Wright Analysis Tutorial

James Ivers
September 28, 1998

This distribution is meant to be a self-contained, self-guided demonstration of the analysis of Wright [Wri98] specifications. In this distribution, we have combined a paper introducing Wright with several examples from the paper that are mechanically checkable. In the tutorial script to follow, we will explain how Wright specifications are checked, what the results look like, and how to interpret the results.

In the toolset we are developing at Carnegie Mellon University, a Wright specification is written in either an ASCII or LaTeX representation. The Wright specification is then translated into a CSP specification by a tool called *wright*. This tool also inserts some of the checks pre-defined by Wright (like connector deadlock freedom) into the CSP specification making them directly invocable from FDR menus. The CSP specification can then be checked by the FDR checker, where counter-examples may be viewed for failed checks.

Since FDR [FDR98] is a commercial tool, however, we cannot distribute it. We have instead taken screen shots of our use of FDR with these examples. This should serve to illustrate Wright analyses as well as give an idea of FDR's capabilities.

While the *wright* tool is not in this distribution either, an alpha version may be downloaded from the Wright website. Work on this tool is still in progress. It was used to perform most of the translations found in this package, but not all. Several features of the Wright language are not yet supported by the tool, nor are all of the pre-defined checks.

The following files are contained in this distribution:

this file	A script that will walk the reader through the tutorial.
wright-tosem97-revision.ps	A paper containing an introduction to Wright and several small example Wright specifications [AG97].
*.wrt	These are the Wright specifications from the paper encoded in an ASCII representation.[1]
*.fdr2	These are the same specifications after being converted to CSP. There is one .fdr2 file for each .wrt file.

The reader should read the paper contained in this distribution before proceeding with the tutorial. This tutorial does assume that the reader is familiar with CSP; for those not familiar with CSP, suitable introductory material can be found in either [Hoa85] or [Ros98].

Please send any questions or comments to wright-support@cs.cmu.edu.

[1]. The Wright language has changed slightly since the inlcuded paper was written. As such, there are some minor differences between the examples in the paper and the .wrt files. For example, in the paper, connector glue is sometimes defined using a *let p in q* style while the current language defintion calls for a *q where p* style. The current definition of Wright can be found in [All97].

1. Wright examples included

The specification files contained in this distribution correspond to the examples from the accompanying paper (wright-tosem97-revision.ps):

File	Example
bogus.*	Figure 4: connector Bogus
client-server.*	C-S-connector in Section 5.2 Note: A data type for x and y had to be added to the .fdr2 file in order to make this example checkable.
compat.*	Pipe from Figure 5, Datafile from Figure 6, and a variant of the Datafile from Figure 6. Note: This is an incomplete specification. It provides just enough to illustrate the port/role compatibility check.
pipe.*	Figure 5
shared-data1.*	Figure 4: connector Shared Data1
shared-data2.*	Figure 4: connector Shared Data2
shared-data3.*	Figure 4: connector Shared Data3

This tutorial will discuss a representative sample of these examples.

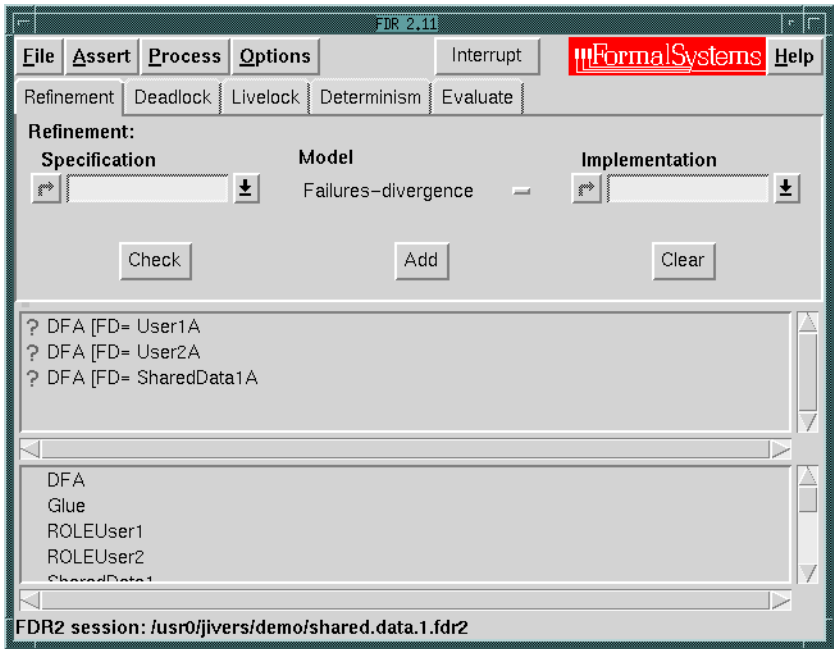
2. Using FDR to analyze Wright specifications

```
Style SharedData
Connector SharedData1
    Role User1 = set -> User1 |~| get -> User1 |~| Tick
    Role User2 = set -> User2 |~| get -> User2 |~| Tick

    Glue = User1.set -> Glue [] User2.set -> Glue
      [] User1.get -> Glue [] User2.get -> Glue [] Tick

End Style
```

