Interface Improvements for IPDL

Kristina Sojakova Mihai Codescu

1 General Design

We have decided to integrate IPDL as a new language in the formal specification environment SpeX, developed and maintained by Ionuţ Ţuţu at the Institute of Mathematics of the Romanian Academy. The main motivation in using SpeX is that it provides us with parsing, static analysis and printing capabilities, as well a command-line interface. These are things that we don't have to reimplement ourselves. SpeX is also implemented in Maude and has been designed with the intention of making the cost of adding new languages and logics relatively small, and we greatly benefit from that. We have completed the integration in Spex of a fragment of IPDL: plain channels are supported but families of protocols are not, and the proof language is restricted to the part needed in the helloWorld case study[1]. For this language fragment we have implemented the concrete syntax (CS) and its representation in the abstract syntax (AS) takes place directly in the current prover state. Thus, the advantages of working with a CS are available: the source file is more readable, we can hide from the user some proof steps that only have to do with the way the protocols are represented, and we no longer have to use Maude identifiers, so names are no longer preceeded by a prime character. The support for IPDL is still quite minimal (and will be fully extended to the whole language as part of the milestone 7), but the interface improvements that we planned are already finished, as a proof of concept, for the fragment of IPDL that is currently supported.

2 Installation

In order to install IPDL, Maude must be installed, using the following steps:

1. download the 3.3.1 version of Maude from its GitHub site[2]
2. extract the files in the downloaded ZIP archive to a convenient directory:

   ```
   sudo unzip Maude-linux.zip -d /usr/local/maude/
   ```

3. make sure that all users may run maude:

[1] See `https://github.com/kristinas/IPDL-Maude/blob/main/lib/csHelloWorld.ipdl`.
[2] `https://github.com/SRI-CSL/Maude/releases`

```
sudo chmod a+x /usr/local/maude/Linux64/maude.linux64
```

4. make a discoverable link to the Maude executable, e.g.

   ```
   sudo ln -s /usr/local/maude/Linux64/maude.linux64
              /usr/local/bin/maude
   ```

5. add to the .bashrc file an environment variable for Maude:

   ```
   export MAUDE_LIB=/usr/local/maude/Linux64
   ```

Now we should be able to run **maude** at the command line.

IPDL can now be installed by downloading the archive at `https://github.com/kristinas/IPDL-Maude/blob/main/src/IPDL.tar.xz` and running the following commands:

```
tar -xJf IPDL.tar.xz && cd IPDL
autoreconf -i
./configure
make
sudo make install
```

Now we can run **ipdl** at the command line, and to test the helloWorld example, run **load csHelloWorld.ipdl**.

3 Highlighting

We have implemented a simple coloring scheme for protocols. For protocol compositions, which can be arbitrarily long, the color of each operand changes by looping through a list of three colors. This will have the benefit that in the case of branching, same protocols will have the same color on different branches (once IPDL will be fully supported in this implementation).

4 Current Protocol

The command **current-protocol** provides us the current protocol and the command **get-channel X** allows us to inspect only the channel **X** from the current protocol. Since the result of applying a rule to a configuration is no longer displayed as a full configuration, like in the original Maude implementation, which was difficult to read, this command becomes more important now.

5 Subproofs

The original goal here was displaying the protocol that we need to rewrite when doing a subproof. We have replaced this with a workflow that provides with full support for subproofs. The main idea can be easily explained with the help of the simple example of the **SYM** rule (adapted to channel), that says that if our current configuration is

C = pConfig(Sigma, Delta2, P2, I, O2, A)

and we can rewrite the configuration

C' = pConfig(Sigma, Delta1, P1, I, O1, A)

for some protocol **P1** to **C**, then we can rewrite **C** to **C'**. This means one has to provide the subproof that **C'** rewrites to **C**. In large case studies, working on a subproof is tedious as it requires us to write the start configuration by hand, as a separate proof.

The idea behind our workflow is that instead of working with a configuration, we work with a stack of configurations. When we want to apply a rule with subproofs, the start configurations for them will be known and can be pushed in the stack (in our case **C'**). The syntax for this is

try sym from P1 : todo

with the marker **todo** showing that the proof still needs to be provided. The user can then continue working directly on **C'** by adding proof steps (which may have subproofs themselves, and then the stack grows):

try sym from P1 : proofStep1 then proofStep2

and when the subproof is finished it can be closed

try sym from P1 : proofStep1 then proofStep2 done

and this entire construction is equivalent with an application of the **SYM** rule:

sym from P1 (proofStep1 then proofStep2)

This generalizes to the case of rules with two subproofs in the expected way, when the first subproof is finished we can pop from the stack and continue working on the second subproof. IPDL rules have at most two subproofs.

Being able to work on the subproofs directly makes proof discovery much easier and less error-prone.