Formal Methods

Table of contents

1. Formal Methods in Component-based Development

Formal methods in software development allow reasoning about various properties of the software. Particularly, in the component-based development, we are interested in properties related to the communication among components. A property may be application specific, like ensuring that methods of a component are invoked in correct sequence, or general, like dead-lock. The result of formal method application is often a proof, ensuring that the application is correct in some sense.

In order to reason about software and properties, we need a formalism for capturing the model of the software and expressing the properties. The formalism should be strong enough to capture important aspects of the system. On the other hand, the models should be practically decidable.

The SOFA 2 component model features the formalism of Behavior Protocols allowing users to describe component behavior. The integrated tool is then able to decide whether composition of components is correct or not.

2. Behavior Protocols

A behavior protocol is an expression describing the behavior of a component; the behavior means the activity on component interfaces. The activity comprises method calls and method returns on provided and required interfaces.

Since the components within an application are nested one in another, a property that can be verified is, whether all the subcomponents of a component "implement" the "specification", i.e., whether the subcomponents of a component when composed together act in a compatible way with respect to the specification of their supercomponent. This relation is often referred to as compliance.

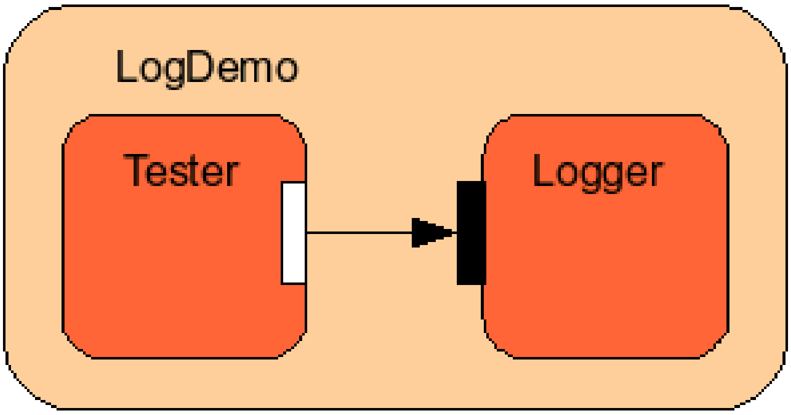
Compliance of component behavior in case of Behavior Protocols means absence of certain types of communication errors. In particular:

- Bad activity – a component emits a method call or return from a method call at a moment when no other component is able to accept such an event.
- No activity – no component is able to perform any event yet at least one component waits for an event.

In fact, there are two versions of Behavior Protocols. The original one, simply denoted as Behavior Protocols (BP) and Extended Behavior Protocols (EBP). Both formalisms share the aforementioned properties.

BP

Original BP can be seen as the minimal formalism allowing verification of absence of communication errors as described above. A specification of a component in BP takes the form of an expression representing the set of sequences of method calls and returns from method calls appearing on the component exported interfaces. To illustrate the formalism, consider the following example:



At this example, you can see a composite component `LogDemo` and two primitive components `Tester` and `Logger`. The specification corresponding to particular components follows:

```
    LogDemo:   NULL
    Tester:    !iLog.log*
    Logger:    ?iLog.log*
```

As you can see at this example, there is only a single binding within the component application `LogDemo`, in particular between components `Tester` and `Logger`. Since there is not any kind of input for the application, the protocol for the `LogDemo` component is empty (`NULL`).

The specification of the `Tester` component declares that the component is able to call (`!`) the `log` method on its `iLog` interface as many times as it wants to (`*`). Similarly, the `Logger` component is able to accept (`?`) these calls on its provided interface `iLog`. Hence, in this example, there is no communication error.

As the example suggests, calling of the method `m` on the interface `i` is captured by the exclamation mark – `!i.m`, while the method acceptance is preceeded by the question mark – `?i.m`. If there are any actions to be done during the method call, they are specified inside curly braces: `?i.m{!j.n;!j.m}`

Regarding the general syntax, the expression denoting method calls may be combined using common regular operators ('`+`' for alternative, '`;`' for sequence, and '`*`' for finite repetition. Besides, there is a special operator (being a syntactic sugar, nonetheless) '`|`' denoting parallel composition of its operands (i.e., it generates all interleavings of all traces yielded by its operands).

There are several limitations when using BP as the specification language. At the first place, BP entirely abstract from data, i.e., when data influences the control flow, it is necessary to overspecify the components not to omit a composition error. Also, paralelism must be explicitly modeled by parallel operator. This means that we must know the number of calling threads in advance. But the behavior description of the reusable component should not depend on particular architecture. The problem of multiple bindings can be solved by duplication and renaming methods, but this solution suffers with similar problems. Next, BP are limited to modeling regular behavior only, therefore stuff like general recursion is not allowed.

For more information about BP, see the papers: 1 2

EBP

Having in mind the limitations of original BP and facing the problems appearing when applying the formalism of BP onto a real-life sized case study (see this page for details), we identified several issues that should be subject of extensions of BP. In particular, method parameters influencing the reaction of methods called and local variables for storing the current component state turned out to be very beneficial for making the specification clearer and easily maintainable.

