {

  /**
   * Returns the name of a class with value sets.
   */
  String getFcValueSetsClass();

  /**
   * Assigns a name of the class with value sets to a component.
   */
  void setFcValueSetsClass(String valueSetsClass);
  /**
   * Returns the Java code for user-defined stub.
   */
  String getFcUserStubCode();

  /**
   * Assigns Java code for user-defined stub to a component.
   */
  void setFcUserStubCode(String userStubCode)

  /**
   * Returns the map of event names to Java code
   * for user-defined drivers.
   */
  Map getFcUserDriversCode();

  /**
   * Assigns a map of event names to Java code
   * for user-defined drivers to a component.
   */
  void setFcUserDriversCode(Map userDriversCode);

  /**
   * Returns the protocol describing the behavior
   * of the environment (not the inverted frame protocol).
   */
  String getFcProtocol();

  /**
   * Assigns a protocol describing the behavior
   * of the environment to a component.
   */
  void setFcProtocol(String protocol);

  /**
   * Returns a map of Fractal interface names to names
   * of manually-created stub implementation classes.
   */
  Map getFcItfStubs();

  /**
   * Assigns a map of Fractal interface names to names
   * of manually-created stub implementation classes.
   */
  void setFcItfStubs(Map itfStubs);
}
```

We implemented this controller for Julia in the form of a mixin-class (`org.objectweb.fractal.behprotocols.julia.EnvironmentControllerMixin`).

4.3. Extensions to Fractal ADL

As discussed in Section 4.1 and Section 4.2, we associate a frame protocol with each component of an application, and also some environment-related information with each primitive component of an application. Thus, to enable the users to use Fractal ADL for describing the architecture of Fractal applications, we had to extend the Fractal ADL syntax to accommodate the frame protocol and environment declarations.

We have extended the Fractal ADL by adding elements `protocol` and `environment` as children of `definition` and `component` elements. The following code shows an example of architecture definition written in the extended ADL (the extensions are highlighted in **bold**).

```
<definition name="LoggerDemo">
  <component name="client">
    <interface name="log" role="client" signature="logger.Log"/>
    <content class="logger.ClientImpl"/>
    <protocol value="!log.open;!log.log;!log.log;!log.close"/>

   <environment>
     <valuesets classname="logger.LoggerEnvValues"/>
   </environment>

  </component>
  <component name="logger">
    <interface name="log" role="server" signature="logger.Log"/>
    <content class="logger.LoggerImpl"/>
    <protocol value="?log.open;?log.log*;?log.close"/>

   <environment>
     <valuesets classname="logger.LoggerEnvValues"/>
   </environment>

  </component>
  <binding client="client.log" server="logger.log"/>
</definition>
```

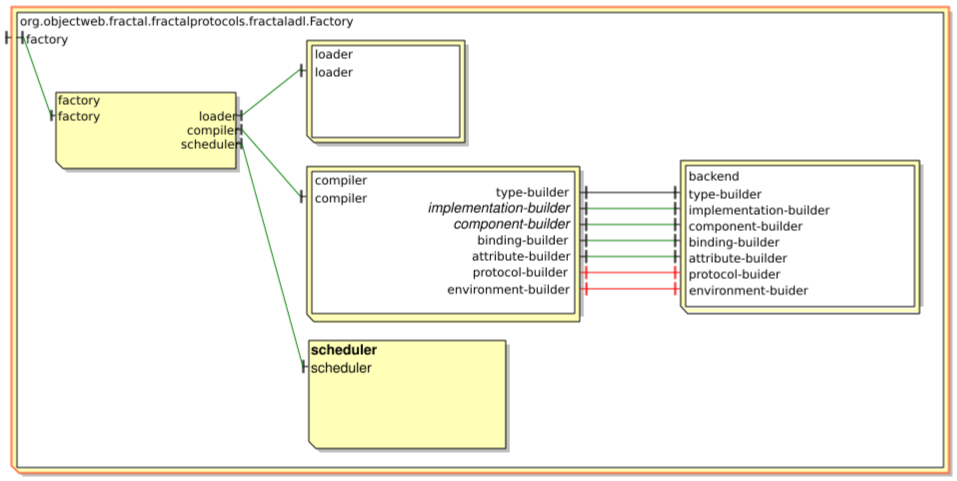
To be precise, we have changed the Fractal ADL DTD in the following way:

```
<!ELEMENT definition (interface*,component*,binding*,content?,
  attributes?,controller?,template-controller?,protocol?,
  environment?)>
<!ATTLIST definition
  name CDATA #REQUIRED
  extends CDATA #IMPLIED
>

<!ELEMENT component (interface*,component*,binding*,content?,
  attributes?,controller?,template-controller?,protocol?,
  environment?)>
<!ATTLIST component
  name CDATA #REQUIRED
  definition CDATA #IMPLIED
>


<!ELEMENT protocol EMPTY >
<!ATTLIST protocol
  value CDATA #REQUIRED
>



<!ELEMENT environment (valuesets,userstub?,userdriver*,protocol?,
   itfstub*)>
```

```
<!ELEMENT valuesets EMPTY >
<!ATTLIST valuesets
  classname CDATA #REQUIRED
>

<!ELEMENT userstub EMPTY >
<!ATTLIST userstub
  file CDATA #REQUIRED
>

<!ELEMENT userdriver EMPTY >
<!ATTLIST userdriver
  event CDATA #REQUIRED
  file CDATA #REQUIRED
>

<!ELEMENT itfstub EMPTY >
<!ATTLIST itfstub
  name CDATA #REQUIRED
  classname CDATA #REQUIRED
>
```

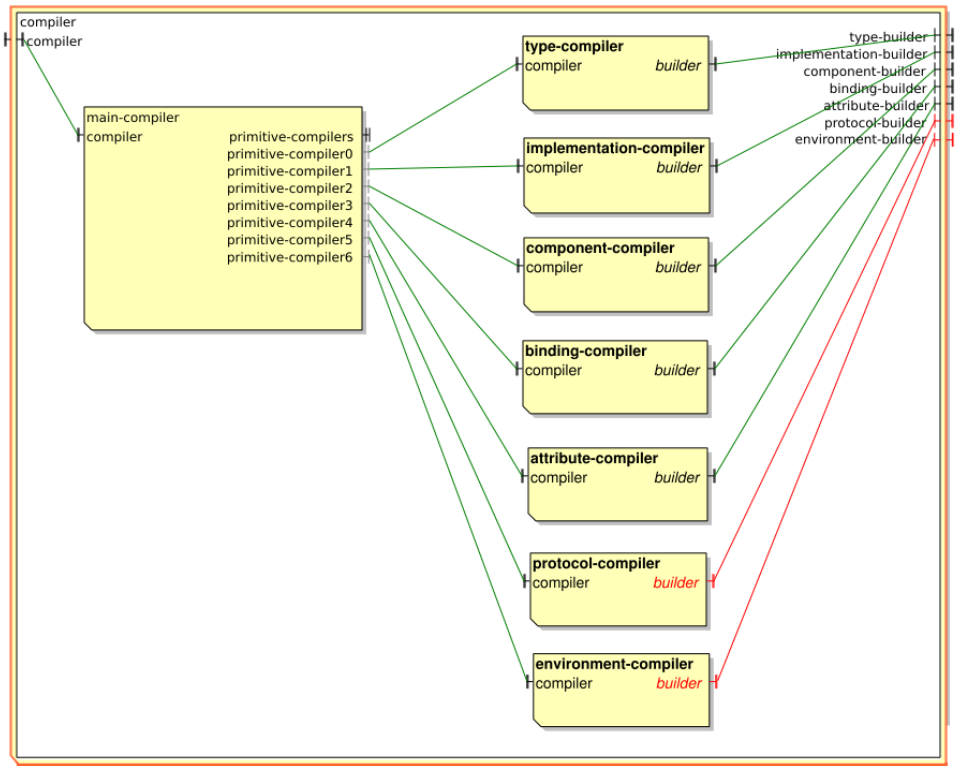
The Fractal ADL framework has been built as a component-based application. This allows us to easily extend it with new features (such as handling the `protocol` and `environment` elements). The top-level architecture is shown in Figure 4.1. It divides responsibilities to the loader, which parses the ADL, the compiler, which checks its validity and processes it, and to the backend which builds the application being described by the input ADL. Our modification to the Factory are denoted by red color; we have added interfaces for handling `protocol` and `environment` declarations to the compiler and the backend component.

Figure 4.1. Fractal ADL factory with support for a protocol and an environment



We have modified the compiler component by adding subcomponents for processing the `protocol` and `environment` declarations and passing it to the backend. The Figure 4.2 shows the extended compiler component.

Figure 4.2. Fractal ADL compiler with support for a protocol and an environment

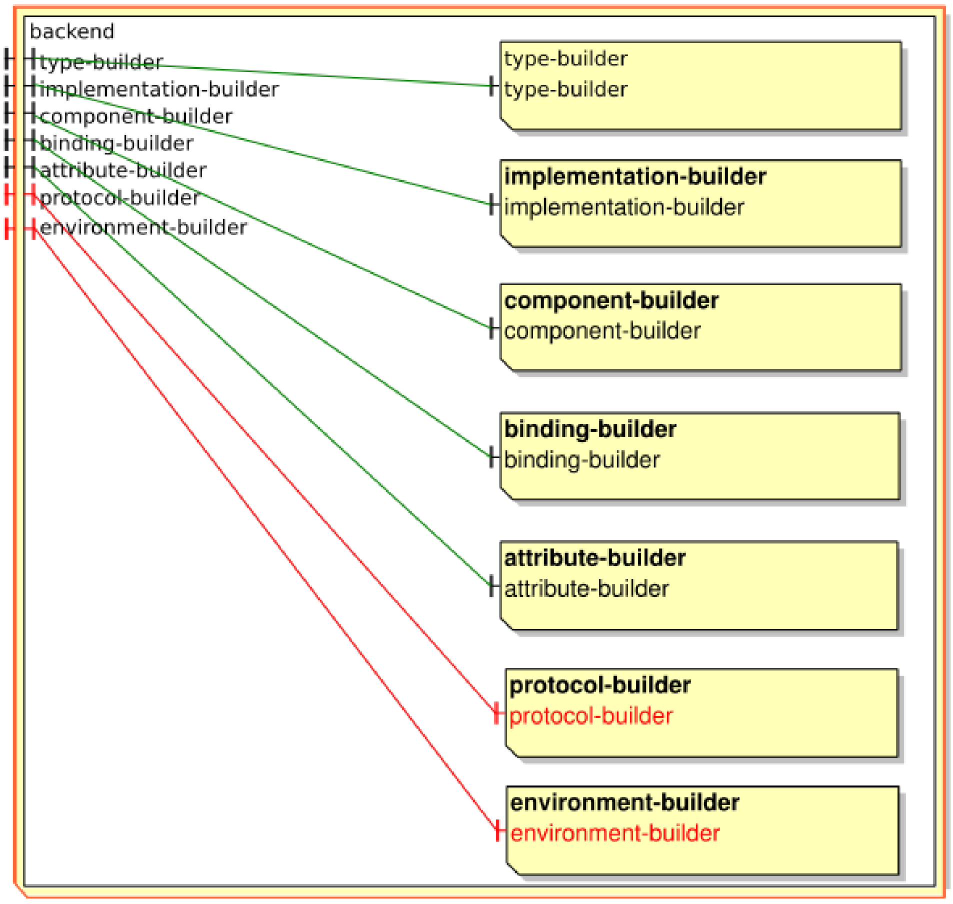


Fractal ADL allows for different backends. The choice of a backend influences how a resulting component application is built. As for now, there are four different backends available: Fractal, static Fractal, Java, and static Java. Fractal and static Fractal use Fractal API to instantiate and run components. The difference between the standard and static variant is that the standard variant directly instantiates and runs the components, while the static variant generates Java-code, which (when executed) performs all the instantiation and execution steps. The Java and static Java backend work the same way, only they do not use Fractal API to instantiate and run the components, they rather instantiate the components as ordinary Java classes.

Our approach to behavior checking relies on having runtime information about components' structure and protocols associated with them, which is not easily possible with the Java backends. The use of the static Fractal backend does not make a good sense for static checking of protocol compliance. Thus, we have decided to support only the standard Fractal backend.

We have extended the backend to handle a `protocol` element by calling `setFcProtocol` method on the protocol controller associated with a respective component, and to handle an `environment` element by calling `setFcValueSetsClass`, `setFcUserStubCode`, `setFcUserDriver-sCode`, `setFcProtocol` and `setFcItfStubs` methods on the environment controller associated with a respective component. The architecture of the extended backend is shown in Figure 4.3

Figure 4.3. Fractal ADL backend with support for a protocol and an environment

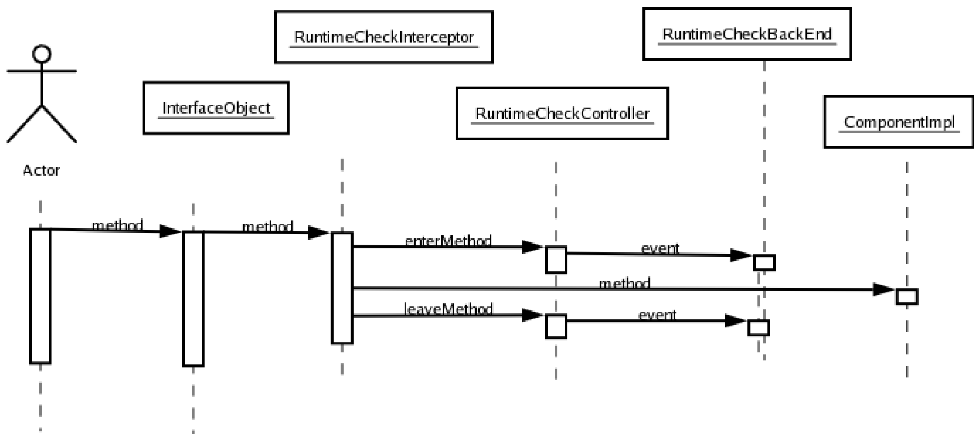


4.4. Interceptors

While extending Fractal and Julia with support for runtime checking of compliance of component behavior with the specified protocol, we have encountered a number of issues, some of which have required modifications to Julia. In this section, we describe the Fractal and Julia extensions we developed to support the runtime checking.

In principle, runtime checking is achieved by introducing an interceptor for each business interface of the component being checked; on each event (method entry or exit), this interceptor notifies the *runtime-check controller* introduced into the controller part of the component. This controller creates an instance of the runtime-checker backend with the specified protocol, and notifies the checker backend of each such event. In case the checker detects that the event violates the protocol, the error is recorded; optionally, the application may be notified by throwing a `ProtocolViolationException`. The typical interaction among these parts is shown in the sequence diagram in Figure 4.4.

Figure 4.4. Sequence diagram capturing interaction in the runtime-check subsystem



4.4.1. Identity-aware interceptors

The Julia interceptor framework features several Interceptor generators. Of these, the `SimpleCodeGenerator` at the first sight seems to perform the task required by the runtime-checking extensions - deliver a method call to a controller whenever a method call starts or completes, and also provides the interface name. However, `SimpleCodeGenerator` can only provide the name of the language type used by the interface, or a possibly configurable string, which can however only depend on the language type of the interface, and cannot provide the concrete name of the interface. This difference is particularly obvious when a component features multiple interfaces based on the same language type. Furthermore, Julia originally did not provide a way to configure an interceptor (such as to provide it with getter/setter methods to set configuration properties), as it was not possible to specify an interface to be implemented by an interceptor, and it would not be possible to call such a method without the use of reflection.

The first enhancement to Julia was to allow an interceptor generator to specify Java interfaces to be implemented by the generated interceptor class. For a class generator, this had already been possible by overriding the `getImplementedInterfaces` method specified in the `ClassGenerator` interface. We have introduced the `getImplementedInterfaces` method also into the `CodeGenerator` interface, and extended `InterceptorClassGenerator.getImplementedInterfaces` to merge requirements from all its subordinate `CodeGenerator` objects.

To handle this modification of the `CodeGenerator` interfaces, we have provided a default implementation of this newly introduced method into all the Julia classes implementing this interfaces, `SimpleCodeGenerator` and `MetaCodeGenerator`. These extensions have been committed to the Julia CVS repository and have been included in the recent release of the Fractal project (2.3.1).

The subsequent task was to use these extensions to introduce *identity aware interceptors*. Here, we consider the identity of an interface to consist of its name, `isClient` value, contingency (mandatory/optional), cardinality (singleton/collection), and signature type. While the runtime checking framework is particularly interested only in the name and `isClient` value, we have decided to introduce more general extensions, realized in the `IdentityAwareInterceptor` interface (see Figure 4.5). Here, an additional way to express the identity of an interface is via a reference to the interface object, which allows to obtain the interface type via the `getFcItfType` method to access the additional attributes.

Figure 4.5. Declaration of interface `IdentityAwareInterceptor`

```
public interface IdentityAwareInterceptor {
 public ComponentInterface getFcItfInstanceRef();
 public void setFcItfInstanceRef(ComponentInterface itfRef);
 public String getFcItfInstanceName();
 public void setFcItfInstanceName(String newItfInstanceName);
 public void setFcItfIsClient(boolean itfIsClient);
 public boolean getFcItfIsClient();
}
```

The interceptor code generator is responsible for providing implementations of these methods. In the case of the `RuntimeCheckInterceptorCodeGenerator` provided in our framework, these methods are generated via the ASM toolkit; the method implementations are simple accessor (getter/setter) methods for the respective local private attributes.

Please note that we have been also considering an alternative approach: instead of generating the methods for each interceptor class, these methods might also be inherited from a common base class. However, the current Julia interceptor framework does not permit selecting the base class of an interceptor, and it is not feasible to make it configurable without significantly changing the structure of component descriptors.

Additional issue related to the introduction of identity aware interceptors into Julia was assigning the responsibility to initialize the interceptors. While it was originally considered that this task would be done by the newly introduced `RuntimeCheckController` controller object, this approach would not address interceptors associated with collection interfaces. For a collection interface, the interface object is created by cloning a template interface object only at the time the particular interface name is used for the first time. A new instance of the interceptor object is created (cloned) while cloning the interface object. To properly handle this situation, the responsibility for initializing the interceptor object must be assigned to the interface object.

In the newly introduced `BasicIdentityAwareComponentInterface` class, we have overridden the `setFcItfName` method of the `BasicComponentInterface` class to call the setter method of the interceptor object, if the interface has an interceptor and the interceptor implements the `IdentityAwareInterceptor` interface. Hence, to properly handle collection interfaces, it is necessary to use a customized interface object, using `BasicIdentityAwareComponentInterface` instead of `BasicComponentInterface` as the base class. We show the relevant fragment of the configuration file in Figure 4.6

Figure 4.6. Fragments of the Julia configuration file related to interface objects.