Figure 1: shared-data1.wrt

Our first example is the file *shared-data1.wrt*. This file contains the Wright specification for the first Shared Data connector from Figure 4 of the accompanying paper and is shown in Figure 1. The accompanying file, *shared-data1.fdr2*, contains the same specification after being translated to CSP using the wright tool. In performing the translation, the tool attempts to maintain the structure of the original Wright specification so that any counter-examples found in checking assertions with FDR can be traced back to the original specification (consequently, a user rarely needs to look at the generated CSP specification).

After opening FDR and loading the *.fdr2* file for this example FDR displays the view shown in Figure 2. This is FDR's main window. From here we can select which assertion to check and are notified whether or not the assertion is true. While we could create new assertions on the fly using FDR's interface, when the Wright specification was translated, certain assertions were automatically inserted by the wright tool. These correspond to checks pre-defined in Wright. Assertions appear in the middle portion of the window.



Refinement:
Specification
Model
Failures-divergence
Implementation

? DFA [FD= User1A
? DFA [FD= User2A
? DFA [FD= SharedData1A

DFA
Glue
ROLEUser1
ROLEUser2

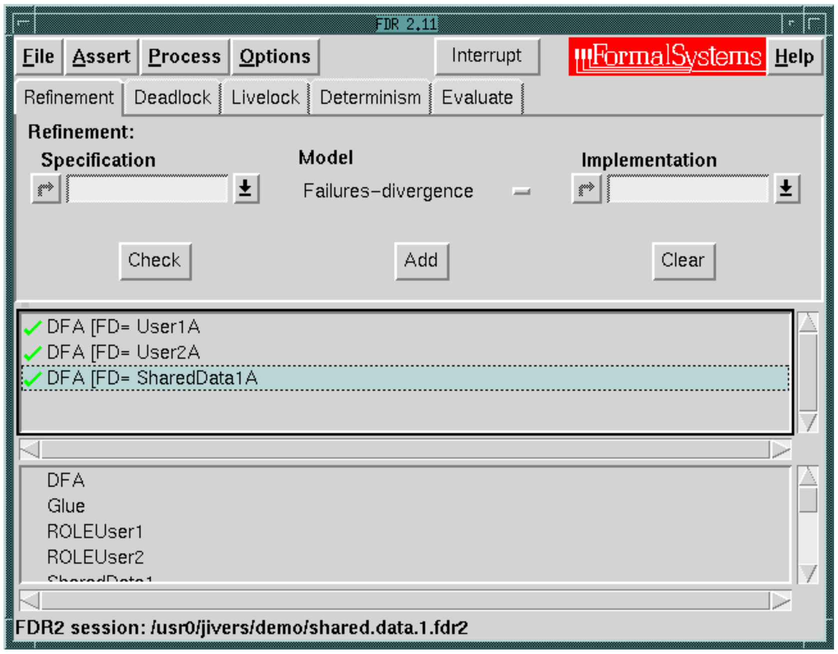
FDR2 session: /usr0/jivers/demo/shared.data.1.fdr2

Figure 2: FDR main window

In this case there are three assertions. Each assertion is on a separate line and is initially preceded by a question mark. This indicates that the assertions have not yet been checked. After checking an assertion, the question mark changes to either a check mark or an "X" depending on whether the assertion is true or false. To check an assertion, we simply double click on it. Should an assertion be false, we can again double click on the assertion to bring up a debugging view that displays a counter-example to the assertion.

The three assertions inserted in this example correspond to checks that are pre-defined by Wright. Specifically, the checks that appear here are Role Deadlock Freedom and Connector Deadlock Freedom. Looking back at the Wright specification, we notice that this example describes a connector that has two roles. As such, the first two assertions inserted by the wright tool assert that each role is free of deadlock. The third assertion inserted asserts that the connector as a whole (that is, the two roles in parallel with the glue) is free of deadlock. Each assertion is expressed as a refinement assertion (where "[FD=" is the symbol for the refinement operator under the Failures Divergence model); in this case, each asserts that some process is a refinement of an abstract, deadlock free process (DFA). If such a process is a refinement of DFA, then the process is free of deadlock. In order to compare an arbitrary process to DFA, however, the wright tool must produce a version of the process that renames all the events in that process to an abstract event name. Thus the process that is checked for deadlock freedom is a renaming of the original process and an "A" is appended to the name of the process to distinguish this abstract version.

