CS-Note* 65

Bill J. Ellis
October 30, 2003

SPADEWiz: A SPADE Wizard

1 Introduction

This short note describes the intended role of a front end for SPADE. It is argued that the front end should not be viewed as the master of SPADE, but rather as a slave of SPADE that can be called upon by the user as desired. The proposed architecture for supporting this approach is described.

2 Role of a SPADE front end

The front end will present the user with an abstract view of the SPADE proof checker. The aim is to provide the user with powerful proof tactics that, behind the scenes, decompose into a series of SPADE proof checker commands. These powerful proof tactics should minimise user interaction in completing a proof and allow the user to guide the proof in a more intuitive manner.

Note there is no need for a flashy interface to achieve the above. However, a previous project demonstrated that neat interfaces can be created with minimum fuss and to good effect.

3 Front end as the master

In its normal behaviour SPADE reads in a VC file, takes commands from the user, and (all going well) eventually concludes true. The only output from SPADE is a copy of the commands entered by the user. Thus the problem is how to link SPADE into the front end. Two approaches are considered below.

3.1 Build front end into SPADE

We have a Sicstus Prolog port of SPADE and a rough idea of how it works. It would be possible to extend this code to create a larger program that includes both SPADE and the new front end. There are some advantages with this approach:

- **Direct access to SPADE database -** The front end will reside alongside SPADE, giving it direct access to the SPADE database. Thus SPADE can be used for loading VCs and rule files and the front end need only interrogate the resulting database accordingly.

However, there are also some disadvantages with this approach:

- **Changing SPADE is a bad idea -** SPADE is quite a large program. Even with good intentions to avoid changing the SPADE core program there is the possibility of adding errors to SPADE which may be difficult to trace. A possible solution is to package SPADE as a separate module and access it effectively as a Prolog library.
- **SPADE as library makes database read only -** If treating SPADE as a library its database should be accessed as read only. Thus to process the data inside SPADE will require working within another database.
- **SPADE is a bind -** SPADE is written in several thousand lines of code, slowing the debug cycle if consulted alongside the developed application. This might be avoided by appropriate use of saved states and a related makefile[1]. SPADE requires a specific Prolog environment. This environment is probably not ideal for new systems. Although quite minor issues, these indicate that directly extending SPADE should be avoided if possible.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

[1] Sicstus Prolog offers features to support the development of larger Prolog programs. However, these have subtle features and must be treated carefully.

- **Adding front end to command loop -** Once SPADE is started it enters its own command loop. The new front end may exist within this command loop. Alternatively the command loop may be cleanly terminated, the front end ran, then the command loop started again.

  Terminating the SPADE command loop presents problems as SPADE relies on state information stored explicitly in the Prolog database and implicitly on the Prolog stack. It might be possible to implement this by exploiting the SPADE save and load mechanisms. However, it would probably be more straight forward to have a single uninterrupted SPADE session.

  The front end may be embedded within the SPADE command loop. This can be achieved by creating a new SPADE command that launches the front end. This is viable so long as the front end is text based like SPADE. If the front end is not text based then problems arise. As SPADE must be started first the front end must be a Prolog program. Any graphical front end will be achieved by calls from Prolog. This is quite messy in practise and conceptually unintuitive (The interface should be pushing the proof checker, rather than the other way round).

3.2 Build front end outside SPADE

If developing within SPADE there is a natural tendency to partition into two systems, the SPADE library and the front end, interacting via a shared Prolog database. By making this database externally visible the two system can be developed separately. However, there are still some issues with this approach:

- **SPADE is the master -** If the front end is the master it must become involved in uninteresting issues like selecting which VC file and particular VC to work on. If in full control the front end might also be expected to support proof traversal operations, in particular undoing wrong proof steps. This would be quite complicated.

- **Staying in synch -** As SPADE is the master, the front end must open a fairly complex communication channel with SPADE to provide the behaviour desired. Keeping the front end in synch with SPADE over the duration of a proof will be difficult.