```
(interface-class-generator
  (org.objectweb.fractal.julia.asm.InterfaceClassGenerator
    org.objectweb.fractal.behprotocols.\
        julia.BasicIdentityAwareComponentInterface
  )
)
```

We would also like to document one technical aspect related to future extensions of Julia. In its initial design, Julia was supposed to support reconfiguration of a single component instance, in particular, optimizing/deoptimizing the component. This vision included also dynamically introducing/removing interceptor objects; this would likely be done via the `setFcItfImpl` method of the `ComponentInterface` interface. As such a reconfiguration is not used in Julia, we do not provide any special means to handle it - i.e., to update the interface identity stored in the identity aware interceptors possibly involved. Should a need arise to do so, it would be possible to modify Julia to support such a reconfiguration. Changes would have to be introduced into the `setFcItfImpl` method featured by interface objects; as the implementation of this method is generated by the `InterfaceClassGenerator`, it would be necessary to modify the way it is generated. The method could either update the identity directly, or it might call a method inherited from the interface object base class; introducing the `setItfImpl` method into the base class would also make future extensions related to dynamic reconfiguration much easier.

4.4.2. Controllers: RuntimeCheck and LifeCycle

The key responsibility of the `RuntimeCheckController` is to manage the checker backend, to collect events from the interceptors, and to pass these events to the checker backend. Furthermore, the `RuntimeCheckController` may also collect information on the component execution, capturing its *execution trace* and the list of method calls currently in progress; this information may be used by a monitoring toolset. The interface of the `RuntimeCheckController` is shown in Figure 4.7. The controller functionality is implemented in the `BasicRuntimeCheckControllerMixin` class.

Please note that the event tokens are internally stored as strings; the notation is the same as the one used by the checker backend, i.e., the event token string starts either with an exclamation mark (`"!"`) for an event emitted or with a question mark (`"?"`) for an event absorbed, followed by the name of the interface, concatenated with a dot (`"."`) with the name of the method, followed by a either the character `"^"` to denote a method request, or by `"$"` to denote a method response. For a pair of events forming a single procedure call, the initial character of the request (`"?"` or `"!"`) is the opposite to the initial character of the response. Both the operations `getFcCurrentMethods()` and `getFcMethodHistory()` return an array of strings following this format.

The runtime checking subsystem is inherently tied with the lifecycle of the component being monitored. When the component starts, monitoring has to start, with the protocol configured for the component. When the component stops, it is necessary to verify that the protocol permits to stop at the given point in the component's execution history, i.e., whether the corresponding automaton managed by the checker backend is in an accepting state.

To properly address there requirements, we have put the responsibility to manage the lifecycle of the `RuntimeCheckController` to the life-cycle controller; in Julia, this is realized via the class `RuntimeCheckLifeCycleMixin`, to be included in the lifecycle controller object. The `setFcStarted()` method of this mixin obtains the protocol configured for the component from the `ProtocolController`, and uses this protocol to initialize the `RuntimeCheckController`. The `setFcStopped()` method stops the runtime check controller, which verifies that the protocol permits to stop.

Figure 4.7. Declaration of interface `RuntimeCheckController`

```
public interface RuntimeCheckController {
 public void enterFcMethod(String itfName, String methodName,
   boolean isClient, Object params[]);
 public void leaveFcMethod(String itfName, String methodName,
   boolean isClient, Object params[]);
 public String[] getFcCurrentMethods();
 public String[] getFcMethodHistory();
 public void startFcRtcheck(String protocol);
 public void stopFcRtcheck();
}
```

4.4.3. Handling protocol violations

An important issue to decide is what action should the runtime checking system take when it detects a protocol violation. In such a situation, it is already known that the application violates the protocol specified for the particular component, but that may not be a sufficient reason to terminate the application. Instead, it may be useful to collect more information on this application failure, such as collecting the subsequent events observable on the interfaces of the faulty components. The default behavior is to log and report the error (including the execution trace so far collected for the component, and also the current stack trace); optionally, a runtime exception (`ProtocolViolationException`) may be thrown to inform the application that the attempted call is not permitted by the protocol. Here, please note that raising the exception prevents the method call from actually occurring when the erroneous event detected is a request event, but obviously cannot prevent the call in the case of a response event. The error handling policy can be configured via JVM properties, described in the following section.

4.4.4. Technical notes

The runtimecheck subsystem may be configured via the following JVM properties:

`fractal.protocols.rt-check.recorderrors` (values: `true` or `false`; default: `true`)	sets whether the runtime check controller should record all erroneous events.
`fractal.protocols.rt-check.recordtrace` (values: -1, 0 or a positive integer; default: -1)	sets how many recent events should be kept to aid with locating the source of an error. Special values: -1 (unlimited storage) and 0 (no events recorded).
`fractal.protocols.rt-check.stoponerror` (values: `true` or `false`; default: `true`)	sets whether runtime-checking should stop for a component when a violation of the component's protocol is detected. If false, the erroneous event is ignored and checking resumes from the current position in the state-space.
`fractal.protocols.rt-check.throwerrors` (values: `true` or `false`; default: `false`)	sets whether an exception should be thrown when a protocol violation is detected.
`fractal.protocols.rt-check.verbosity` (values: 0, 1, 2 or 3; default: `1`)	sets the level of output on `stderr` produced by the runtimecheck subsystem. (0: no output, 1: only protocol violations, 2: report on controller initialization and successful completion, 3: report on event processing.)

4.5. Extensions to protocols

4.5.1. Multiple bindings

In Fractal, any interface can participate in more than one binding (if this is the case, we say that the interface has *multiple bindings*). As behavior protocols were originally developed for a component model, where every interface can have at most one binding, this alternative did not come into question when the algorithm for architecture protocol construction was designed. Therefore, the algorithm had to be revisited for Fractal.

Let `C` be a component whose subcomponents `S1`, ..., `Sn` have the frame protocols `F1`, ..., `Fn`. The classical construction of `C`'s architecture protocol (not considering multiple bindings) is done in two steps. In the first step, the interface names in the frame protocols `F1`, ..., `Fn` are replaced by the binding names (names of unbound interfaces remain unmodified). In the second step, the protocols are composed using the consent operator.

The purpose of the first step of the algorithm is to ensure that the emission and absorption of any event (specified in different frame protocols) is denoted by tokens which differ only in the prefix (`?` or `!`). If this was not the case, the consent operator would not work correctly.

To guarantee proper functionality of the consent operator also in the presence of multiple bindings, the first step of the algorithm has to be modified. The idea behind the modification is the following: If a provided interface has multiple bindings and the protocol of its component denotes acceptance of a method call on the interface, the call will be absorbed from just one of those bindings. On the other hand, if a required interface has multiple bindings and the protocol of its component denotes that a method call can be emitted on the interface, the method will be called on all of the bindings (multicast). As taking an assumption on the order of the calls would be too restrictive, the calls on those bindings are considered to happen in parallel. Every particular ordering of the calls is compliant with this assumption.

Formally, the protocols are transformed as follows:

(a) Names of unbound interfaces remain unmodified.

(b) Name of an interface which has exactly one binding is replaced with the name of that binding.

(c) If `P` is the name of a provided interface of a subcomponent `Sk` with multiple bindings `<C1:I1-Sk:P>`, ..., `<Cm:Im-Sk:P>`, absorption of a method call on `P` in the frame protocol of `Sk` of the form `?P.a` is replaced with the protocol

```
?<C1:I1-Sk:P>.a + ... + ?<Cm:Im-Sk:P>.a
```

In a similar way, absorption of a method call of the form `?P.m{Q}`, where `Q` is an arbitrary protocol, is replaced with the protocol

```
?<C1:I1-Sk:P>.a{Q} + ... + ?<Cm:Im-Sk:P>.a{Q}
```

(d) If `R` is the name of a required interface of a subcomponent `Sk` with multiple bindings `<Sk:R-C1:I1>`, ..., `<Sk:R-Cm:Im>`, emission of a method call on `R` in the frame protocol of `Sk` of the form `!R.a` is replaced with the protocol

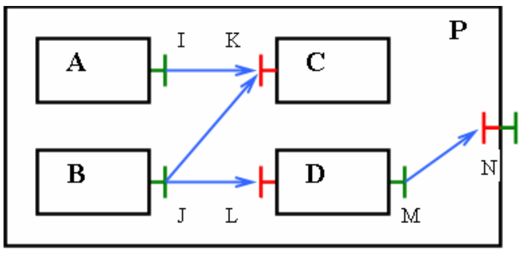
```
!<Sk:R-C1:I1>.a | ... | !<Sk:R-Cm:Im>.a
```

In a similar way, emission of a method call of the form `!P.a{Q}`, where `Q` is an arbitrary protocol, is replaced with the protocol

```
!<Sk:R-C1:I1>.a{Q} | ... | !<Sk:R-Cm:Im>.a{Q}
```

(e) Explicit requests/responses on the interfaces with multiple bindings are forbidden.

Figure 4.8. Example of multiple bindings



We demonstrate the rules on the application shown on Figure 4.8. The frame protocol of `B` of the form `!J.x*` (`x` is name of a method) will be transformed to

```
(!<B:J-C:K>.x | !<B:J-D:L>.x)*
```

The frame protocol of `C` of the form `?K.x*` will be transformed to

```
(?<A:I-C:K>.x + ?<B:J-C:K>.x)*
```

The rest of the protocols is transformed in the classical way.

4.5.2. Atomic actions

4.5.2.1. Overview

Atomic actions (AA) are a behavior protocols construct allowing cooperating components to synchronize. They have been added to behavior protocols as a consequence of component synchronization problems which arised during the work on specification of the Airport Internet Access Application components. Although in some cases the behavior of a component may be described using behavior protocols without AA, a version using AA are usually not only much easier to construct, but also more readable afterwards. Furthermore, using AA, behavior protocols correspond with component implementation in a more straightforward way. As an example of a behavior protocol containing an atomic action (enclosed in square brackets '[' and ']'), consider the following example:

```
?IDhcpController.Start^ ; !IListenerController.Start^ ;
[?IListenerController.Start$, !IDhcpController.Start$]
```

4.5.2.2. Syntax

An atomic action may occur in a behavior protocol at positions where a single event and an abbreviation may. Atomic action starts with '[' and ends with ']'. There is a coma-separated list of events (the use of abbreviations is not allowed as their use doesn't make sense here) between '[' and ']'.

4.5.2.3. Semantics

Basically, an atomic action is treated as a single event, i.e., it is supposed to be "executed" in a single step. An atomic action is in one of the two states - *enabled* or *disabled*. It can be executed in the enabled state only. An atomic action is enabled in the current state if and only if for each accept event (an event starting with '?') in the atomic action there exists a component in the composition able to emit the corresponding request event in the current state. If there's not a component able to emit a request event corresponding to an accept event of the atomic action, the atomic action is disabled. The corresponding accept and request events yield, as in a common case, a tau action; consider the following protocol fragment:

```
...[?ma^, !mc^]... (consent) ...!ma^... ->
    ...[#ma^, !mc^]...
```

The application of the consent operator to behavior protocols containing atomic actions may result in a protocol containing the bad activity composition error. This situation arises in the following case: The atomic action contains no accepting event (an event starting with '?'), i.e., it contains internal and emitting events (events starting with '#' and '!', respectively) only, and there's an emit event in the atomic action that is not accepted in the current state by any component in the composition.

4.5.2.4. Notes

For each two components combined via the consent operator there may be at most one event inside of an atomic action that is also contained in the set of synchronization events for these two components. This requirement reflects the fact that a component cannot perform more than one event (a simple event or an atomic action) in a single step, which causes the consent operator not to be associative when applying to behavior protocols containing atomic actions. In other words, the result of the composition depends on the order the components are composed together.

4.6. Checker enhancements

4.6.1. Static checking

4.6.1.1. Incomplete bindings

The behavior protocol checker is able to detect incomplete bindings (Section 2.4.3). If a set of operations of unbound interfaces is given to the checker, the methods on provides interfaces are supposed not to be called and in the case a method of an unbound required interface is called the checker detects and reports the unbound interface call error. This type of error is detected on the top level (i.e., the place of the last use of consent) and the time requirements are therefore acceptable. There is no command line option for turning the detection of unbound requirements off; instead, an empty set of these operation can be passed as the last parameter in order not to check for incomplete bindings.

4.6.1.2. Collection interfaces

In Fractal, a component type is a list of interface types. Every interface type specifies (in addition to the name, signature, role, and contingency) also the cardinality of the interface type: singleton or collection.

As the interfaces defined by a collection interface type are created lazily, their names are in general not known at compile time. However, behavior protocols can specify only the traffic on the interfaces with known names. Therefore, the checker supports only singleton interfaces and collection interfaces with names known at compile time.

4.6.1.3. Multiple bindings

Handling multiple bindings basically means to replace interface names in frame protocols as described in Section 4.5.1. This is implemented by parsing the frame protocols and replacing parts of the parse trees.

4.6.1.4. Atomic actions

Atomic actions (Section 4.5.2) are handled by the checker as standard actions; the binding of an atomic action has the same time requirements as the binding of standard actions (as there may be only one single action inside an atomic action that can be bound on a single component binding).

4.6.1.5. Substantial performance improvements

Since its first version, a lot of new features have been added to the behavior protocol checker. They include state space and parse tree visualization, consent operator, atomic actions, runtime checking and the Fractal interface. Although the implementation of new features has required substantial changes to the code of checker, resulting into a more complex and more time-consuming application, the performance has actually improved, by implementing a new state representation and a faster state space generation aglorithm.

As the checker uses on-the-fly state space generation, a suitable and efficient state representation for storing information about visited states and for state comparison is needed. In the current version of the behavior protocol checker, a state is represented as a bit-field. Management of such state identifiers is easy and very fast, however the drawback of this representation is that (because of possible non-determinism) it is not possible to determine the exact state identifier size in advance (sizes for different states may even differ). Thus, there may be some unnecessary memory reallocation needed during checker computation, but this is probably inherent for any "memory-reasonable" representation.

Since the generation of possible transitions from the current state is the far most time-consuming operation of the compliance checking process, this operation is optimized for the best performance using state pregeneration and by computing all the information not depending on the current states in advance.

This optimization has further improved the performance of the checker without significant increase in memory requirements.

For more information about the optimization included in the checker please refer to Section 5.1.3.

4.6.2. Run-time checking

The checker is able to check if the real-time behavior of a component conforms to its declared behavior specified by its frame protocol. The runtime checker does not perform an exhaustive traversal through the state space defined by the protocol (as in the case of static checking), but the state space traversal is driven by the information about method calls provided by the component interface interceptors. Should an event violating the frame protocol occur, i.e., the event is not among events allowed at this particular point (with respect to the history of events), there are two options: (1) the application is stopped or (2) an error message is printed to the output and the application continues, but no further checking is performed (as there is no method known for recovering from such a state). At the end of the application run, the checker provides information whether the component has successfullly satisfied its frame protocol (i.e., whether an accepting state has been reached).

4.7. Cooperation of Java PathFinder with protocol checker

As already said in Chapter 3, we use JPF for checking primitive Fractal components implemented in Java against behavior protocols. However, it is not directly possible to use JPF for checking whether a primitive component is bounded by a protocol, because JPF is, by default, able to check only properties like deadlocks and assertions. In order to solve this, we decided to use JPF in combination with the protocol checker for code analysis. In other words, we decided to let JPF and the checker cooperate on code analysis while traversing their own state spaces. Since JPF and the checker work at different levels of abstraction, we had to define a mapping from the JPF state space into the state space of the checker to make such cooperation possible. For more information on the mapping, please refer to [PPK].

4.7.1. Checker for code analysis

We have modified the behavior protocol checker for static testing by adding several methods to make the cooperation with JPF possible. In particular, the checker has been enriched by a method for notification of actions performed (method called and finished) in the JPF and uses this for coordination of the state space traversal. Each time JPF moves along a transition corresponding to a method call or return from a method call, it notifies the checker of this event. Checker moves along the corresponding transition in its own state space. Should not such a transition exist within the checker's state space, an error is reported to the user and the implementation is considered not to be bound by the protocol. To treat all the combination of implementations and protocols correctly as well as to be able to handle cycles, it is necessary to coordinate the traversal in the following way: Each time JPF would backtrack within the state space because of being in an already visited state it asks the checker for permission. Only in situations when both JPF and the checker would backtrack at this point when executed on their own (i.e., if being in an already visited state), backtracking is allowed. Hence, the bounding relation can be checked correctly.

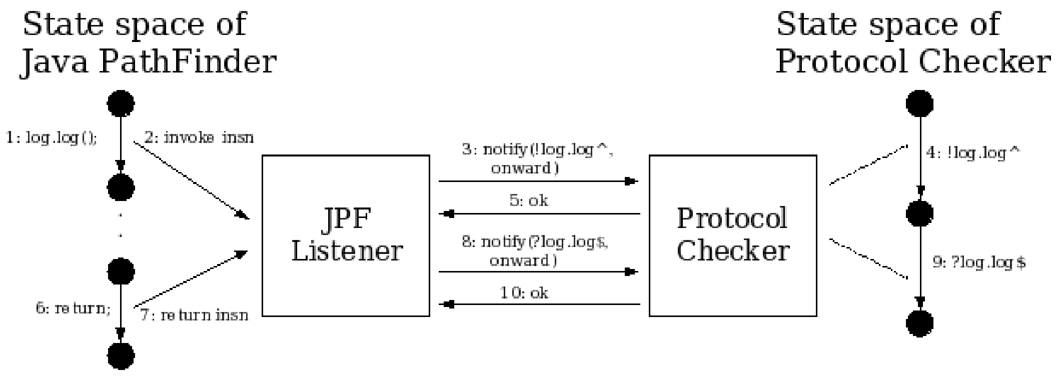
4.7.2. Extensions to the Java PathFinder

The mapping between JPF and the checker needed for code analysis is implemented via a JPF listener (i.e., via a plugin for JPF). During traversal of the JPF state space, the listener traces execution of all invoke and return byte code instructions that correspond to methods of the provided and required interfaces of a target component, and notifies the checker about such instructions. This way, the listener instructs the checker what transition to take in its state space. The notification is done also during backtracking in order to instruct the checker to also backtrack.

Additionally, the JPF listener also notifies the checker when it reaches an end state. In that case, if the protocol checker is not in an end state of its state space, an error is reported. This can happen, for example, when JPF comes to the end of the `main` method in its state space, but the checker still expects some more events to occur.

Communication between JPF and the checker during the checking of the `Client` component (see Section 2.6) is shown on Figure 4.9. The left part shows the JPF state space and the right part shows the state space of the checker; numbers determine the order of related activities in JPF and the checker.

Figure 4.9. Communication between JPF and Checker during traversal of state spaces in the onward direction


State space of Java PathFinder
State space of Protocol Checker
1: log.log();
2: invoke insn
6: return;
7: return insn
JPF Listener
3: notify(!log.log^, onward)
5: ok
8: notify(?log.log$, onward)
10: ok
Protocol Checker
4: !log.log^
9: ?log.log$

While implementing the mapping between the JPF state space and the checkers' state space, we had to make two modifications to the JPF source code. First, we had to modify the code responsible for partial order reduction, so that a transition between states is terminated when an invoke or return instruction corresponding to a method of a frame interface of a target component is executed. Second, we had to enhance the JPF search engine, which drives the traversal of the state space, so that JPF asks the checker for a permission to backtrack - we call this *coordination of backtracking*. For motivation of the changes to JPF, please see [PPK].

Chapter 5. Implementation

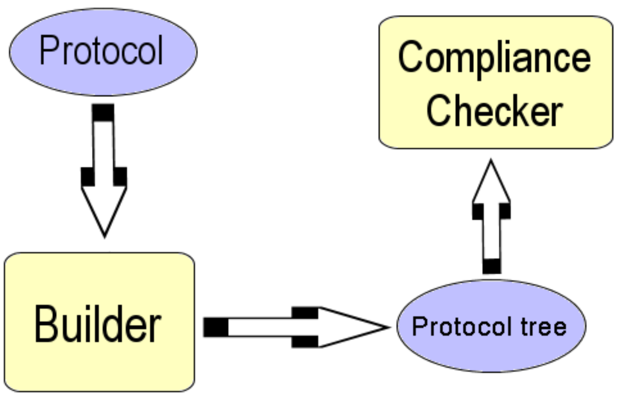
5.1. Behavior protocol checker - static version

5.1.1. Implementation overview

The behavior protocol checker was implemented in Java (using sdk1.4.2_03, version 1.5 is not supported since Java PathFinder doesn't support the 1.5 version bytecode) and a preliminary version has been a part of the SOFA technology (see http://sofa.objectweb.org). The checker has been substantially enhanced; now the checking process is much more efficient in both memory and time requirements. A rough structure of the behavior protocol checker is depicted on Figure 5.1. The protocols to be checked are parsed by the parser (`Builder`) and the trees representing the protocol structure are built. Hence, for illustration we will use the following protocols:

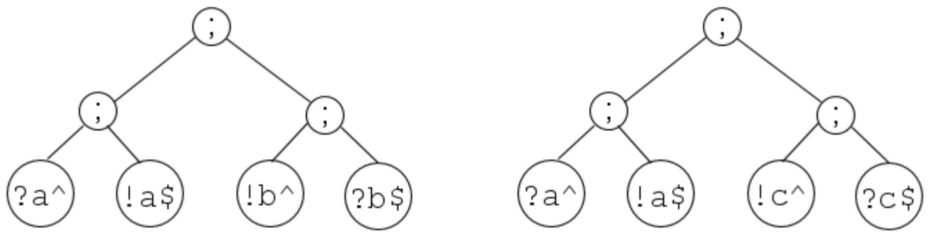
```
?a; !b
?a; !c
```

Figure 5.1. Basic structure of the checker



The parse trees representing these two protocols are on Figure 5.2. To find composition errors of a set of n components connected together via their interfaces, protocols (their parse trees) of these components are combined together using the (binary) consent operator (consent operator is applied (n-1) times - each time one component is composed with the result created so far). Using consent operator for component composition enables us to detect three types of errors: bad activity, no activity and infinite activity. The resulting structure (i.e., the parse tree of the composed protocol) is used to generate the state space. The consent operator itself can detect bad activity and no activity errors. As the infinite activity is not a property of a single state, this error is detected within the traverser component of compliance checker. The strategy used for traversing the state space is known as Depth First Search. Should an error (bad-, no- or infinity-activity) be detected, the traversing is stopped and the checker reports to the user the error type found and an error trace describing the problem found.

Figure 5.2. Parse trees for ?a; !b and ?a; !c



As the state space of a more complicated protocol may be very large, the memory available for the checking may become insufficient (state explosion problem). To solve this problem, i.e., to be able to check compliance of such protocols, we use on-the-fly automata that are generated during the computation as needed. This greatly enhances the usability of the checker. The drawback of this method is the lower speed of the checking compared to the "state space pregeneration" approach. To improve the performance we use optimizations such as explicit automata, forward cutting, multinodes (for more information see [PTA]).

5.1.2. Basic structure and interaction

The checker can be used as a standalone tool, and is also integrated into the Fractal environment. Interaction with the Fractal application is realized via the `FractalStaticChecker` class. The first thing to be done here (after protocol transformation to handle multiple bindings) is parsing the input protocols and building parse trees. While constructing the parse tree, the multinodes optimization is applied: if there are more than two operands of the same binary operator (sequence, alternative, and-parallel), where a subtree (Figure 5.3) should be build, the nodes are instead collapsed into a single multinode (Figure 5.4).

Figure 5.3. Original parse tree

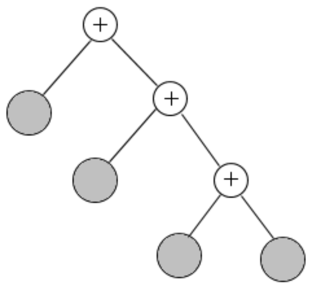
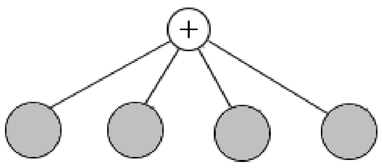


Figure 5.4. Multinode optimization



The resulting parse tree represents the same protocol as the original one. This optimization can be easily performed at this point without any loss of parsing speed while saving both the time and the space needed later during the checking. After composing all the input protocol parse trees via the consent operator a composition and protocol compliance check can be performed.

5.1.3. Optimizations

Before the protocol checking is performed, other optimization is done - building explicit (i.e., pregenerated) automata for small parse subtrees; this sometimes enhances the speed of the subsequent checking.

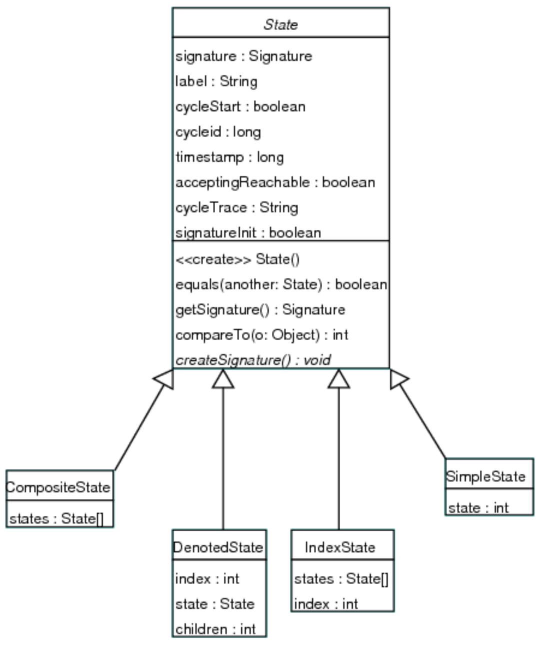
Because cycles and forward edges may appear in the transition graph of the automaton, the use of a global state cache improves the checking speed, since states may be visited and walked through more than once. The problem arising with the use of such a global state cache is again caused by the size of state space - in some cases it can be simply impossible for all the states visited so far to fit into the cache because of the limited amount of memory available. The solution used here is to "forget" some of the states being stored in the cache when the cache size exceeds a specified size. This of course decreases the checking speed, but the performance is still better than in the case of not using the global cache at all. The variant of the DFS algorithm with "forgetting" states from the global cache is called Depth First Search with Replacement (DFSR).

5.1.4. The composition and conformance test

As mentioned above, a test basically means searching for an error state causing a compliance violation of the given protocols. In the process of traversing the state space, the comparison of states is necessary. Since the automaton (i.e., the states and transitions to other states) is generated on-the-fly, the comparison of states via comparison of their references simply doesn't work. Therefore the approach of state signatures representing the internal structure is used here. A state signature represents both the shape (the structure) of the corresponding parse tree and the position in it. For example, the state of the automaton representing the protocol `?a^;!a$;!b^;?b$` when the trace having been traversed is `?a^;!a$` has the signature `1100` denoting the path from the root node to the leaf `!a$`. The second digit of the signature (`1`) expresses that the second action has been already performed. As the state of each simple automaton (i.e., automaton accepting exactly one word (e.g., ?a^)) is represented by a single bit, both state comparison based on these signatures and their management is very fast and compact.

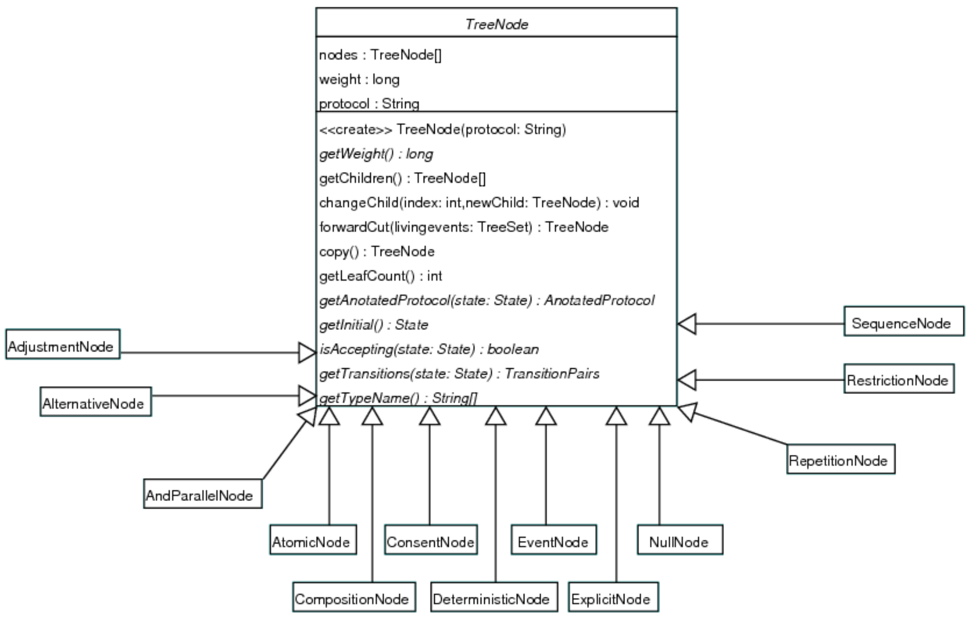
The position within the state space is represented by an instance of the class `State` (see Figure 5.5). In general, for each node of the parse tree, we construct a finite automaton generating the language represented by the corresponding subtree of the parse tree starting in this node. The state of such an automaton consists of the states of the automata of the node's children nodes, enriched with information from this node. The information added in the node depends on the node type. For example, for `AlternativeNode` the piece of information would be the index of the subtree that represents the branch being currently traversed (only one branch is traversed at a time in this case).

Figure 5.5. The State class hierarchy



As mentioned above, there are various types of nodes within the parse tree. The node type corresponds to the operator used in the protocol. The node class hierarchy is depicted on Figure 5.6. The node is responsible for computing the transitions from the state of the automaton corresponding to that node, testing the state for being accepting and providing the initial state of the automaton. The algorithms for computing the transitions from the transitions of subautomaton (or subautomata in some cases) are straightforward in most cases.

Figure 5.6. The TreeNode class hierarchy



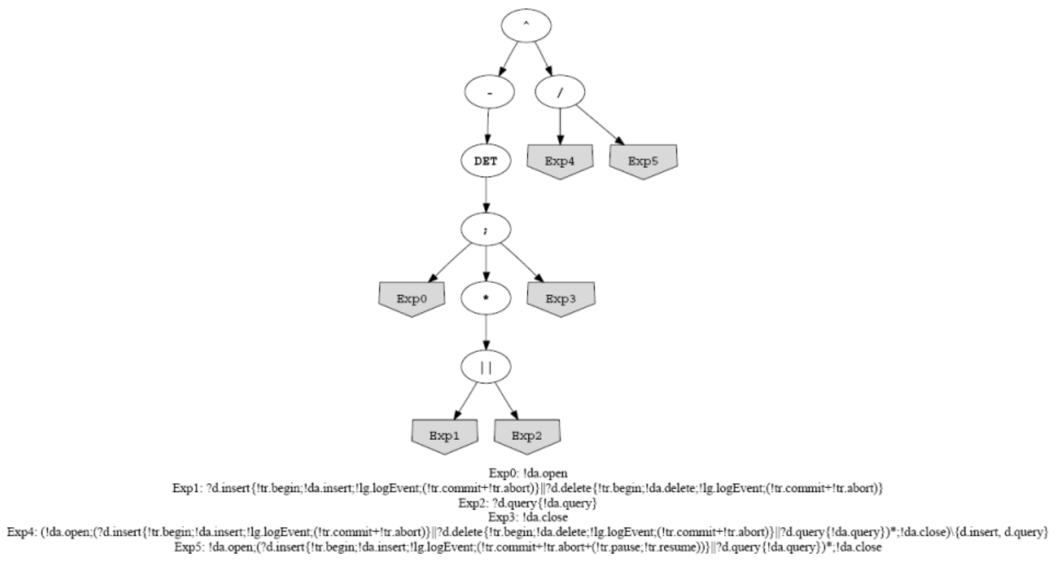
5.1.5. Visualization

The program has also the ability to visualize both the protocol parse trees and their corresponding automata. It generates the source file for the *dot* tool, a part of the *GraphViz* package (available at `http://www.graphviz.org/`). Names of the files containing the description of a parse tree start with the prefix `'pt_'`, and names of files with automata description start with the prefix `'a_'`. The *dot* tool supports a large number of output formats; of these, EPS and VRML seem to be most useful. The files consist of descriptions of nodes and transitions among them, while the placement of the nodes and transitions is left up to the *dot* tool. For example, to obtain the parse tree diagram shown in Figure 5.7, the source file may have the form:

```
digraph G {
  size = "11,7";
  Intersection20 [label="^", fontname="Courier-Bold"];
 Intersection20 -> Complement1;
  Complement1 [label="-", fontname="Courier-Bold"];
 Complement1 -> Deterministic2;
  Deterministic2 [label="DET", fontname="Courier-Bold"];
 Deterministic2 -> Sequence3;
  Sequence3 [label=";", fontname="Courier-Bold"];
 Sequence3 -> Explicit4;
  Explicit4 [label="Exp0", fontname="Courier", shape=invhouse,
             style=filled, fillcolor="grey85"];
 Sequence3 -> Repetition5;
  Repetition5 [label="*", fontname="Courier-Bold"];
 Repetition5 -> OrParallel6;
  OrParallel6 [label="||", fontname="Courier-Bold"];
 OrParallel6 -> Explicit7;
  Explicit7 [label="Exp1", fontname="Courier", shape=invhouse,
             style=filled, fillcolor="grey85"];
 OrParallel6 -> Explicit8;
  Explicit8 [label="Exp2", fontname="Courier", shape=invhouse,
             style=filled, fillcolor="grey85"];
 Sequence3 -> Explicit9;
  Explicit9 [label="Exp3", fontname="Courier", shape=invhouse,
             style=filled, fillcolor="grey85"];
 Intersection20 -> Adjustment10;
  Adjustment10 [label="/", fontname="Courier-Bold"];
 Adjustment10 -> Explicit11;
  Explicit11 [label="Exp4", fontname="Courier", shape=invhouse,
              style=filled, fillcolor="grey85"];
 Adjustment10 -> Explicit12;
  Explicit12 [label="Exp5", fontname="Courier", shape=invhouse,
              style=filled, fillcolor="grey85"];
  label="Exp0: !da.open\nExp1: ?d.insert{!tr.begin;!da........

}
```

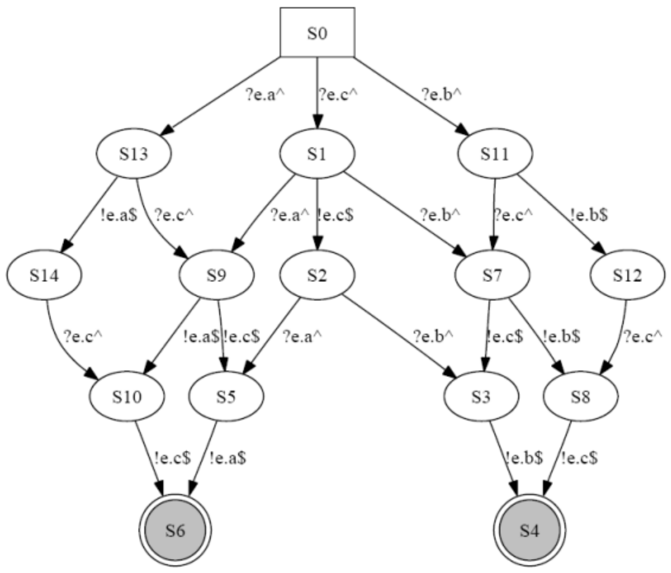
This listing describes the nodes of a parse tree and relations between them; each node has an id (e.g., Sequence3) and a label (e.g., ";"). The transitions join a parent node with its children (e.g., "Sequence3 -> Explicit4", "Sequence3 -> Explicit9"). In the parse tree diagrams, the explicit automata are displayed as gray pentagons and the protocols being represented by such subnodes are displayed at the bottom of the graph. In automaton diagrams, the initial state is displayed as a rectangle and the accepting states are gray circles with double border. Simple examples can be seen at Figure 5.7 and Figure 5.8. Note that the automaton visualization of a more complex protocol may result into a figure having hundreds or thousands of states, which will unfortunately not be of much help. Here, the VRML visualization can be used as an output (note that a VRML browser able to handle complex files and hardware fast enough to view the diagrams are needed in this case).

Figure 5.7. Parse tree visualization



```
Exp0: !da.open
Exp1: ?d.insert{!tr.begin;!da.insert;!lg.logEvent;(!tr.commit+!tr.abort)}||?d.delete{!tr.begin;!da.delete;!lg.logEvent;(!tr.commit+!tr.abort)}
Exp2: ?d.query{!da.query}
Exp3: !da.close
Exp4: (!da.open;(?d.insert{!tr.begin;!da.insert;!lg.logEvent;(!tr.commit+!tr.abort)}||?d.delete{!tr.begin;!da.delete;!lg.logEvent;(!tr.commit+!tr.abort)}||?d.query{!da.query})*;!da.close)\{d.insert, d.query}
Exp5: !da.open;(?d.insert{!tr.begin;!da.insert;!lg.logEvent;(!tr.commit+!tr.abort+(!tr.pause;!tr.resume))}||?d.query{!da.query})*;!da.close
```

Figure 5.8. Automaton visualization



5.1.6. Further information

For detailed information about the classes details see the checker javadoc documentation [javadoc/index.html].

5.2. Implementation of the runtime checker

5.2.1. Overview

The implementation of the runtime checker exploits a lot of functionality of the static checker implementation. The core part of the static checker, i.e., the on-the-fly generation of the transition graph, can be reused without any changes. As the state space is not exhaustively traversed during the runtime check, but the traversal is driven by the information about method calls provided by the component interceptors, only one transition at a point representing the event being performed is taken.

5.2.2. Atomic actions

Atomic actions need to be handled in a special way during the runtime checking. As only one event may be executed in each step and a protocol containing an atomic action thus can't be satisfied at runtime checking, each atomic action is replaced with a protocol consisting of atomic action events combined using the and-parallel operator expressing the necessity that each of the atomic action events has to be executed, but the order doesn't matter. The transformation is done during the protocol parsing process, so it is invisible to the other parts of the system.

5.2.3. Implementation details

The runtime checker class provides two methods: a method for notification about the event being performed and a method testing whether the current state is accepting (i.e., whether the protocol allows the component to finish). The class also remembers the current state (the initial state is set in the constructor) and each time the notify method is called, this state is updated. Should an event not allowed by the protocol occur, the notify method returns bad activity information.

5.3. Cooperation of Java PathFinder with protocol checker

As already said in Section 4.7, the cooperation between JPF and the checker is implemented via (i) a JPF listener that notifies the checker of invoke and return instructions corresponding to method calls on frame interfaces of a checked component, and (ii) an enhanced JPF search engine that differs from the standard search engine in that it asks the checker for a permission to backtrack when JPF comes to an already visited state. In this section, we describe the modifications of JPF and the checker necessary to successfully implement the JPF listener and the enhanced JPF search engine.

The main entry point is the `JPFChecker` class. Its `check` method accepts an instantiated root Fractal component and then for each primitive component in the hierarchy (i) uses the environment generator to generate an environment of the component from its frame protocol, (ii) configures JPF, and (iii) runs JPF with the checker to check whether the component is bounded by its frame protocol.

5.3.1. Checker for code analysis

At the side of the checker, cooperation is implemented by the `JPFTraverser` class that is able to accept notifications from JPF, and by the `JPFCooperatingTraverser` class that extends the `JPFTraverser` with support for coordination of backtracking. The checker for code analysis works in a way similar to the static checker - in each state it generates the list of all possible transitions and moves along one of them. The only difference between the static version and this one is, that via notification of the transition taken in JPF state space, JPF chooses the transition to be taken and tells the checker when to backtrack. The `wantsBacktrack` method of the `JPFStaticChecker` returns true only in the cases when the checker is in an already visited state[1]. Each time JPF gets into an already visited state, it asks the checker for a permission to backtrack. If the checker agrees, both JPF and checker backtrack, otherwise the state space traversal goes on by visiting the JPF-already-visited states again (and visiting unexplored states on the checker's side).

5.3.2. Extensions to the Java PathFinder

The JPF listener is represented by the `ProtocolListener` class which implements the `SearchListener` interface, a part of the JPF API. The checker receives notifications as method calls on its `JPFTraverser` instance - the object is provided by the checker via a call of the `getNotifee`

[1]Note that JPF does not ask for a "permission" to backtrack in an end state, as there is no other way to go on.

method of the `JPFStaticChecker` class. In addition to notifications, the listener also looks for occurrences of end states in the JPF state space. If no end state is visited at all, a short warning is printed at the end, alerting that there is probably an infinite loop in the code.

The extended JPF search engine is implemented by the `JPFCheckerSearch` class that implements the `SearchListener` interface provided by JPF. It is based on the `DFSearch` class, also provided by JPF, that represents the standard search engine based on DFS.

As for changes to the core of JPF, we modified the POR-related code so that a transition is terminated when an invoke or return instruction corresponding to a frame interface method call is executed. We implemented it by making such instructions scheduling relevant. The list of relevant methods of the frame interfaces of a target component is stored as a static attribute of the `JVM` class that is a part of JPF; it is provided to JPF before it is started in the `check` method of the `JPFChecker` class.

Besides the changes to the JPF core and the extensions related to cooperation, we used the MJI abstraction for re-implementation of several classes from the standard `java.lang` and `java.io` packages, because those classes contain some native methods, and the MJI native peers for them are not distributed with JPF. As the functionality provided by those classes is necessary for Fractal (and Julia) to work, we had to re-implement the classes and provide corresponding MJI native peers in order to enable checking of programs that use the Fractal API with Java PathFinder. In particular, we had to extend JPF with support for class loaders and file I/O. Additionally, we also extended Java PathFinder with support for modeling time.

Chapter 6. User's manual

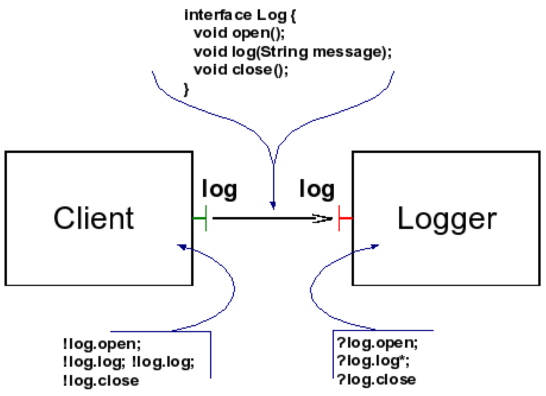
This chapter illustrates typical scenarios of using the protocol checker. The structure of this chapter follows the two main ways a component application is being built (i.e., using Fractal ADL vs. building an application directly) and how a protocol is associated with a component. The last part of the chapter is dedicated to using the protocol checker as a standalone tool independent of the Fractal component model.

6.1. Fractal ADL protocol checking

6.1.1. Getting started

The easiest way of checking the compliance of component frame protocols in Fractal is to augment the Fractal ADL of an existing application to contain frame protocol definition. Recall the example from Section 1.2.1 (shown in Figure 6.1).

Figure 6.1. Example of a component application with behavior protocols



The example consists of two components. The right one implements a logger. The left one implements a client which uses the logger. The protocol of the Logger component prescribes that first the log maintained by the logger has to be opened (by calling the method `void open()`), then arbitrary number of log entries can be written (by calling the method `void log(String message)`). Eventually, using the log is completed by closing the log (calling the method `void close()`).

The introduction of behavior protocols does not change how an application is implemented. Only the Fractal ADL architecture definition (here, the file LoggerDemo.fractal) has to be augmented with the protocol specification as shown below; the lines specificying the behavior are marked with bold font. This file along with a sample implementation of the components can be found in directory `examples/logger`.

```
<definition name="LoggerDemo">
  <interface name="run" role="server"
    signature="java.lang.Runnable"/>

  <component name="client">
    <interface name="log" role="client" signature="logger.Log"/>
    <interface name="run" role="server"
      signature="java.lang.Runnable"/>
    <content class="logger.ClientImpl"/>
    <protocol value="!log.open;!log.log;!log.log;!log.close"/>
  </component>
  <component name="logger">
    <interface name="log" role="server" signature="logger.Log"/>
    <content class="logger.LoggerImpl"/>
    <protocol value="?log.open;?log.log*;?log.close"/>
  </component>

  <binding client="client.log" server="logger.log"/>

  <binding client="this.run" server="client.run"/>
</definition>
```

Once having specified the behavior protocols of the components, the compliance of the components can be checked easily by a special FractalADL launcher:

```
# java org.objectweb.fractal.behprotocols.staticchecker.Launcher \
  -check logger.LoggerDemo

Checking for compliance ... OK
```

The response indicates that the two components in our example have compliant behavior protocols.

6.1.2. Launcher

Checking of protocol compliance is realized by a special Fractal ADL application launcher:
`java org.objectweb.fractal.behprotocols.staticchecker.Launcher` [-check] *`definition`* [*`itf`*]

The argument *definition* is the ADL file containing the definition of the component to be instantiated and started. The argument *itf* is the name of the top-level component's Runnable interface, if it has such. If not given, an interface named `run` is assumed.

By default, the component is run without checking of protocol compliance. The checking can be selected with the *-check* switch. In this case, the component application is only instantiated (without being started) and checked[1]. The results are printed out to the standard output. The meaning of the error reports is explained in Section 6.3.1.

6.1.3. Configuring Julia

The protocol checker uses the runtime representation of components. At runtime, a protocol is associated with a component using the protocol controller that holds the protocol. In order to use this settings, it is necessary to customize the Fractal runtime to attach a protocol controller to newly created components. The way a protocol controller is attached to a component is specific to a particular Fractal implementation. In the case of Julia, this is achieved by modifying the Julia configuration (e.g., the julia.cfg file) in the following way:

```
# Protocol Controller interface
(protocol-controller-itf
  (protocol-controller
      org.objectweb.fractal.behprotocols.ProtocolController)
)

# Protocol Controller implementation
(protocol-controller-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    ProtocolControllerImpl
```

[1]Checking compliance on the tree of instantiated components allows us to uniformly support both applications built from ADL as well as applications built directly from code (as described in Section 6.2).