Next, we double click on each assertion in turn. The resulting screen is shown in Figure 3; each assertion is true (all are marked with check marks). In the next example, we'll examine an assertion that is false.



Failures−divergence

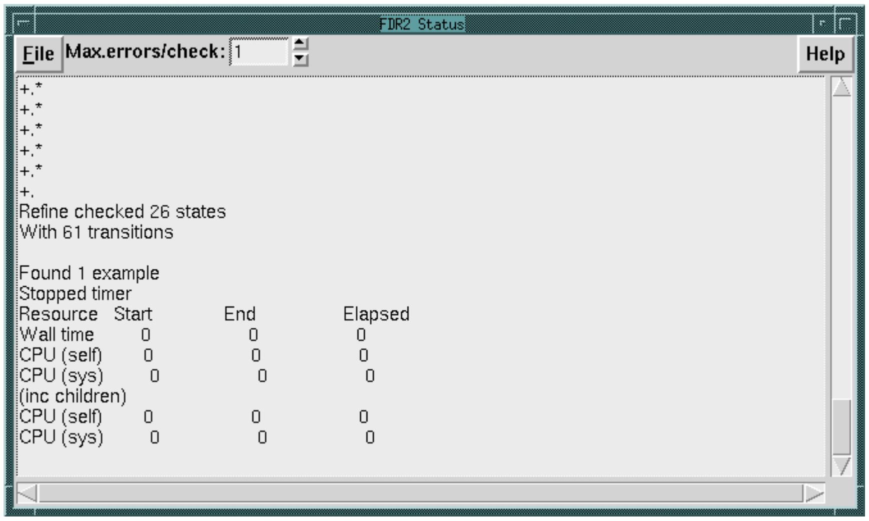
✓ DFA [FD= User1A
✓ DFA [FD= User2A
✓ DFA [FD= SharedData1A

DFA
Glue
ROLEUser1
ROLEUser2

FDR2 session: /usr0/jivers/demo/shared.data.1.fdr2

Figure 3: FDR after checking assertions

Before concluding this example, we select "Show status" from the Options menu. This gives us information about how many states were checked and how long the checking took. FDR builds a state transition model of the CSP processes then exhaustively examines all possible states and transitions for violations of a given refinement assertion. Depending on the complexity of the process, the internal model can become quite large. In this case, as shown in Figure 4, the model is relatively small. The status shown here is for the last assertion, Connector Deadlock Freedom. It contains 26 states and 61 transitions, and took so little time that the elapsed time shows up as zero seconds.



```
+.*
+.*
+.*
+.*
+.*
+.
Refine checked 26 states
With 61 transitions

Found 1 example
Stopped timer
Resource   Start        End          Elapsed
Wall time     0           0            0
CPU (self)    0           0            0
CPU (sys)     0           0            0
(inc children)
CPU (self)    0           0            0
CPU (sys)     0           0            0
```

Figure 4: Status window

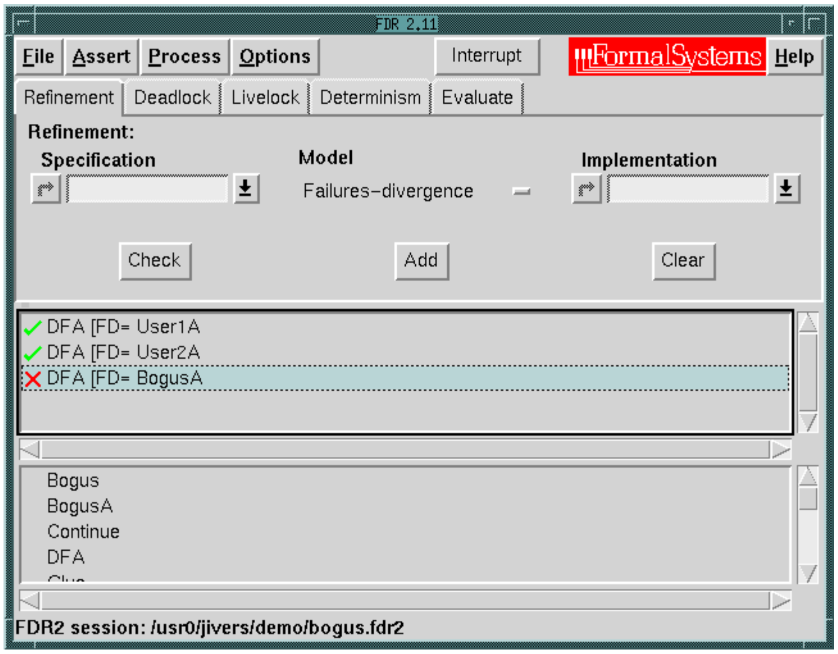
FDR also scales to much larger problems. We have used it to check models with approximately 3 million states and 17.5 million transitions in about 3 hours on a mid-size workstation. Additionally, FDR supplies some compression functions, which can be used to reduce the number of states and transitions that have to be explored in a given model. These functions are a comparatively advanced topic (and will not be covered in this demonstration), but they can be used effectively to manage state-space explosion in many circumstances.