4 Front end as the slave

The important aspect of the new front end is tackling a problematic VC. The front end is more correctly viewed as a slave to SPADE, providing user assistance when desired. This kind of interaction is similar to that seen in a Microsoft Wizard[2]. The following interaction is envisaged:

> *Inside a SPADE session a user has trouble solving the current goal. They execute a command which stores the current goal to a file and launches the SPADE front end with this file. The front end tries to advance the goal, generating a SPADE command log if successful. At the termination of the front end SPADE resumes, executing any generated command log.*

4.1 Build front end outside SPADE

The front end can be built outside the SPADE system without worrying about the weaker communication with SPADE as the front end will only tackle the current SPADE goal. This has various advantages:

- **Focus on the interesting problems -** It will be possible to create a front end that is the master of SPADE. Although a worthwhile exercise, this does not directly address the problems of proof automation or proof interaction. These can be targeted directly by allowing SPADE to stay in control.

- **Clean presentation -** A 'SPADE wizard' that can be called upon as desired is a clean approach for exploring a proof focused front end for SPADE.

- **Neat graphical interface and Prolog access -** As argued in CS-note 49 the combination of Sicstus Prolog and Java via the Jasper interface gives ready access to the power of Prolog and the graphics facilities of Java. This approach becomes available as the front end will exist as a separate program.

[2] Only, helpful!

5 Implementing: SPADEWiz!

SPADE is the proof checker provided by Praxis Critical System. SPADEWiz is a wiz at SPADE, acting as a SPADE wizard.

The entire system is composed of two processes, SPADE and SPADEWiz, interacting via common files. Decent interprocess communication via sockets is not required[3]. The common files are:

```
/tmp/spadewiz/alarm/live.pl           -- Existance denotes SPADEWiz is running.
/tmp/spadewiz/alarm/resumechecker.pl  -- Existance denotes SPADE should resume.
/tmp/spadewiz/alarm/resumespadewiz.pl -- Existance denotes SPADEWiz should resume.
/tmp/spadewiz/result/log.cmd          -- Holds the result of SPADEWiz.
/tmp/spadewiz/state/dump.pl           -- Describes the current goal.
```

The interaction of the two systems is described in a UML statechart diagram shown in figure 1. Note that to support this interaction a slightly modified version of the SPADE checker (called checkerprime) is required.

6 Working with SPADEWiz

A development directory has been created for the initial SPADEWiz architecture. The main directories are:

```
./bin             -- The directory for executable files.
./Example         -- Some example SPARK code.
./CheckerPrime    -- The source code of checkerprime.
./Setup           -- Setup files for development on different machines.
./SPADEWiz        -- Source for SPADEWiz.
./SPADEWiz/java   -- The java side of SPADEWiz.
./SPADEWiz/prolog -- The prolog side of SPADEWiz.
./doc             -- Documents.
```

Important files for development are:

```
./GO                   -- To begin: source this file!
./buildspadewiz_java   -- Build the java side of SPADEWiz.
./buildspadewiz_prolog -- Build the prolog side of SPADEWiz.
./runspadewiz          -- Run SPADEWiz.
./settings             -- Settings. These are automatically used by GO.
```

Useful documentation resources:

```
http://java.sun.com/docs/books/tutorial/uiswing/          -- Swing tutorial.
http://java.sun.com/j2se/1.4.2/docs/api/                  -- Java API.
http://www.sics.se/sicstus/docs/latest/html/sicstus.html/ -- Sicstus prolog
http://www.sics.se/sicstus/docs/latest/html/jasper/       -- Jasper
```

Note the web can be unhelpful for Swing and Prolog. Heriot-Watt library has some good books. I found *Up to speed with Swing: user interfaces with Java foundation classes* by Steven Gutz to be a good practical book for getting things done in Swing[4].

[3]The two processes can sensibly be expected to run on the same machine. Besides, Prolog and Java socket programming would be a pain...

[4]Signs of a good book: pages falling out the middle and various pencil scribblings in the margins.