```
      org.objectweb.fractal.julia.BasicControllerMixin
      org.objectweb.fractal.behprotocols.julia.ProtocolControllerMixin
    ))
  )

  # Protocol Controller added to "primitive" component kind
  (primitive
    (
      'interface-class-generator
      (
        'component-itf
        'binding-controller-itf
        'super-controller-itf
        'lifecycle-controller-itf
        'name-controller-itf
        'protocol-controller-itf
      )
      (
        'component-impl
        'container-binding-controller-impl
        'super-controller-impl
        'lifecycle-controller-impl
        'name-controller-impl
        'protocol-controller-impl
      )
      (
        (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
          org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
        )
      )
      org.objectweb.fractal.julia.asm.MergeClassGenerator
      'optimizationLevel
    )
  )
```

6.2. Building application directly from code

6.2.1. Associating a protocol with a component instance

When an application is built directly from Java-code, protocols are assigned to components by calling the method `setFcProtocol` on the protocol controller associated with a component. This is illustrated in the following code snippet which shows how a protocol is set for the Logger component. Protocols of the other components in the application are assigned in the same way (not shown in the example).

```
  ...

  Component boot=Fractal.getBootstrapComponent();
  GenericFactory cf=Fractal.getGenericFactory(boot);
  ...

  Component logger=cf.newFcInstance(loggerType, "primitive", ...);

  ProtocolController loggerProtoCont =
   (ProtocolController) logger.getFcInterface("protocol-controller");

  loggerProtoCont.setFcProtocol("?log.open;?log.log*;?log.close");
```

```
...
```

6.2.2. Checking instantiated components

The checking of protocol compliance is performed on an instantiated component application before it is started (using its lifecycle-controller). The checker is invoked by calling static method `check` on the class `org.objectweb.fractal.behprotocols.staticchecker.ProtocolChecker`. The checker goes through a given component nesting hierarchy, and on each level of nesting, it checks the compliance of a component's frame protocol with the architecture protocol constructed from the frame protocols of the component's direct sub-components. The parameter passed to the checker is the root component of the nesting hierarchy that is to be verified. Thus, in the example bellow, we pass to the checker the application's top-level component.

```
...

System.out.print("Checking for compliance ...");
NestedCheckingResult res = ProtocolChecker.check(rootComp);

if (res.getErrorType() != NestedCheckingResult.ERR_OK) {
  System.out.println(" Error:");
  System.out.println(res.toString());
} else {
  System.out.println(" OK");
}

...
```

The meaning of the output is described in detail in 5.3.

6.2.3. Configuring Julia

Again, as in the case of instantiating components using Fractal ADL, it is necessary to customize the Fractal implementation used, in order to attach a protocol controller to newly created components. The way a protocol controller is attached to a component is specific to a particular Fractal implementation. In the case of Julia, this is achieved by modifying the Julia configuration (e.g., julia.cfg) as described in 5.1.3.

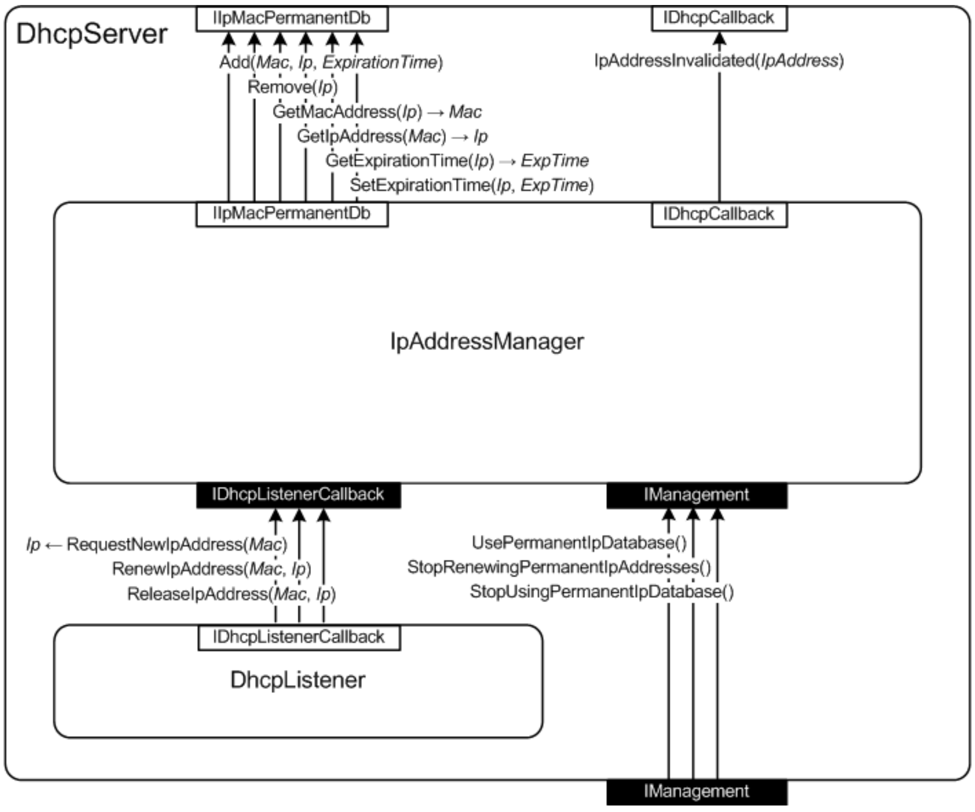
6.3. Protocol checker user manual for the standalone version

6.3.1. Getting Started

6.3.1.1. A sample component design

The following picture shows an example of a composite component DhcpServer. The DhcpServer component contains two inner components - the IpAddressManager and the DhcpListener. The DhcpServer works as a component implementation of a DHCP server, where the DhcpListener communicates with the network clients via the DHCP protocol and the IpAddressManager component is responsible for managing the IP addresses assigned to them. The key functionality (besides being a DHCP server for the local network) is to notify other components of disconnected clients (after a client's IP address is released) via the IDhcpCallback interface. The whole DhcpServer component can be managed via the IManagement interface and optionally can use an external database of IP address/MAC address mappings (the database is accesses via the IIpMacPermanentDb interface). Usage of an external database is enabled by the UsePermanentIpDatabase method of the IManagement interface.

Figure 6.2. DhcpServer composite component



The picture shows both provided (server) interfaces (shown as solid black rectangles) and required (client) interfaces (shown as solid white rectangles) of the components and also bindings between them. The bindings are shown as arrows going in the direction of method calls. Each arrow in the picture actually represents a method call or method binding, so only the whole bunch of such arrows leading from one interface to another represents the binding between the interfaces. It can be deduced from the picture that the inner IpAddressManager component is bound the outer interfaces IManagement (provided interface type), IIpMacPermanentDb (required interface type - this interface does not need to be bound) and IDhcpCallback (required interface type) of the DhcpServer composite component. The DhcpListener primitive component is bound to the IpAddressManager component via the IDhcpListenerCallback interface.

This example is a simplified version of the DhcpServer composite component from the Demo. The components that are "missing" in this simple example are in fact considered to be inside the IpAddressManager component presented here. At this level of abstraction, the IpAddressManager component can be simply viewed as a black box with its provided and required interfaces and defined behavior, and one does not need to worry about the real composite nature of the component implementation.

6.3.1.2. Writing protocols

First we want to check the compliance of the two inner components IpAddressManager and DhcpServer together. The behavior protocols of these two components follow:

Figure 6.3. IpAddressManager Behavior Protocol

```
(
  (
    (
      ?IDhcpListenerCallback.RequestNewIpAddress
      +
      ?IDhcpListenerCallback.RenewIpAddress
      +
      ?IDhcpListenerCallback.ReleaseIpAddress {
        (!IDhcpCallback.IpAddressInvalidated + NULL)
      }
    )*
    |
    (
      (!IDhcpCallback.IpAddressInvalidated + NULL)
    )*
  )
  +
  (
    (
      (
        (
          ?IDhcpListenerCallback.RequestNewIpAddress
          +
          ?IDhcpListenerCallback.RenewIpAddress
          +
          ?IDhcpListenerCallback.ReleaseIpAddress {
            (!IDhcpCallback.IpAddressInvalidated + NULL)
          }
        )*
        |
        (
          (!IDhcpCallback.IpAddressInvalidated + NULL)
        )*
      )
      |
      ?IManagement.UsePermanentIpDatabase^
    ) ; !IManagement.UsePermanentIpDatabase$ ; (
      (
         (
          ?IDhcpListenerCallback.RequestNewIpAddress {
            !IIpMacPermanentDb.GetIpAddress
          }
          +
          ?IDhcpListenerCallback.RenewIpAddress
          +
          ?IDhcpListenerCallback.ReleaseIpAddress {
            (!IDhcpCallback.IpAddressInvalidated + NULL)
          }
        )*
        |
        (
          (!IDhcpCallback.IpAddressInvalidated + NULL)
        )*
       )
      |
      ?IManagement.StopUsingPermanentIpDatabase^
    ) ; !IManagement.StopUsingPermanentIpDatabase$
  )*
)
```

Figure 6.4. DhcpListener Behavior Protocol

```
(
  !IDhcpListenerCallback.RequestNewIpAddress
  +
  !IDhcpListenerCallback.RenewIpAddress
  +
  !IDhcpListenerCallback.ReleaseIpAddress
)*
```

To check the compliance of these two protocols, we need to prepare an input file for the static behavior checker. The file will contain both behavior protocols and a specification of actions (method calls) via which the IpAddressManager (first protocol) and the DhcpListener (second protocol) components are bound together. Because the components are bound together only by the IDhcpListenerCallback interface, the RequestNewIpAddress, RenewIpAddress and ReleaseIpAddress methods from the IDhcpListenerCallback will be the only actions written in the behavior checker input file. In this step we don't want to check for incomplete bindings, so the unbound operations in the input file will be empty. The corresponding input file for the behavior checker may take the form:

Figure 6.5. Static behavior checker input file

```
#DhcpListener
(
  !IDhcpListenerCallback.RequestNewIpAddress
  +
  !IDhcpListenerCallback.RenewIpAddress
  +
  !IDhcpListenerCallback.ReleaseIpAddress
)*
#eop

      #synchro ops
      IDhcpListenerCallback.RequestNewIpAddress,
      IDhcpListenerCallback.RenewIpAddress,
      IDhcpListenerCallback.ReleaseIpAddress
      #eop

#IpAddressManager
(
  (
    (
      ?IDhcpListenerCallback.RequestNewIpAddress
      +
      ?IDhcpListenerCallback.RenewIpAddress
      +
      ?IDhcpListenerCallback.ReleaseIpAddress {
        (!IDhcpCallback.IpAddressInvalidated + NULL)
      }
    )*
    |
    (
      (!IDhcpCallback.IpAddressInvalidated + NULL)
    )*
  )
  +
  (
    (
      (
        (
          ?IDhcpListenerCallback.RequestNewIpAddress
          +
          ?IDhcpListenerCallback.RenewIpAddress
          +
          ?IDhcpListenerCallback.ReleaseIpAddress {
            (!IDhcpCallback.IpAddressInvalidated + NULL)
          }
        )*
        |
        (
          (!IDhcpCallback.IpAddressInvalidated + NULL)
        )*
      )
      |
      ?IManagement.UsePermanentIpDatabase^
    ) ; !IManagement.UsePermanentIpDatabase$ ; (
      (
         (
          ?IDhcpListenerCallback.RequestNewIpAddress {
            !IIpMacPermanentDb.GetIpAddress
          }
          +
          ?IDhcpListenerCallback.RenewIpAddress
          +
          ?IDhcpListenerCallback.ReleaseIpAddress {
            (!IDhcpCallback.IpAddressInvalidated + NULL)
          }
        )*
        |
        (
          (!IDhcpCallback.IpAddressInvalidated + NULL)
        )*
       )
      |
      ?IManagement.StopUsingPermanentIpDatabase^
    ) ; !IManagement.StopUsingPermanentIpDatabase$
  )*
)
#eop

#unbound ops
#none
#eop
```

Lines beginning with the "#" sign are just comments. The only exception is "#eop", which means "End Of a Protocol". This token serves as a delimiter and allows an input file to use advanced protocol formating to improve readability. The file contains three types of sections. Each section is separated by the "#eop" delimiters (or by the start or the end of a file). The first section type contains frame protocols. The first protocol is a frame protocol of the architecture while the other protocols describe the subcomponents' behavior. Between each two protocol sections there is a synchro-operation section. This section type contains synchro-operations that represent all methods of the interfaces bound between the two components). The third section type is the last section. This section enumerates all methods of unbound interfaces, i.e., operations that should not be performed (if it is a provides interface) or must not be performed (if it is a requires interface, otherwise causing an unbound-requires-called error).

6.3.1.3. Checking for compliance

After the protocols are completed, their compliance check can be performed. For the example above, the command line would be:

java -jar checker.jar --action=testconsent -f architecture.bp

The output of this command should be:

```
OK
```

If the user wants more detailed information, the static behavior checker can be run with the verbose option:

java -jar checker.jar --action=testconsent --verbose=1 -f architecture.bp

The output of the checker is then a bit more complex, as can be seen in the following listing. The important parts of the output, i.e., the number of states visited and the result of the protocol checking, are highlighted in bold:

```
Protocol Build: start
(!IDhcpListenerCallback.RequestNewIpAddress+!IDhcpListenerCallback.
RenewIpAddress+!IDhcpListenerCallback.ReleaseIpAddress)*
Protocol Build: finished
Synchroops: IDhcpListenerCallback.RequestNewIpAddress,IDhcpListener
Callback.RenewIpAddress,IDhcpListenerCallback.ReleaseIpAddress
Protocol Build: start
(((?IDhcpListenerCallback.RequestNewIpAddress+?IDhcpListenerCallbac
k.RenewIpAddress+?IDhcpListenerCallback.ReleaseIpAddress{(!IDhcpCal
lback.IpAddressInvalidated+NULL)})*|((!IDhcpCallback.IpAddressInval
idated+NULL))*)+((((?IDhcpListenerCallback.RequestNewIpAddress+?IDh
cpListenerCallback.RenewIpAddress+?IDhcpListenerCallback.ReleaseIpA
ddress{(!IDhcpCallback.IpAddressInvalidated+NULL)})*|((!IDhcpCallba
ck.IpAddressInvalidated+NULL))*)|?IManagement.UsePermanentIpDatabas
e^);!IManagement.UsePermanentIpDatabase$;(((?IDhcpListenerCallback.
RequestNewIpAddress{!IIpMacPermanentDb.GetIpAddress}+?IDhcpListener
Callback.RenewIpAddress+?IDhcpListenerCallback.ReleaseIpAddress{(!I
DhcpCallback.IpAddressInvalidated+NULL)})*|((!IDhcpCallback.IpAddre
ssInvalidated+NULL))*)|?IManagement.StopUsingPermanentIpDatabase^);
!IManagement.StopUsingPermanentIpDatabase$)*)
Protocol Build: finished
Synchroops:
Protocol Build: start

Protocol Build: finished
State space estimate: 824
Optimizing the parse tree for the composition test......done.
Cache created with capacity of 2112000 items.
Cache created with capacity of 2147483647 items.
1024
Stack size:51
1024
Stack size:51
1024
Stack size:51
1024
Stack size:51
1024
Stack size:51
1979 states visited.
Protocols are composition error free.
OK
Taken 1 seconds.
```

Now the user can create behavior protocol describing the composite component DhcpServer:

Figure 6.6. DhcpServer composite component behavior protocol

```
(
  !IDhcpCallback.IpAddressInvalidated*
  |
  (
    ?IManagement.UsePermanentIpDatabase^ ; (
      !IIpMacPermanentDb.GetIpAddress*
      |
      (
        !IManagement.UsePermanentIpDatabase$ ;
        ?IManagement.StopUsingPermanentIpDatabase^
      )
    ) ; !IManagement.StopUsingPermanentIpDatabase$
  )*
)
```

For testing compliance of the frame and architecture protocols, the following input file is created:

Figure 6.7. Input file for the behavior protocol checker

```
#DhcpServer frame protocol
(
    !IDhcpCallback.IpAddressInvalidated*
    |
    (
      ?IManagement.UsePermanentIpDatabase^ ; (
        !IIpMacPermanentDb.GetIpAddress*
        |
        (
          !IManagement.UsePermanentIpDatabase$ ;
          ?IManagement.StopUsingPermanentIpDatabase^
        )
      ) ; !IManagement.StopUsingPermanentIpDatabase$
    )*
)
#eop

      #synchro ops between frame and architecture protocols
      IManagement.UsePermanentIpDatabase,
      IManagement.StopUsingPermanentIpDatabase,
      IIpMacPermanentDb.GetIpAddress,
      IDhcpCallback.IpAddressInvalidated
      #eop