As to the syntax, we extended the original one to be able to capture the extensions introduced. The general structure of a EBP specification takes the following form:

```
component component_name {
  types {
    types_definition
  }

  vars {
    vars_definition
  }

  behavior {
    behavior_definition
  }
}
```

Within the `types` definition, enumeration data types for local variables and parameters are defined, while within the `vars` section, the local variables are defined. These variables are not visible from outside the component, hence only the specification in the `behavior` section may reference them. On contrary, the types are visible from outside because of passing parameters to methods from other components.

As an example, consider a similar component application as above. The difference is that the `Logger` component now uses an additional component to perform the log writing – `Writer`. The `Logger` component then have the following specification:

```
component Logger {
  types {
    SEVERITY_MODE {important, other}
  }

  vars {
    SEVERITY_MODE mode
  }

  behavior {
    ?iLog.setMode(SEVERITY_MODE newmode) {
      mode <- newmode;
    }
    |
    ?log.log(SEVERITY_MODE _messagetype) {
      switch(mode) {
        case other:
          !WriterIfc.write

        case important:
          switch(_messagetype) {
            case important: !WriterIfc.write
            default: NULL
          }
      }
    }*
  }
}
```

EBP solve the problem of missing data in original BP. However, the other BP limitations are still present there and are subject of ongoing research.

For more information about EBP, see the publications: 1

3. Standalone Tools

The formalisms described so far does not rely on the SOFA component system. In order to support a wide range of component systems, we provide a set of standalone tools for checking component behavior.

The tools are written in Java, so they may be executed on the wide range of platforms. However, the supporting scripts are available only for Unixes.

Architecture description

The minimal information necessary to check the architecture consists of

- Protocol associated with each component
- Description of composite components – namely
  - list of used component instances
  - list of bindings among instances

We believe that such information is shared by all hierarchical component systems. The description is sufficient also for the flat component systems, as they are just a special case of hierarchical ones.

Technically, the architecture description is a set of textual files. For each component A, there is a file `A.bp` containing the protocol associated with the A component. For each composite component B, there is a file `B.archbp` listing the component instances and bindings. In the case of flat component systems, there is just one file with the `archbp` extension.

The `LoggerDemo.archbp` follows:

```
    frame "null.bp"

    instantiate logger from "logger.bp"
    instantiate tester from "tester.bp"

    bind tester.log to logger.log
```

The `archbp` file contains the following constructs:

- The `frame "`fileName.bp`"` line associates the protocol with the architecture – this protocol describes the external behavior. In the Log example, the architecture is closed – the file `null.bp` contains just the `NULL` protocol. The parameter enclosed in the quotes contains the relative path to the file containing the specification.
- The `instantiate` instName `from "`fileName.bp`"` lines create named instances of particular components. The parameter instName is the name of the new instance. The second parameter enclosed in the quotes again contains the relative path to the specification.
- The `bind requiredInterface to providedInterface` line binds the interfaces of the instantiated components. The instances must be already created. External interfaces of the architecture are referenced using the implicit `this` instance.

It is recommenced, although not required, to store the architecture description in the directories following the hierarchical structure of the application. It is important to have just one protocol for each composite component used in different roles – first as the frame protocol and second as the instance protocol.

3.1. Checking BP — dChecker

Description

The dChecker (Distributed BP Checker) is a tool for detection of communication errors within a composite component. The behavior of components is specified in BP, not EBP. It can be invoked in two modes – local mode and distributed mode, which employs the resources of multiple machines.

Build

Requirements: jdk 1.5, ant, bptools.jar

1. Obtain BPTools distribution package
2. Checkout dChecker

   ```
       svn checkout
   svn://svn.forge.objectweb.org/svnroot/sofa/trunk/fm/dchecker
   ```

3. Modify the property `bptools_dist` in the `build.xml` file to contain the path to the bptools `dist` directory. The implicit value corresponds to the situation when dchecker and bptools directories share their parent directory.

4. Build the dChecker – run the ant in the directory containing the `build.xml` file.

   ```
       ant dist
   ```

   The result is the `dist` directory

Install

Requirements: jre 1.5, bash

1. Obtain the `dist` directory – you can build your own, or use our build.
2. Set the `PATH` variable to contain `dist/bin` directory.

Usage – Local mode

Invoke

```
    localChecker file.archbp
```

You can watch the progress of checking. If the architecture is correct, the checker finishes with the following message:

```
    Checked. M states visited
```

Otherwise, the shortest call stack to the error is printed out.

Usage – Distributed mode

1. Install the dChecker to all machines you intend to use.
2. Edit the setEnvironment script in the `dist/bin` directory and run it on all machines. The important variables are `DCHECKER_SERVERADDR` and `MEMORY` variables.