3. Looking at counter-examples

```
Style SharedData
Connector Bogus
    Role User1 = set -> User1 |~| get -> User1 |~| Tick
    Role User2 = set -> User2 |~| get -> User2 |~| Tick

    Glue = User1.set -> Continue [] User2.set -> Continue []
            Tick
    where {
      Continue = User1.set -> Continue
            [] User2.set -> Continue
            [] User1.get -> Continue
            [] User2.get -> Continue
            [] Tick
    }

End Style
```

Figure 5: bogus.wrt

Next we'll open the file *bogus.fdr2* in FDR. This example (shown in Figure 5) is similar to the previous example, but was constructed with an intentional error. After checking the pre-defined assertions (as before), we get the results shown in Figure 6. As you can see, each of the roles is still free of deadlock. This time, however, the connector is not free of deadlock. Next we double click on this false assertion to view a counter-example.



Refinement:
Specification
Model
Failures-divergence
Implementation

✓ DFA [FD= User1A
✓ DFA [FD= User2A
✗ DFA [FD= BogusA

Bogus
BogusA
Continue
DFA
Glue

FDR2 session: /usr0/jivers/demo/bogus.fdr2

Figure 6: FDR after an assertion failed

FDR's debugging window is shown in Figure 7. From here, we can look at each process in the connector description to figure out why the connector deadlocks.