#DhcpListener
(
  !IDhcpListenerCallback.RequestNewIpAddress
  +
  !IDhcpListenerCallback.RenewIpAddress
  +
  !IDhcpListenerCallback.ReleaseIpAddress
)*
#eop

      #synchro ops
      IDhcpListenerCallback.RequestNewIpAddress,
      IDhcpListenerCallback.RenewIpAddress,
      IDhcpListenerCallback.ReleaseIpAddress
      #eop

#IpAddressManager
(
  (
    (
      ?IDhcpListenerCallback.RequestNewIpAddress
      +
      ?IDhcpListenerCallback.RenewIpAddress
      +
      ?IDhcpListenerCallback.ReleaseIpAddress {
        (!IDhcpCallback.IpAddressInvalidated + NULL)
      }
    )*
    |
    (
      (!IDhcpCallback.IpAddressInvalidated + NULL)
    )*
  )
  +
  (
    (
      (
        (
          ?IDhcpListenerCallback.RequestNewIpAddress
          +
          ?IDhcpListenerCallback.RenewIpAddress
          +
          ?IDhcpListenerCallback.ReleaseIpAddress {
            (!IDhcpCallback.IpAddressInvalidated + NULL)
          }
        )*
        |
        (
          (!IDhcpCallback.IpAddressInvalidated + NULL)
        )*
      )
      |
      ?IManagement.UsePermanentIpDatabase^
    ) ; !IManagement.UsePermanentIpDatabase$ ; (
      (
         (
          ?IDhcpListenerCallback.RequestNewIpAddress {
            !IIpMacPermanentDb.GetIpAddress
          }
          +
          ?IDhcpListenerCallback.RenewIpAddress
          +
          ?IDhcpListenerCallback.ReleaseIpAddress {
            (!IDhcpCallback.IpAddressInvalidated + NULL)
          }
        )*
        |
        (
          (!IDhcpCallback.IpAddressInvalidated + NULL)
        )*
       )
      |
      ?IManagement.StopUsingPermanentIpDatabase^
    ) ; !IManagement.StopUsingPermanentIpDatabase$
  )*
)
#eop

#unbound ops
#none
#eop
```

Checker can be run using following command:

java -jar checker.jar --action=test -f compliance.bp

The checker would report a bad activity error in this case:

```
Composition error detected -
   bad activity (!IDhcpCallback.IpAddressInvalidated^):
(S0) #IDhcpListenerCallback.RequestNewIpAddress^
(S5) #IDhcpListenerCallback.RequestNewIpAddress$
(S6) #IManagement.UsePermanentIpDatabase^
(S7) #IDhcpCallback.IpAddressInvalidated^
(S14) #IDhcpListenerCallback.ReleaseIpAddress^
(S15)
```

To find out the reason of the bad activity, we need to analyze the input protocols and the error trace output. In the following listing, the lines in bold highlight the error trace in the protocol (note that some actions are decomposed into separate request and response events - e.g., !m -> !m^; !m$). The event causing the bad activity is emphasized in italics:

Figure 6.8. Input file with highlighted error trace

```
#DhcpServer frame protocol
(
    (!IDhcpCallback.IpAddressInvalidated^;
     ?IDhcpCallback.IpAddressInvalidated$)*
    |
    (
      ?IManagement.UsePermanentIpDatabase^ ; (
        !IIpMacPermanentDb.GetIpAddress*
        |
        (
          !IManagement.UsePermanentIpDatabase$ ;
          ?IManagement.StopUsingPermanentIpDatabase^
        )
      ) ; !IManagement.StopUsingPermanentIpDatabase$
    )*
)
#eop

      #synchro ops between frame and architecture protocols
      IManagement.UsePermanentIpDatabase,
      IManagement.StopUsingPermanentIpDatabase,
      IIpMacPermanentDb.GetIpAddress,
      IDhcpCallback.IpAddressInvalidated
      #eop

#DhcpListener
(
  (!IDhcpListenerCallback.RequestNewIpAddress^;
  ?IDhcpListenerCallback.RequestNewIpAddress$)
  +
  !IDhcpListenerCallback.RenewIpAddress
  +
  !IDhcpListenerCallback.ReleaseIpAddress
)*
#eop

      #synchro ops
      IDhcpListenerCallback.RequestNewIpAddress,
      IDhcpListenerCallback.RenewIpAddress,
      IDhcpListenerCallback.ReleaseIpAddress
      #eop

#IpAddressManager
(
  (
    (
      ?IDhcpListenerCallback.RequestNewIpAddress
      +
      ?IDhcpListenerCallback.RenewIpAddress
      +
      ?IDhcpListenerCallback.ReleaseIpAddress {
        (!IDhcpCallback.IpAddressInvalidated + NULL)
      }
    )*
    |
    (
      (!IDhcpCallback.IpAddressInvalidated + NULL)
    )*
  )
  +
  (
    (
      (
        (
          (?IDhcpListenerCallback.RequestNewIpAddress^;
          !IDhcpListenerCallback.RequestNewIpAddress$)
          +
          ?IDhcpListenerCallback.RenewIpAddress
          +
          ?IDhcpListenerCallback.ReleaseIpAddress {
            (!IDhcpCallback.IpAddressInvalidated + NULL)
          }
        )*
        |
        (
          ( (?IDhcpCallback.IpAddressInvalidated^;
             !IDhcpCallback.IpAddressInvalidated$) + NULL)
        )*
      )
      |
      ?IManagement.UsePermanentIpDatabase^
    ) ; !IManagement.UsePermanentIpDatabase$ ; (
      (
         (
          ?IDhcpListenerCallback.RequestNewIpAddress {
            !IIpMacPermanentDb.GetIpAddress
          }
          +
          ?IDhcpListenerCallback.RenewIpAddress
          +
          ?IDhcpListenerCallback.ReleaseIpAddress {
            (!IDhcpCallback.IpAddressInvalidated + NULL)
          }
        )*
        |
        (
          (!IDhcpCallback.IpAddressInvalidated + NULL)
        )*
       )
      |
      ?IManagement.StopUsingPermanentIpDatabase^
    ) ; !IManagement.StopUsingPermanentIpDatabase$
  )*
)
#eop

#unbound ops
#none
#eop
```

After the protocol analysis, it can be seen that the event !IDhcpCallback.IpAddressInvalidated^ emitted from the IpAddressManager component is already accepted by the only instance of its complementary event ?IDhcpCallback.IpAddressInvalidated^ in the DhcpServer frame protocol. So another !IDhcp-

Callback.IpAddressInvalidated^ event occurring inside of the ?IDhcpListenerCallback.ReleaseIpAddress call cannot be accepted by the DhcpServer component. From this, it can be deduced that the frame protocol of the DhcpServer component needs to be extended to accept two calls of IDhcpCallback.IpAddressInvalidated^ in parallel:

Figure 6.9. DhcpServer composite component behavior protocol

```
(
  !IDhcpCallback.IpAddressInvalidated*
  |
  !IDhcpCallback.IpAddressInvalidated*
  |
  (
    ?IManagement.UsePermanentIpDatabase^ ; (
      !IIpMacPermanentDb.GetIpAddress*
      |
      (
        !IManagement.UsePermanentIpDatabase$ ;
        ?IManagement.StopUsingPermanentIpDatabase^
      )
    ) ; !IManagement.StopUsingPermanentIpDatabase$
  )*
)
```

Now the compliance check can be rerun and after several seconds of computation the user obtains a positive result.

In the case when a composition error is found, the checker can be run with visualization option to obtain both the parse tree graph and state space transition graph:

java -jar checker.jar --action=visualizedot -f compliance.bp

The output of the visualization process is the *dot* format (see Section 6.3.3 for details). Note that visualization only makes sense when the order of magnitude of the state space size does not exceed hundreds, as larger state spaces are not handled properly by the dot tool.

6.3.1.4. Checking for incomplete bindings

If we consider the example from the Section 6.3.1.2 and Section 6.3.1.1, we know that IIpMacPermanentDb requires interface should be an optional interface and therefore does not need to be bound to another interface. We denote the methods of this interface as unbound:

```
...
#unbound ops
IIpMacPermanentDb.GetIpAddress
#eop
```

After running the checker we obtain the error trace:

```
Composition error detected - missing binding
for request '!IIpMacPermanentDb.GetIpAddress^':
(S0) #IDhcpListenerCallback.ReleaseIpAddress^
(S1) #IManagement.UsePermanentIpDatabase^
(S8) #IDhcpListenerCallback.ReleaseIpAddress$
(S9) #IDhcpCallback.IpAddressInvalidated^
(S16) #IDhcpListenerCallback.RenewIpAddress^
```

```
(S17) #IDhcpCallback.IpAddressInvalidated$
(S22) #IDhcpCallback.IpAddressInvalidated^
(S23) #IDhcpListenerCallback.RenewIpAddress$
(S24) #IDhcpCallback.IpAddressInvalidated$
(S33) #IManagement.UsePermanentIpDatabase$
(S35) #IDhcpListenerCallback.RequestNewIpAddress^
(S40)
```

From the error trace it can be seen that the error is caused by the IIpMacPermanentDb.GetIpAddress method called on an unbound interface. The problem arises because as the provided IManagement interface is bound, the highlighted #IManagement.UsePermanentIpDatabase$ event can be performed to enable the use of the external IP/MAC address database. This implies that the IIpMacPermanentDb required interface can be unbound only if the IManagement provided interface is also unbound:

```
...
#unbound ops
IManagement.StopUsingPermanentIpDatabase,
IManagement.UsePermanentIpDatabase,
IIpMacPermanentDb.GetIpAddress
#eop
```

Adding methods of IManagement interface into the list of unbound operations will result into a successful compliance check.

6.3.2. Command line Reference

The checker can be used as a part of the SOFA and Fractal environments or as a standalone tool. This section focuses on the latter case. The syntax of the command line when using the checker as a standalone tool is the following:

java -jar checker.jar (--action|-a)=<test|testconsent|visualizedot> [(--verbose|-v)=<0|1|2>] [--nomultinodes|-m] [--noexplicit|-e] [--noforwardcut|-f] [--nobadactivity|-b] [--nonoactivity|-n] [(--ifiniteactivity|-i)=<no|notrace|yes>] [(--totalmem|-t)=<size>] [(--cachesize|-s)=<size>] <((--file|-f)=<inputfile>) | (<protocol1> <provisions_1_2> <protocol2> [<provisions_2_3> <protocol3>, ...] <unbound_operations)>

The parameters have the following meaning:

[--action|-a]=test - the checker will perform the test for composition errors and consensual compliance of an inverted frame protocol **(<protocol1>)** and the composition of the other protocols (the protocols are composed from back to front - i.e., **protocol_n** and **protocol_n-1** are composed first. In the case the protocols are not consensually compliant or a composition error is detected, a counterexample (i.e., the trace that cannot performed by both of the two components behaving according to the first and the second protocol respectively) is provided and an html file denoting the actual position within the protocols is created.

[--action|-a]=testconsent - the checker will perform the composition test only using the consent operator - i.e., the first protocol is not inverted in this case, but it is composed together with the other protocols. Again, the protocols are composed in a backward order.

[--action|-a]=visualizedot - the checker will create the files for the *dot* visualization tool, representing the parse trees and the automaton that would be used in the compliance check. In this case, the first protocol is supposed to be the frame protocol, and is therefore inverted before visualization.

[--verbose|-v]=level - there are 3 levels of verbosity: 0 (not verbose, default), 1 (normal verbose mode), 2 (extremely verbose mode - useful for debugging). The program will, according to given verbose level, print out information about each step of the compliance test or visualization process, and at the end also prints out the length of the test in milliseconds.

[--nomultinodes|-m] - disables the multinode optimization performed while parsing the input protocols. Option can be useful for benchmarking reasons.

[--noexplicit|-e] - disables the conversion of subautomata to the very fast explicit automata. Option can be useful for benchmarking purposes.

[--noforwardcut|-f] - disables the forward cutting optimization which eliminates those transitions in the resulting automaton, that would be discarded by the use of restriction operator. Option can be useful for benchmarking purposes. Note that when using the consent operator for finding composition errors, forward cutting is not applied, since the consensual compliance is not based on the subset relation.

[--totalmem|-t]=size - specifies in MB the size of memory available for the checker data structures. Note that the checker needs additional 10 - 15 MB of memory for the code and the basic data structure (that cannot be optimized) and that the java virtual machine may allocate more memory if available. To restrict the memory allocated, use the -Xmx option of java virtual machine. The default value is 60.

[--cachesize|-s]=size - specifies in MB the size of memory dedicated for the state cache. This number cannot be higher than the total memory amount. The default value is 48.

[--file|-f]=inputfile - reads the protocols from the file specified. Each protocol and set of synchro operations has to be ended by an **#eop** token at the beginning of a new line. The protocols can be formated using standard whitespace character for better readability.

protocol<n> - n-th protocol (the first is a frame protocol)

provisions<n> - synchronization operations for n-th and (n+1)-th protocols

unbound_operations - all operation of unbound interfaces (used for detection of incomplete bindings)

6.3.3. Visualization

As mentioned above, the checker is able to create a visualization of the parse trees and the automata that are used for compliance checking. The program produces input files for the *dot* visualization tool that is a part of the *GraphViz* package (see http://www.research.att.com/sw/tools/graphviz). The *dot* tool supports multiple format output (for example EPS, PNG or VRML) and is thus very flexible.

If the switch **--action=visualizedot** is given to the checker, the program creates several files - those with names starting with *'pt_'* contains parse trees description and those with names starting with *'a_'* contain automata transition graph.

For simplicity of the graphs, the nodes of a parse tree are not labeled with the operator names, but with the following symbols instead:

`/`	adjustment operator
`+`	alternative operator
`|`	and parallel operator
`-`	complement operator
`&`	composition operator
`%`	consent operator
`DET`	determinization operator
`EXPx`	explicit automaton is used here, corresponding protocol is printed below the parse tree
`^`	intersection operator
`||`	or parallel operator
`*`	repetition operator

`\`	restriction operator
`;`	sequence operator

The subtrees of the parse tree that are replaced in the consequence of *Explicit Automata Optimization* with explicit automata for the compliance test are shown as gray pentagons and the corresponding protocol (i.e., the expression generating the language being accepted by this automaton) is printed out below the parse tree as the graph legend.

6.3.4. Example of protocol input file

```
# Hash sign '#' denotes comments
#
# As the first protocol is a frame protocol,
# the checker should be launched with the option --action=test
#

#the frame protocol
(?aIN.m || ?bIN.m)*
#eop

# events which the components communicate through
  aIN.m,
  bIN.m
#eop

#another protocol
(
  (?aIN.m {
    (?b.s {!aOUT.m}; ?b.u)
    +
    (!a.s;!aOUT.m;!a.u)
  })
  +
  (?b.s {?aIN.m^}; ?b.u^; !aOUT.m; !aIN.m$; !b.u$)
  +
  (?b.s; ?aIN.m {?b.u^; !aOUT.m}; !b.u$)
  +
  (?b.s; ?b.u)
)*
#eop

#events
  aOUT.m,
  a.s,
  a.u,
  b.s,
  b.u
#eop

#another protocol
(
  (?bIN.m {
    (?a.s {!bOUT.m}; ?a.u)
    +
    (!b.s; !bOUT.m; !b.u)
  })
  +
  (?a.s {?bIN.m^}; ?a.u; (!b.s; !bOUT.m; !b.u))
  +
  (?a.s; ?a.u)
)*
#eop

#events
  bOUT.m
#eop

#another protocol
(?aOUT.m + ?bOUT.m)*
#eop

#there are no unbound operations, close the empty list with #eop
#eop
```

Note that you should get a bad activity error when checking these protocols.

6.4. Fractal extensions: Run-time checker

6.4.1. Getting started

Runtime checking is integrated into Fractal by the means of the *runtime-check controller*. The runtime-check controller closely cooperates with the protocol controller and with runtime-check interceptors, which notify the controller as method requests and responses pass through the component membrane. The runtime-check controller is responsible for initializing the interceptors, obtaining the protocol set in the protocol controller, and creating an instance of the runtime-checker backend implementation.

The easiest way to apply runtime-checking to an application is to start the application augmented with behavior protocol specifications in the same way as described in Section 6.1.1; the only additional step required is to activate the runtime-check controller framework, which is achieved by modifying the controller descriptors used for primitive and composite components (please see Section 6.4.2).

We demonstrate this on a modified version of the Fractal ADL HelloWorld example, augmented with simple behavior protocols. This example is available in the `examples/helloworldadl` directory. The example can be started via the protocol-aware Fractal ADL launcher, `org.objectweb.fractal.behprotocols.staticchecker.Launcher`.

The Ant build-file provided with this example allows us to start the demo simply by issuing the command:

```
ant execute
```

The program output includes the (in this configuration) verbose output from the runtime-checking framework, which reports on the protocols applied to the components and the results of the checking. Configuring the behavior and the level of output of the runtime-check framework will be discussed in detail in Section 6.4.3.

6.4.2. Julia configuration

To activate the runtime-checking framework, is is necessary to extend the controller descriptor of both primitive and composite components with the definition of the runtime check controller (implemented by the `BasicRuntimeCheckControllerMixin` class.). The modified controller descriptor definitions are provided in the file `julia-rtcheck.cfg`. Besides introducing the new controller, this file also extends the definition of the lifecycle controller. This additional functionality, implemented in the `RuntimeCheckLifeCycleMixin` class, initializes the runtime check controller at the time the component starts and notifies the runtime check controller when the component stops. To properly handle identity-aware interceptors, the configuration file also introduces a new base-class for the interceptor objects, `BasicIdentityAwareComponentInterface`. The key elements of the configuration file are shown in Figure 6.10.

Figure 6.10. Controller descriptor extensions defined in `julia-rtcheck.cfg`

```
(interface-class-generator
  (org.objectweb.fractal.julia.asm.InterfaceClassGenerator
    org.objectweb.fractal.behprotocols.julia. \
      BasicIdentityAwareComponentInterface
  )
)

# Runtimecheck Controller

(runtimecheck-controller-itf
  (runtimecheck-controller org.objectweb.fractal.behprotocols. \
     RuntimeCheckController)
)

# Runtimecheck Controller implementation

(runtimecheck-controller-impl
 ((org.objectweb.fractal.julia.asm.MixinClassGenerator
   RuntimecheckControllerImpl
   org.objectweb.fractal.julia.BasicControllerMixin
   org.objectweb.fractal.julia.control.name. \
     UseNameControllerMixin
   org.objectweb.fractal.behprotocols.julia. \
     BasicRuntimeCheckControllerMixin
 ))
)

(lifecycle-controller-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    LifeCycleControllerImpl
    org.objectweb.fractal.julia.BasicControllerMixin
    org.objectweb.fractal.julia.UseComponentMixin
    org.objectweb.fractal.julia.control.lifecycle. \
      BasicLifeCycleCoordinatorMixin
    org.objectweb.fractal.julia.control.lifecycle. \
      BasicLifeCycleControllerMixin
    # to check that mandatory client interfaces are bound in startFc:
    org.objectweb.fractal.julia.control.lifecycle.TypeLifeCycleMixin
    # to notify the encapsulated component
    # (if present) when its state changes:
    org.objectweb.fractal.julia.control.lifecycle.
      ContainerLifeCycleMixin
    ##### extensions for runtimecheck controller interaction
    # require a reference to ProtocolController
    org.objectweb.fractal.behprotocols.julia.
      UseProtocolControllerMixin
    # require a reference to RuntimeCheckController
    org.objectweb.fractal.behprotocols.julia.
      UseRuntimeCheckControllerMixin
    # do the interaction
    org.objectweb.fractal.behprotocols.julia.
      RuntimeCheckLifeCycleMixin
  ))
)
```

```
(primitive
  (
    'interface-class-generator
    (
      'component-itf
      'binding-controller-itf
      'super-controller-itf
      'lifecycle-controller-itf
      'name-controller-itf
      'protocol-controller-itf
      'runtimecheck-controller-itf
    )
    (
      'component-impl
      'container-binding-controller-impl
      'super-controller-impl
      'lifecycle-controller-impl
      'name-controller-impl
      'protocol-controller-impl
      'runtimecheck-controller-impl
    )
    (
      (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
        org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
        org.objectweb.fractal.behprotocols.julia. \
   RuntimeCheckInterceptorCodeGenerator
      )
    )
    org.objectweb.fractal.julia.asm.MergeClassGenerator
    'optimizationLevel
  )
)
```

6.4.3. Running a runtime-checked application

To make a Fractal application subject to runtime checking, the only step to be taken is to include the customized Julia configuration file in the list of configuration files to be processed. The command below shows how the HelloWorld example is launched from Ant (except for setting the classpath). The most significant difference is the additional configuration file added to the `julia.config` system property; in addition, the verbosity of the runtime-check framework is increased for demonstration purposes.

```
java -Dfractal.provider=org.objectweb.fractal.julia.Julia \
  -Djulia.loader=org.objectweb.fractal.julia.loader.DynamicLoader \
  -Djulia.config=etc/julia.cfg,etc/julia-rtcheck.cfg \
  -Dfractal.protocols.rtcheck.verbosity=2 \
  org.objectweb.fractal.behprotocols.adl.Launcher \
  WrappedHelloWorld r
```

The runtime-check framework can be configured via properties; the properties are `fractal.protocols.rtcheck.recordtrace`, `fractal.protocols.rtcheck.stoponerror`, `fractal.protocols.rtcheck.throwerrors`, `fractal.protocols.rtcheck.recorderrors`, and `fractal.protocols.rtcheck.verbosity`. The properties control whether the runtimecheck framework records the complete trace of the component execution, what behavior is desired when an error occurs (throw a `ProtocolViolationException`, log the error and continue execution), whether the component should log all errors encountered, and what level of verbosity is desired. The properties and their default values are described in detail in Section 4.4.4.

We demonstrate the runtime-check framework on an example based on the Fractal ADL HelloWorld demo. We have augmented the demo with behavior protocol specifications; in addition, we have "wrapped" the `client` and `server` components into composite components `clientWrapper` and `serverWrapper`. The ADL of this demo, available in the file `WrappedHello-World.fractal`, is also shown in Figure 6.11.

Figure 6.11. ADL specification of the HelloWorld demo

```
<definition name="WrappedHelloWorld">
  <interface name="r" role="server"
    signature="java.lang.Runnable"/>
  <component name="clientWrapper">
    <interface name="r" role="server"
      signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <component name="client">
      <interface name="r" role="server"
        signature="java.lang.Runnable"/>
      <interface name="s" role="client" signature="Service"/>
      <content class="ClientImpl"/>
      <protocol value="?r.run{!s.print}*"/>
    </component>
    <binding client="this.r" server="client.r"/>
    <binding client="client.s" server="this.s"/>
    <protocol value="?r.run{!s.print}*"/>
  </component>
  <component name="serverWrapper">
    <interface name="s" role="server" signature="Service"/>
    <component name="server">
      <interface name="s" role="server" signature="Service"/>
      <content class="ServerImpl"/>
      <attributes signature="ServiceAttributes">
        <attribute name="header" value="-> "/>
        <attribute name="count" value="1"/>
      </attributes>
      <controller desc="primitive"/>
      <protocol value="?s.print*"/>
    <binding client="this.s" server="server.s"/>
    <protocol value="?s.print*"/>
  </component>
  <binding client="this.r" server="clientWrapper.r"/>
  <binding client="clientWrapper.s" server="serverWrapper.s"/>
  <protocol value="?r.run;?r.run*"/>
</definition>
```

By running the HelloWorld example with the command `ant execute`, we obtain the following output:

```
CLIENT created
SERVER created
starting checker for component client with protocol
  ?r.run{!s.print}*
starting checker for component server with protocol
  ?s.print*
starting checker for component WrappedHelloWorld with protocol
  ?r.run;?r.run*
```

```
starting checker for component clientWrapper with protocol
  ?r.run{!s.print}*
starting checker for component serverWrapper with protocol
  ?s.print*
Server: print method called
    at ServerImpl.print(ServerImpl.java:35)
    at org.objectweb.fractal.julia.generated.C3b8aff70_0. \
         print(INTERCEPTOR[Service])
    at org.objectweb.fractal.julia.generated.C41c1ff86_0. \
         print(INTERCEPTOR[Service])
    at org.objectweb.fractal.julia.generated.C2deafae5_0. \
         print(INTERCEPTOR[Service])
    at org.objectweb.fractal.julia.generated.Ca0b05a1f_0. \
         print(INTERFACE[Service])
    at org.objectweb.fractal.julia.generated.C9ec05a0f_0. \
         print(INTERCEPTOR[Service])
    at ClientImpl.run(ClientImpl.java:35)
    at org.objectweb.fractal.julia.generated.C600cae0c_0. \
         run(INTERCEPTOR[Runnable])
    at org.objectweb.fractal.julia.generated.C78281da2_0. \
         run(INTERCEPTOR[Runnable])
    at org.objectweb.fractal.julia.generated.C78281da2_0. \
         run(INTERCEPTOR[Runnable])
    at org.objectweb.fractal.julia.generated.C6a0cd3b_0. \
         run(INTERFACE[Runnable])
    at org.objectweb.fractal.behprotocols.adl.Launcher. \
         main(Launcher.java:105)
Server: begin printing...
-> hello world
Server: print done.
rtcheck: client: protocol satisfied
rtcheck: server: protocol satisfied
rtcheck: WrappedHelloWorld: protocol satisfied
rtcheck: clientWrapper: protocol satisfied
rtcheck: serverWrapper: protocol satisfied
```

A variation of this example featuring incorrect protocols is available in the file `WrappedHello-WorldIncorrect.fractal`. The runtime-check framework can in principle detect two kinds of errors: bad activity (event occurring when not permitted by the protocol), and incorrect end (component stops when not permitted by the protocol). The first kind of error is demonstrated on the `clientWrap-per` component. The erroneous protocol `?r.run{?r.run;!s.print}*` asks for a nested call of `r.run` before issuing `s.print`. The error is reported by a `ProtocolViolationException`. The second kind of error can be observed in the top-level component (WrappedHelloWorldIncorrect), where the erroneous protocol `?r.run;?r.run;?r.run*` requires that the method r.run is called at least twice. The error is reported at the time the demo stops by printing message `"protocol does not permit to stop here"`.

By experimenting with configuration of the runtime-check framework via properties, the various ways to handle an error can be observed. The alternative ADL file `WrappedHelloWorldIncor-rect.fractal` may be easily launched with the command

```
ant -Drun.parameters="WrappedHelloWorldIncorrect r" execute
```

6.4.4. Case Study: Applying runtime-checker on the Airport internet lounge demo

To demonstrate the runtime-checking on a non-trivial case study, we have used the implementation of the airport internet lounge demo described in the Demo description [http://kraken.cs.cas.cz/ft/doc/demo/Demo-Description.pdf]; technical details of the implementation are described in the separate document Demo - implementation notes [http://kraken.cs.cas.cz/ft/doc/demo/ftdemo.html]. We assigned runtime protocols to all demo components. There are minor differences between protocols for static and runtime checking.

In the runtime protocols, we had to remove the numbered suffixes from method names (e.g., TokenInvalidated_1 or TokenInvalidated_2) for methods where they were required for static checking. The suffixes are used to distinguish processing of several parallel calls of the same method. As they all represent a single method in the implementation, the Julia interceptor will know only the method name without a suffix when passing it to the runtime checker. As the core of the runtime checker is the same as the core of the static checker, which does not interpret the method suffixes (i.e., TokenInvalidated_1 and TokenInvalidated_2 are simply two different methods for the static checker core), the runtime checker would not be able to match a method name from the interceptor (without suffix) to a method name from a static behavior protocol (with suffix). This is the reason why we had to remove the suffixes and to create the separate runtime protocols.

Another feature used in static procotols are atomic actions. In the protocols for demo components, they are used to specify the synchronization behavior for some methods and to distinguish the initialization stage and "running" stage of components, as these two behaviors are much more easy to describe and comprehend with atomic actions. As an atomic action requires that all the methods it contains are processed at the same time (note that this is stronger than just a "simple" parallel operator). However, the runtime checker is not multithreaded, and all method callbacks from the Julia interceptors are processed in a sequential order. This means that is makes no sense to use the atomic actions in runtime protocols. Our solution was to replace all atomic actions with other "standard" behavior protocol constructs.

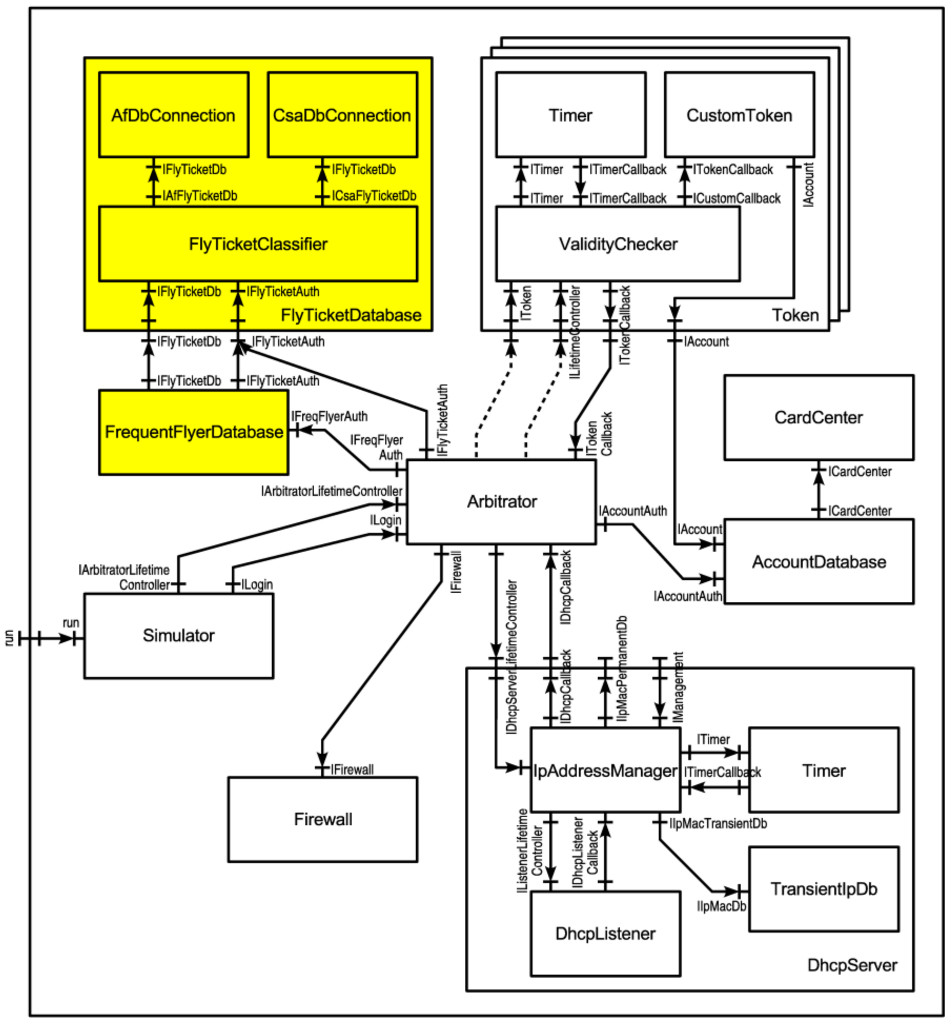
The Fractal implementation demo is in fact a set of "independent" components that are only connected to communicate with each other. However, as the components are designed to serve to the users of the system, none of them is able to work autonomously. In order to function, the components must receive requests from the "outer" world (their environment). The three components responsible for such communication are the DhcpServer component (more precisely, one of its subcomponents, the DhcpListener), the Arbitrator and the AccountDatabase. To simulate the environment of these components, we created the Simulator component. It implements a simple hard-wired test of all the "client" accessible methods of the demo components - i.e., it emulates requests accepted from 3 virtual clients passed to the Arbitrator component (via a virtual web server) and also calls several methods on the DhcpListener component simulating the DHCP protocol packets coming from clients. Here is the main part of the Simulator run method:

```
iArbitratorLifetimeController.Start();

byte[] mac1=new byte[] { 0, 0, 0, 0, 0, 0 };
byte[] mac2=new byte[] { 0, 0, 0, 0, 0, 1 };
byte[] mac3=new byte[] { 0, 0, 0, 0, 0, 2 };

InetAddress addr1=dhcpListener.RequestNewIpAddress(mac1);

iLogin.LoginWithAccountId(addr1,"","");