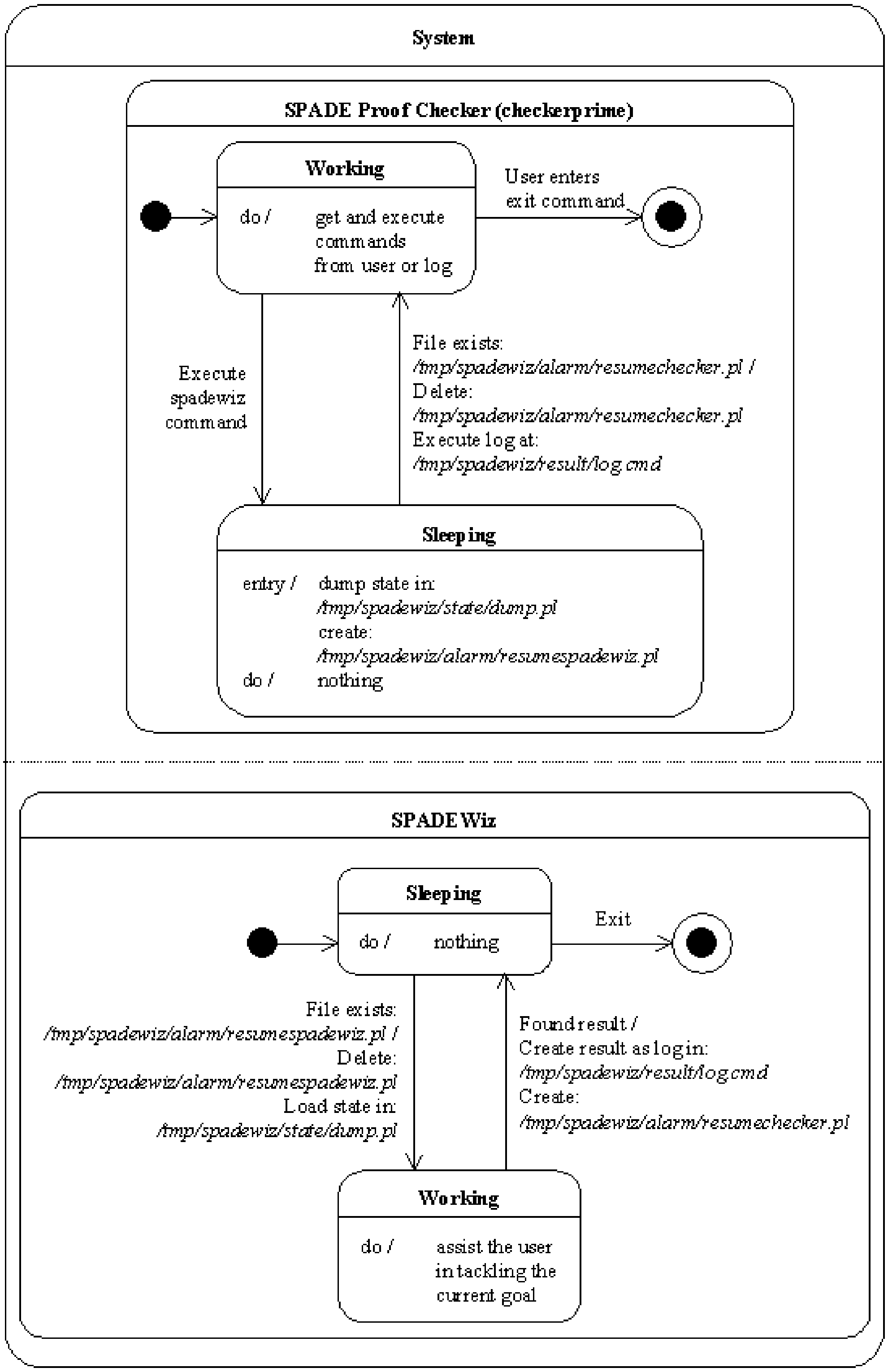

Figure 1: SPADE and SPADEWiz interaction