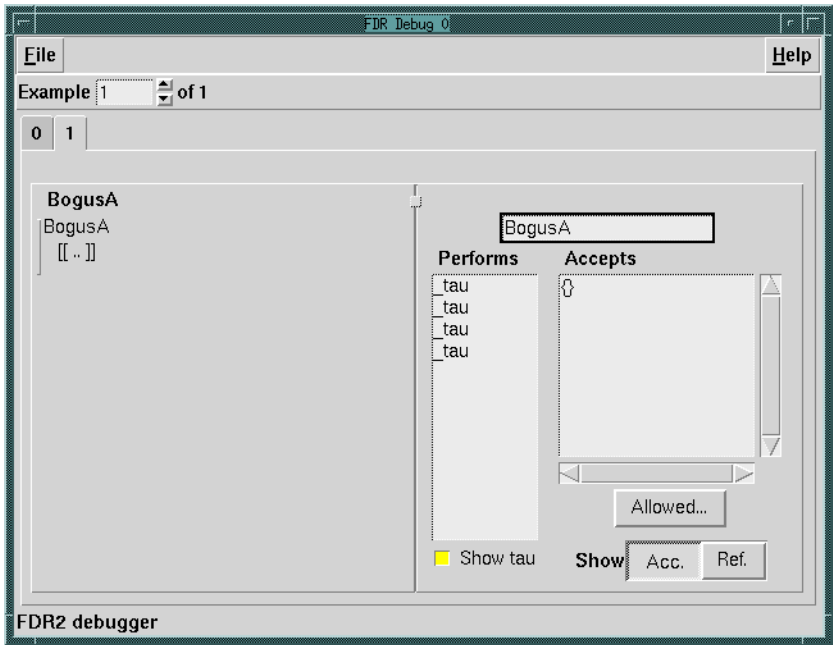


Figure 7: Debugging window

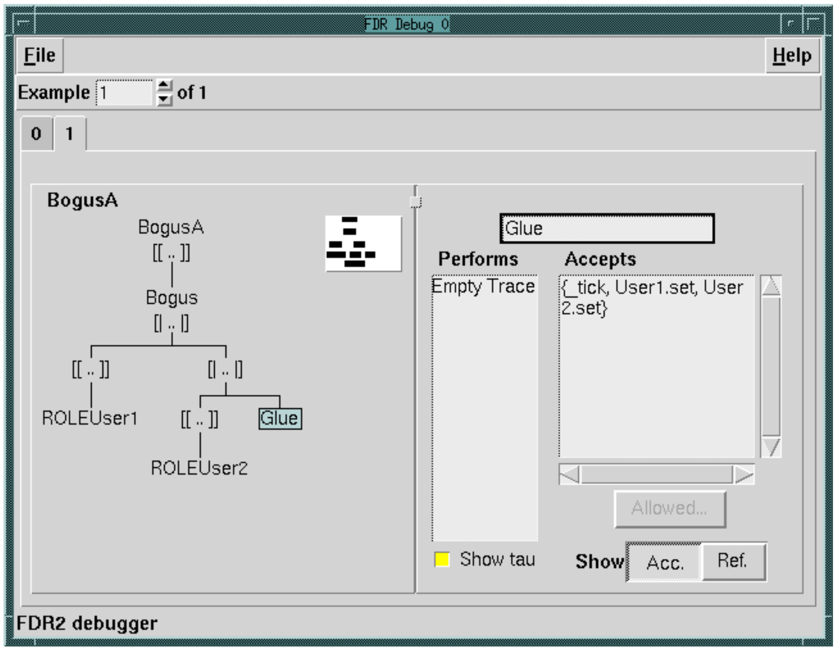
The left-most portion of this window is a tree representation of the Wright specification. Double-clicking on any "[[..]]" or "[| .. |]" will expand the current process definition into its component parts (a single process that has been renamed or a parallel composition of sub-processes, respectively).

The portion to the right of the divider shows the final status relative to the process that is currently selected (highlighted) in the tree view. It is divided into two halves, "Performs" and "Accepts." The Performs column lists the events that have been performed by the currently selected process up to the point at which the assertion failed (i.e., deadlock was detected in our example). The Accepts column lists the events that the currently selected process will accept (agree to perform) after the trace of events specified in the Performs column. The process refuses to perform any events not appearing in the Accepts column.

Looking at the initial view of the Bogus connector, we see that the connector has not performed any events. The Performed column contains only `tau`'s, which are internal transitions corresponding to internal choices in a Wright specification. There is an option at the bottom of the screen to filter these out. We also see that the connector is not willing to accept any events. This is the definition of deadlock. In order to see why the connector is stuck, we have to look at the subprocesses within the connector. Typically it is the case that the various subprocesses are willing to accept events, but they are not willing to accept the same events. Since they disagree, no events may be performed and the connector is deadlocked.

In Figure 8, we see the tree view fully expanded. The process BogusA simply consists of the two roles in parallel with the glue (see Section 6 of the accompanying paper). Two processes shown on the same level are parallel processes. Also, since the parallel operator is a binary operator, only two processes are shown in parallel at a time. Thus the expression

A || B || C is treated as A || (B || C).



Empty Trace

{_tick, User1.set, User2.set}

Figure 8: Debugging view of the Glue process

After selecting the glue process as a starting point, we can see in the Accepts column that the Glue is willing to accept three different events: `{_tick, User1.set, User2.set}`. Looking at the Accepts column in Figure 9, in which the first role (User1) is selected, we see that this process is only willing to accept the `get` event. In Figure 10 we see that the role User2 is also only willing to accept the `get` event. This is our mismatch. Both of the roles will only agree to perform a `get` event, while the glue insists that one of the roles must perform a `set` event. Since no two processes agree[2] on which event may be performed next, we have deadlock (for exactly the reason described in Section 5.2 of the accompanying paper).

[2]. Even the two roles do not agree on the `set` event. Recall that the definition of a connector calls for each role to be relabeled such that all events in the role are prefixed by the name of the role. Thus two roles will never synchronize on any event except `tick`, which is unaffected by relabeling. The relabeling shows up in the tree representation as the "[[..]]" nodes directly above the role processes. These nodes are the relabeled versions of the leaf role nodes; if we were to select one of these nodes, we would see the same event trace and acceptance set with the distinction that all events would be prefixed by the name of role.