InetAddress addr2=dhcpListener.RequestNewIpAddress(mac2);
InetAddress addr3=dhcpListener.RequestNewIpAddress(mac3);
```

```
iLogin.LoginWithFlyTicketId(addr3,"");
iLogin.LoginWithFrequentFlyerId(addr2,"");

dhcpListener.RenewIpAddress(mac1,addr1);

dhcpListener.RenewIpAddress(mac1,addr1);

dhcpListener.ReleaseIpAddress(mac1,addr1);
iLogin.Logout(addr3);
dhcpListener.ReleaseIpAddress(mac3,addr3);

iLogin.Logout(addr2);
dhcpListener.ReleaseIpAddress(mac2,addr2);
```

The whole architecture used in the case study is shown in Figure 6.12.

Figure 6.12. Architecture of the demo



To run the demo (with runtime checking), simply go to the `demo-proto` directory and type `ant check-runtime`. The most important part of the checking output is the following (full listing can be found in a separate document [TXT] [http://kraken.cs.cas.cz/ft/doc/demo/Listing-check-runtime.txt]):

```
# ant check-runtime
...
```

```
     [java] rtcheck: CardCenter: protocol satisfied
     [java] rtcheck: AccountDatabase: protocol satisfied
     [java] rtcheck: Firewall: protocol satisfied
     [java] rtcheck: Arbitrator: protocol satisfied
     [java] rtcheck: FlyTicketClassifier: protocol satisfied
     [java] rtcheck: AfDbConnection: protocol satisfied
     [java] rtcheck: CsaDbConnection: protocol satisfied
     [java] rtcheck: FrequentFlyerDatabase: protocol satisfied
     [java] rtcheck: Timer: protocol satisfied
     [java] rtcheck: DhcpListener: protocol satisfied
     [java] rtcheck: TransientIpDb: protocol satisfied
     [java] rtcheck: IpAddressManager: protocol satisfied
     [java] rtcheck: ValidityChecker: protocol satisfied
     [java] rtcheck: Timer: protocol satisfied
     [java] rtcheck: ValidityChecker: protocol satisfied
     [java] rtcheck: Timer: protocol satisfied
     [java] rtcheck: ValidityChecker: protocol satisfied
     [java] rtcheck: Timer: protocol satisfied
     [java] rtcheck: FlyTicketDatabase: protocol satisfied
     [java] rtcheck: DhcpServer: protocol satisfied
     [java] rtcheck: org.objectweb.dsrg.behprotocols.demo.Token:
              protocol satisfied
     [java] rtcheck: org.objectweb.dsrg.behprotocols.demo.Token:
              protocol satisfied
     [java] rtcheck: org.objectweb.dsrg.behprotocols.demo.Token:
              protocol satisfied

BUILD SUCCESSFUL
Total time: 17 seconds
```

In order to demonstrate the runtime checker we prepared another version of the FlyTicketClassifier that does not call the IAfFlyTicketDb.GetFlyTicketsByFrequentFlyerId method as it should (the behavior protocol describing its correct behavior remains the same as in the previous example - with correct implementation of FlyTicketClassifier component). The runtime checker will then detect the incorrect component implementation. To run the demo with the faulty FlyTicketClassifier, go to the `demo-proto` directory and type `ant check-runtime-fail`. The output of the checking is the following (especially note the "rtcheck: FlyTicketClassifier: checker is already stopped due to error(s) found" message of the runtime checker) - full listing is in a separate file [TXT] [http://kraken.cs.cas.cz/ft/doc/demo/Listing-check-runtime-fail.txt]:

```
# ant check-runtime-fail
...
     [java] rtcheck: CardCenter: protocol satisfied
     [java] rtcheck: AccountDatabase: protocol satisfied
     [java] rtcheck: Firewall: protocol satisfied
     [java] rtcheck: Arbitrator: protocol satisfied
     [java] rtcheck: FlyTicketClassifier: checker is already
              stopped due to error(s) found.
     [java] Erroneous events [1] recorded for component
              FlyTicketClassifier
     [java] !IFlyTicketDb:GetFlyTicketsByFrequentFlyerId$
     [java] Trace [5 of 5] recorded for component
              FlyTicketClassifier
     [java] ?IFlyTicketAuth:CreateToken^
     [java] !IFlyTicketAuth:CreateToken$
     [java] ?IFlyTicketDb:GetFlyTicketsByFrequentFlyerId^
     [java] !ICsaFlyTicketDb:GetFlyTicketsByFrequentFlyerId^
     [java] ?ICsaFlyTicketDb:GetFlyTicketsByFrequentFlyerId$
```

```
    [java] rtcheck: AfDbConnection: protocol satisfied
    [java] rtcheck: CsaDbConnection: protocol satisfied
    [java] rtcheck: FrequentFlyerDatabase: protocol satisfied
    [java] rtcheck: Timer: protocol satisfied
    [java] rtcheck: DhcpListener: protocol satisfied
    [java] rtcheck: TransientIpDb: protocol satisfied
    [java] rtcheck: IpAddressManager: protocol satisfied
    [java] rtcheck: ValidityChecker: protocol satisfied
    [java] rtcheck: Timer: protocol satisfied
    [java] rtcheck: ValidityChecker: protocol satisfied
    [java] rtcheck: Timer: protocol satisfied
    [java] rtcheck: ValidityChecker: protocol satisfied
    [java] rtcheck: Timer: protocol satisfied
    [java] rtcheck: FlyTicketDatabase: protocol satisfied
    [java] rtcheck: DhcpServer: protocol satisfied
    [java] rtcheck: org.objectweb.dsrg.behprotocols.demo.Token:
             protocol satisfied
    [java] rtcheck: org.objectweb.dsrg.behprotocols.demo.Token:
             protocol satisfied
    [java] rtcheck: org.objectweb.dsrg.behprotocols.demo.Token:
             protocol satisfied

BUILD SUCCESSFUL
Total time: 17 seconds
```

In this demo, we have intentionally made the FlyTicketClassifier component non-compliant with its protocol. This is reported as the `ProtocolViolationException` along with the trace that lead to the protocol violation. The other protocols were satisfied, as reported in the output.

6.5. Code analysis of primitive components

6.5.1. Getting started

Code analysis of primitive Fractal components is performed by the Java PathFinder model checker (JPF) in cooperation with the protocol checker for code analysis. The tool accepts a program composed from a target primitive component and its environment as input, and traverses both the state space determined by the program and the state space determined by the protocol.

The easiest way to apply code analysis to an application is to start the application augmented with behavior protocol specifications (Section 6.1.1) and also with necessary information for the environment generator (Section 6.5.3).

6.5.2. Julia configuration

The environment generator must at runtime get for a component being checked the information stored in the ADL definition of the component. This includes the data provided by the protocol and environment controllers, i.e., the frame protocol of the component, the name of the class with value sets, etc. Therefore, an environment controller has to be attached to each component in a similar way as it is done for protocol controller. Specifically, it is necessary to extend the Julia configuration (i.e., the julia.cfg file) in the following way:

```

# Protocol Controller interface
...

# Protocol Controller implementation
...
```

```
# Environment Controller interface
(environment-controller-itf
  (environment-controller
      org.objectweb.fractal.behprotocols.EnvironmentController)
)

# Environment Controller implementation
(environment-controller-impl
  ((org.objectweb.fractal.julia.asm.MixinClassGenerator
    EnvironmentControllerImpl
    org.objectweb.fractal.julia.BasicControllerMixin
    org.objectweb.fractal.behprotocols.julia.
      EnvironmentControllerMixin
  ))
)


# Environment Controller added to "primitive" component kind
(primitive
  (
    'interface-class-generator
    (
      'component-itf
      'binding-controller-itf
      'super-controller-itf
      'lifecycle-controller-itf
      'name-controller-itf
      'protocol-controller-itf
      'environment-controller-itf
    )
    (
      'component-impl
      'container-binding-controller-impl
      'super-controller-impl
      'lifecycle-controller-impl
      'name-controller-impl
      'protocol-controller-impl
      'environment-controller-impl
    )
    (
      (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
        org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
      )
    )
    org.objectweb.fractal.julia.asm.MergeClassGenerator
    'optimizationLevel
  )
)
```

6.5.3. Running the check of primitive components

To make a Fractal application subject to code analysis of primitive components, it is necessary to (i) configure Julia in a proper way and (ii) define a frame protocol and environment-specific information (like name of a class with value sets, etc) for each primitive component in ADL.

We demonstrate the code-checking framework on a sample component application involving the `Client` and `Logger` components (Section 1.2.1). Complete implementation of the example can be found in the `examples/logger` directory.

As for Julia, it is necessary to add a configuration file that supports both the protocol and environment controllers (Section 6.5.2) to the `julia.config` system property and to turn on storing generated classes to a temporary directory via the `julia.loader.gen.dir` system property. Storing classes generated by Julia on disk is necessary for model checking of Fractal components with Java PathFinder to work. More specifically, it is because of our re-implementation of Java classloaders for JPF via its MJI abstraction (Section 5.3.2), which assumes that classes generated by Julia are stored on disk and not only in memory.

The ADL of the example, available also in the file `LoggerDemo.fractal`, is the following:

```
<definition name="logger.LoggerDemo">
  <interface name="run" role="server"
    signature="java.lang.Runnable"/>

  <component name="client">
    <interface name="log" role="client" signature="logger.Log"/>
    <interface name="run" role="server"
      signature="java.lang.Runnable"/>
    <content class="logger.ClientImpl"/>
    <protocol value="?run.run { !log.open; !log.log; !log.log;
      !log.close }"/>

    <environment>
      <valuesets classname="logger.LoggerEnvValues"/>
    </environment>

  </component>

  <component name="logger">
    <interface name="log" role="server" signature="logger.Log"/>
    <content class="logger.LoggerImpl"/>
    <protocol value="?log.open; ?log.log * ; ?log.close"/>

    <environment>
      <valuesets classname="logger.LoggerEnvValues"/>
    </environment>

  </component>

  <binding client="client.log" server="logger.log"/>

  <binding client="this.run" server="client.run"/>

  <protocol value="?run.run"/>
</definition>
```

The `classname` attribute of the `valuesets` element denotes the class that provides sets of values to be used as method parameters by the generated environment. In our example, the `logger.Logger-EnvValues` class is used for this purpose.

The process of code checking can be started with the command `ant check-code`, which first creates (or cleans) the temporary directory where the class files generated by both Julia and the environment generator are stored, and then starts the Fractal ADL launcher with the following command:

```
java -Dfractal.provider=org.objectweb.fractal.julia.Julia \
 -Djulia.loader=org.objectweb.fractal.julia.loader.DynamicLoader \
 -Djulia.config=etc/julia.cfg,../../output/dist/etc/julia-proto.cfg \
 -Djulia.loader.gen.dir=tmp \
 -Xmx256M
 org.objectweb.fractal.behprotocols.adl.Launcher \
     -checkjpf tmp logger.LoggerDemo
```

Notice that it is necessary to pass the `Launcher` class two additional arguments: `-checkjpf`, which turns code checking with JPF on, and the path to the directory for generated class files (`tmp` in the example), followed by the definition of the target component.

When checking the Logger example (via the `ant check-code` command), the output may take the following form:

```
...
    [java] Checking implementation of primitive components with JPF
    [java] Checking component client ...
    [java] Component client ... OK (tmp/client_Output.txt)
    [java] Checking component logger ...
    [java] Component logger ... OK (tmp/logger_Output.txt)
...
```

There are two lines of text printed for each primitive component in the hierarchy. First, the text `Checking component <component's name> ...` is printed before the checking of the component actually starts. Then, after the checking of the component is completed, the text `Component <component's name> ... [OK|ERROR] (<name of file with details>)` is printed. The `OK` message is printed only if no errors were found during checking, otherwise the `ERROR` message is printed. In both cases, the name of file with details is displayed in brackets next to the `OK/ERROR` message. We decided to store detailed output (like error traces, number of states, etc) in a separate file, as the output can be quite complex.

6.5.4. Case Study: Applying code analysis on the Airport internet lounge demo

To demonstrate the code-checking framework on a non-trivial case study, we have used the same application as for demonstration of the runtime-checking framework, i.e., the implementation of the airport internet lounge demo. For the purpose of code analysis, we had to define certain environment-related information for each primitive component in ADL.

More specifically, we had to define the so-called *user stub* for each primitive component, and in the case of the `DhcpListener` component, we also had to define several so-called *user drivers*. All these definitions are stored inside the `environment` element in addition to the `valuesets` subelement. Moreover, we also had to provide a simplified version of component's frame protocol as a specification of environment's behavior for several primitive components (e.g. `Arbitrator`) in order to make checking of these components feasible with respect to CPU time and memory requirements. In such a case, the environment is generated from the simplified frame protocol of the target component, but checking is still done against the original frame protocol.

The purpose of user stubs and user drivers is to extend the generated environment of a component with the functionality of the `Simulator` class (Section 6.4.4), since we check the primitive components one-by-one, and therefore it is not possible to use the `Simulator` class in the same way as in the case of runtime checking.

The `userstub` element contains the `file` attribute which denotes the file with Java code that initializes certain fields of the `Simulator` class, which are used by some components from the Demo. As these components assume that the fields of the `Simulator` class are set to meaningful values at runtime, it is necessary to initialize those fields also for the purpose of code analysis with the Java PathFinder.

The `userdriver` element contains the `event` attribute, which denotes a protocol event, and the `file` attribute which denotes the file with Java code that is used to indirectly invoke the event. The concept of user drivers is useful especially for components which invoke some methods on their required interfaces as a reaction to some outer world events - an example of such a component is the `DhcpListener` component.

Files representing user stubs and drivers for the Demo are available in the `demo-env/stubs` and `demo-env/drivers` directories, respectively.

In addition to defining environment-related information in ADL and providing user-defined stubs and drivers, we had to create stub implementations of several components from the Demo, which are referenced by fields of the Simulator component. All these stub implementations belong to the `org.objectweb.dsrg.behprotocols.demo.env` package. Main purpose of these manually created stubs is to bring down the requirements on CPU time and memory that are necessary for code checking of primitive components with Java PathFinder. Moreover, we had to create a special implementation of the Simulator component for code analysis also for the purpose of reducing time and space requirements of the checking process; therefore, there are two versions of the Simulator component - one for runtime checking (in the `SimulatorRun.source` file) and one for code analysis (in the `SimulatorJpf.source` file), the proper one being selected during compilation and copied to the `Simulator.java` file.

References

[PTA] Mach, M., Plasil, F., Kofron, J.. *Behavior Protocol Verification: Fighting State Explosion*. Published in the International Journal of Computer and Information Science, Vol.6, Number 1, ACIS, ISSN 1525-9293, pp. 22-30, Mar 2005.

[BCS] Bruneton, E., Coupaye, T., Stefani, J.B.. *The Fractal Component Model*. Draft 2.0-3, February 5, 2004, `http://fractal.objectweb.org/specification/`.

[PV02] Plasil, F., Visnovsky, S.. *Behavior Protocols for Software Components*. IEEE Trans. Software Eng. 28(11), 1056-1076, 2002.

[AP05] Adamek, J., Plasil, F.. *Component Composition Errors and Update Atomicity: Static Analysis*. Journal of Software Maintenance and Evolution: Research and Practice 17(5), Sep 2005.

[AP04] Adamek, J., Plasil, F.. *Partial Bindings of Components - any Harm?*. Presented at the SACT 2004 Workshop, Busan, Korea (held in conjunction with the APSEC 2004 conference), and published in the Proceedings of APSEC 2004, IEEE Computer Society, ISBN 0-7695-2245-9, pp. 632-639, Nov 2004.

[CGP] Clarke, E., Grumberg, O., Peled, D.. *Model Checking*. MIT Press, Jan 2000.

[JPF] Mehlitz, P.C., Visser, W., Penix, J.. *The JPF Runtime Verification System*. NASA Ames Research Center, available at `http://javapathfinder.sourceforge.net/`.

[PPK] Parizek, P., Plasil, F., Kofron, J.. *Model Checking of Software Components: Making Java PathFinder Co-operate with Behavior Protocol Checker*. Tech. Report No. 2006/2, Dep. of SW Engineering, Charles University, Jan 2006.

[PP06] Parizek, P., Plasil, F.. *Specification and Generation of Environment for Model Checking of Software Components*. Accepted for publication in Proceedings of Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2006), Vienna, Austria, ENTCS, Mar 2006.