   ```
       . setEnvironment
   ```

3. Run the server

   ```
       dCheckerServer
   ```

4. Run a number of clients on various machines

   ```
       dCheckerClient
   ```

   The variable `DCHECKER_SERVERADDR` is used to reach the server
5. Load the task to the server through console

   ```
       dCheckerConsole serverAddr file.archbp
   ```

   The first argument is the address of the server and second is the file containing architecture description. The console is waiting for the response from the server. Then the result is reported to the user in the similar manner as in the local case. Once, you have the server and clients run, you can run the console many times.

3.2. Checking EBP — BP2Promela

Description

BP2Promela is a tool for transformation of EBP models to the Promela language – input language of the SPIN model checker. The result Promela model contains both – model corresponding to the input EBP model and EBP communication semantics including the definition of communication errors. Therefore, the communication errors in the input EBP model can be detected via inspecting the Promela model by the SPIN.

Build

Requirements: jdk 1.5, ant, bptools.jar

1. Obtain BPTools distribution package
2. Checkout BP2Promela

```
    svn checkout
svn://svn.forge.objectweb.org/svnroot/sofa/trunk/fm/bp2promela
```

3. Modify the property `bptools_dist` in the `build.xml` file to contain the path to the bptools `dist` directory. The implicit value corresponds to the situation when dchecker and bptools directories share their parent directory.
4. Build the BP2Promela – run the ant tool in the directory containing the `build.xml` file.

```
    ant dist
```

   The result is the `dist` directory

Install

Requirements: jre 1.5, bash, gcc, spin

1. Obtain the `dist` directory – you can build your own, or use our build.
2. Set the `PATH` variable to contain `dist/bin` directory. Both `gcc` and `spin` have to be executable from the current directory

Usage

In order to detect communication errors in the input model, invoke

```
    spinChain architectureDescr.archbp resultFile tmpDir
```

The script runs the whole chain – transformation of the model into the Promela language, spin to generate the verifier of the model in the C language, gcc to compile the verifier and the resulting binary verifier. If an error is found, the error trace is interpreted in terms of EBP events and printed out.

The first argument is the name of the file containing architecture description. The second argument is the name of the output file. It contains the verification result and might be used by Makefile to detect architectures that have to be rechecked. The last argument is the name of temporary directory to store the intermediate files.

3.3. BPTools

Description

BPTools is rather a library. It contains the parsers, parse tree transformations and similar things. However, as the output of these transformations might be useful, they can be written to a file.

Build

Requirements: jdk 1.5, ant

1. Checkout dChecker

```
    svn checkout
svn://svn.forge.objectweb.org/svnroot/sofa/trunk/fm/dchecker
```

2. Build BPTools

```
    ant dist
```

3. The result is the `dist` directory.

Install

Requirements: jre 1.5, bash, graphviz

1. Obtain the `dist` directory – you can build your own, or use our build.
2. Set the `PATH` variable to contain `dist/bin` directory.

Usage

In order to transform the input model, invoke

```
    BP2FSM inputFile < -NFSM  | -DFSM > fileName
```

The first parameter is the input model. The file contains either a single protocol or an architecture description. The second parameter is mandatory and determines the type of the required transformation. There are the following options for the second parameter:

- `-NFSM` – Print out the DOT file containing non-deterministic finite automatons representing particular protocols
- `-DFSM` – Print out the DOT file containing deterministic finite automatons representing particular protocols

The last parameter determines the output file.

4. SOFA 2 Integration

How to associate behavior specification with SOFA 2 component

The behavior specification is associated with a component frame. In the specification XML, inside the `frame` tag, there can be multiple behaviors (using different specification platforms) defined. For definition, the `behavior` tag with the mandatory attribute `name` is used. As an example, consider the specification of the `Logger` component frame:

```
    <?xml version="1.0" encoding="ASCII"?>
    <frame name="org.objectweb.dsrg.sofa.examples.logdemo.frame.Logger">

      <provides name="log"
itf-type="sofatype://org.objectweb.dsrg.sofa.examples.logdemo.itf.Log"/>

      <behavior name="bp">?iLog.log*</behavior>

      <behavior name="ebp">
      <![CDATA[
        component Logger {
              types {
                SEVERITY_MODE {important, other}
              }

              vars {
                SEVERITY_MODE mode
              }
              behavior {
                ?iLog.setMode(SEVERITY_MODE newmode) {
                  mode <- newmode;
                }
                |
                ?log.log(SEVERITY_MODE _messagetype) {
                  switch(mode) {
                    case other:
                      !WriterIfc.write

                    case important:
                      switch(_messagetype) {
                        case important: !WriterIfc.write
                        default: NULL
                      }
                  }
                }*
              }
            }
        ]]>
        </behavior>

    </frame>
```

How to check architecture

Having the specification of all components within an application, the Cushion tool can be used to verify the absence of communication errors. The command line takes then the following form:

```
    cushion bcheck (dchecker|ebpspin) architecture
```

`dchecker` causes checking compliance using specification in original Behavior Protocols, while `ebpspin` enables verification of specification in Extended Behavior Protocols using the Spin model checker. In the latter case, you need a working spin environment – see the "Checking EBP – BP2Promela" section.