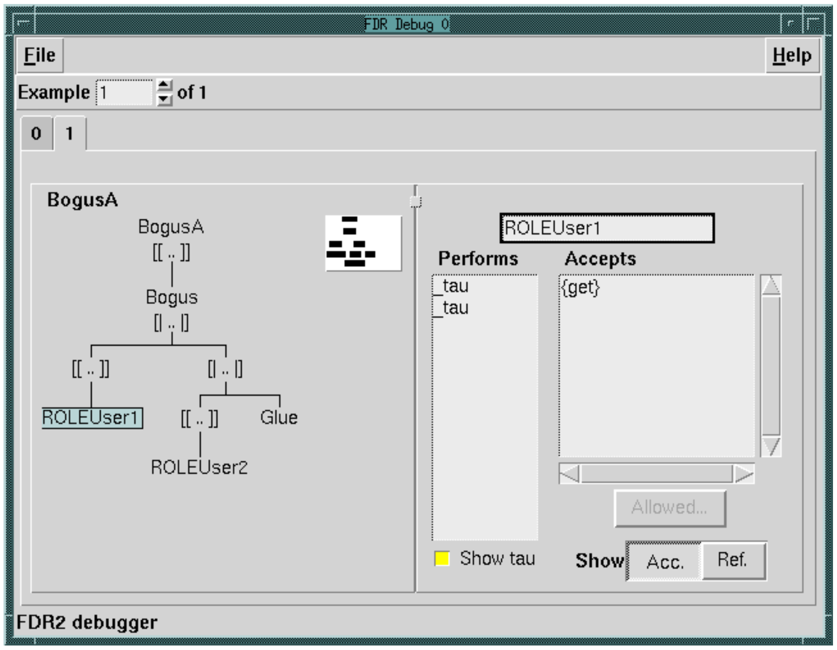


Figure 9: Debugging view of first Role

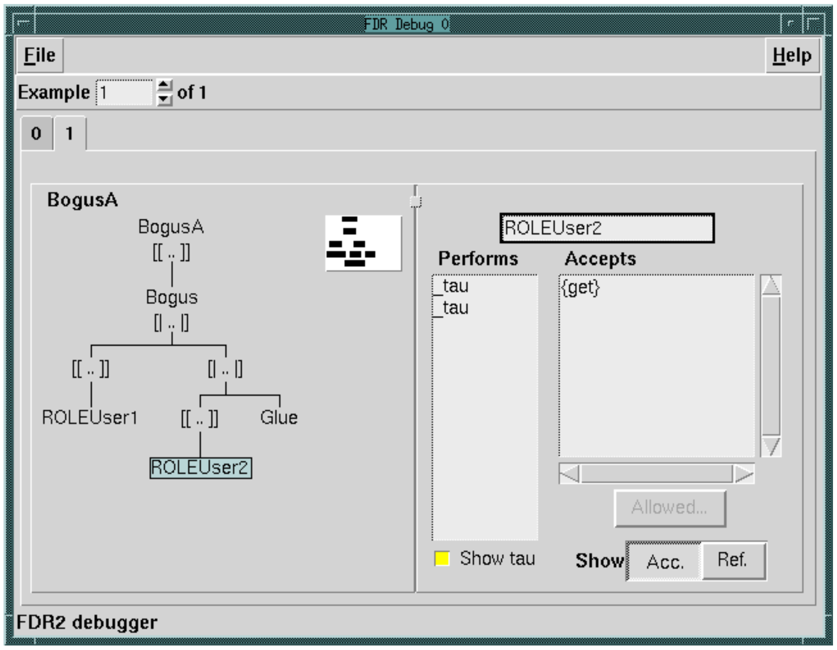


Figure 10: Debugging view of second Role

This concludes our look at how deadlock is detected and a counter-example is displayed in FDR.

4. Port/Role compatibility

Our final example examines a different kind of Wright check: checking the compatibility of a port with a role. This check is described in detail in Section 8.1 of the accompanying paper and will not be elaborated here.

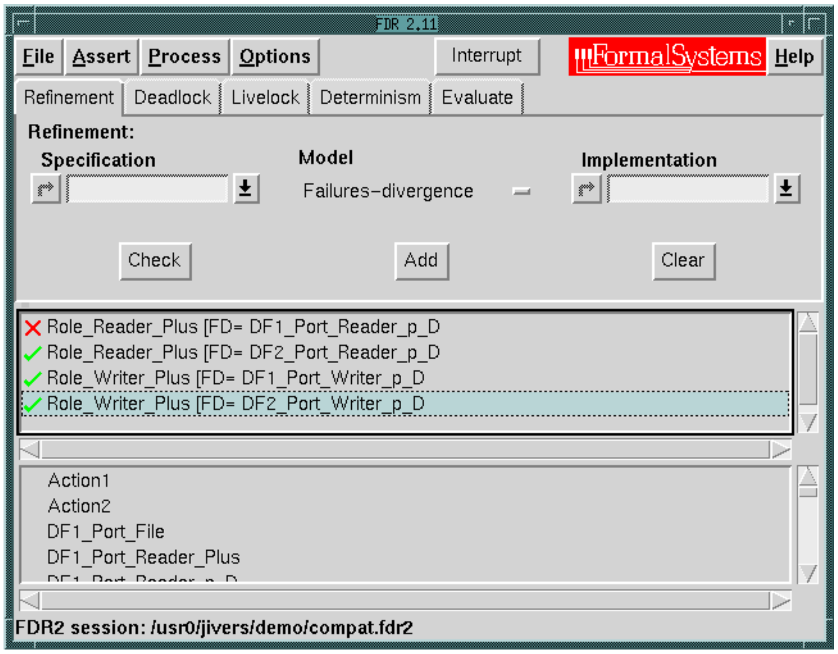
```
Style DataFiles
Component DataFile1
    Port File = Action [] Exit
    where {
      Exit = close -> Tick
      Action = read -> Action [] write -> Action
    }
    Computation = ...

Component DataFile2
    Port File = Action [] Exit
    where {
      Exit = close -> Tick
      Action = read -> File [] write -> File
    }
    Computation = ...

Connector Pipe
    Role Writer = write -> Writer |~| close -> Tick
    Role Reader = DoRead |~| ExitOnly
    where {
      DoRead = read -> Reader [] readEOF -> ExitOnly
      ExitOnly = close -> Tick
    }
    Glue = ...

End Style


Configuration TestCompatibility
Style DataFiles

Instances
    DF1 : DataFile1
    DF2 : DataFile2
    pipe1, pipe2 : Pipe

Attachments
    DF1.File as pipe1.Reader
    DF2.File as pipe2.Reader
    DF1.File as pipe1.Writer
    DF2.File as pipe2.Writer

End Configuration
```

Figure 11: compat.wrt

Looking at file *compat.wrt*, shown in Figure 11, you'll notice that this is a partial specification. Just enough is defined to allow us to illustrate Wright's port/role compatibility check. The Connector is the Pipe from Figure 5 of the accompanying paper. The two components correspond to the Datafile from Figure 6 of the accompanying paper, with one small difference. DataFile1 is a bad specification; it allows an initial choice of Action or Exit, but the definition of Action recurses to itself rather than to File. This means that the option to Exit is only offered once. DataFile2 corrects this problem in the expected way: after any event in Action, the description again offers a choice of Action or Exit.

To illustrate compatibility checking we will check whether each role of the Pipe (Reader and Writer) is compatible with the File port of the two Datafile specifications. Four separate assertions will be made to cover the combinations. Looking at *compat.fdr2* you'll see how we set up the port/role compatibility check. Unlike the other .fdr2 files, this one was hand generated. The lines starting with "assert" mark the assertions that will appear in FDR denoting the four port/role compatibility checks. As with the example in Section 9 of the accompanying paper, the construction of the refinement assertions requires defining the deterministic form of the roles and performing the necessary alphabet extensions.

As shown in Figure 12, the port of DataFile1 is not compatible with the Reader role of the pipe, but is compatible with the Writer role. The port of DataFile2 is compatible with both the Reader and Writer roles of the pipe. The paper gives a brief explanation of why the port of DataFile2 is compatible with both roles of the pipe, so that discussion will not be repeated here.



Refinement:
Specification Model Implementation
Failures–divergence

```
✗ Role_Reader_Plus [FD= DF1_Port_Reader_p_D
✓ Role_Reader_Plus [FD= DF2_Port_Reader_p_D
✓ Role_Writer_Plus [FD= DF1_Port_Writer_p_D
✓ Role_Writer_Plus [FD= DF2_Port_Writer_p_D
```

```
Action1
Action2
DF1_Port_File
DF1_Port_Reader_Plus
DF1_Port_Reader_p_D
```

FDR2 session: /usr0/jivers/demo/compat.fdr2

Figure 12: Results of compatibility checks

What is more interesting is that the bad specification, the port of DataFile1, is compatible with one role of the Pipe, but not the other. Looking first at the Writer role, this process always allows an internal (non-deterministic) choice between writing and closing (until it closes). The DataFile1 port, while not always allowing closing, does always allow writing. This is a valid refinement of the Writer role since it always allows one of the choices that the Writer is permitted. The fact that it is a more deterministic process (only allowing one of the choices Writer is permitted) is fine.

Seeing why the port is not compatible with the Reader role is a little trickier. When looking at a counter-example to a deadlock assertion, we only have to consider one specification.

When looking at a counter-example to a port/role compatibility check, we need to consider two specifications and figure out why one is not a refinement of the other. The view of the debugging window for each specification is shown in Figures 13 and 14. Figure 13 shows the view of the "refining process," the process that is supposed to be a refinement of the other process (the "refined process"). Figure 14 shows the view of the refined process.[3]

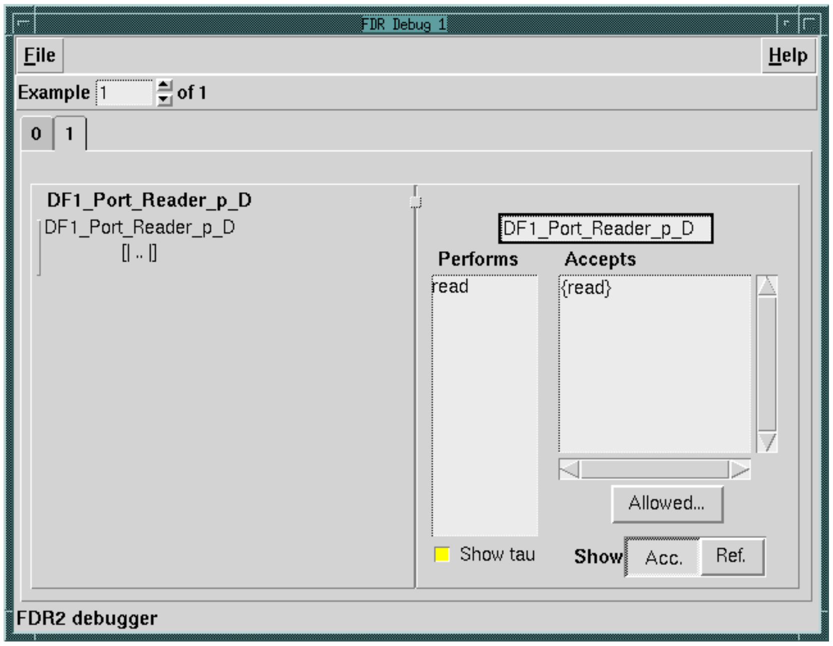
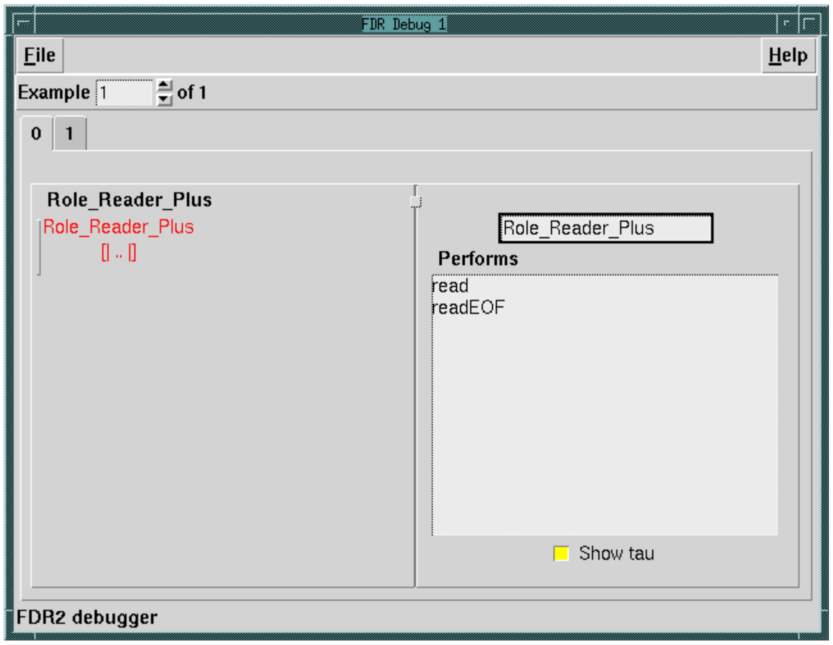


Figure 13: Debugging view of the refining process

[3] Often the refining and refined processes are referred to using the terms "implementation" and "specification" (respectively). We have avoided this usage as the term "specification" is rather overloaded.



Role_Reader_Plus

Role_Reader_Plus

[] .. []

Role_Reader_Plus

Performs

read

readEOF

Figure 14: Debugging view of refined process

Looking first at the refined process view in Figure 14, you'll notice that it only presents the trace of events that were performed. Looking at the refining process view, you'll see the familiar view of a trace and acceptance set. In this example, the problem is that the second event performed by the refined process, `readEOF`, is not an event that the refining process will accept. Thus, the refining process is not a valid refinement and so DataFile1’s File port and Pipe’s Reader role are not compatible.

This may seem confusing at first since the port of DataFile2 is compatible with the Reader role, but that role also does not accept `readEOF`. What's going on?

The Reader role permits an internal choice between closing and allowing an external choice between `read` and `readEOF`. DataFile2 is compatible with this because it always allows closing, thus always allowing one of the options of the internal choice. DataFile1, on the other hand, does not always allow closing. Thus to be compatible, it would always have to allow the other option of the internal choice -- allowing an external choice of `read` or `read-EOF`. Since it does not allow this option either (because it only allows `read`, not `readEOF`), it is not compatible with the Reader role.

To find out more

This concludes our brief demonstration of how Wright specifications are mechanically analyzed. For more information on Wright than what has been presented in the accompanying paper and this tutorial, visit the Wright website at [Wri98]. This website contains pointers to papers about Wright, tools for processing Wright descriptions, and a downloadable version of the files in this tutorial.

More information on FDR, including purchase information, can be found at the FDR website at [FDR98].

Please send any questions or comments to wright-support@cs.cmu.edu.

References

[AG97] Robert Allen and David Garlan. Revised version of *A Formal Basis for Architectural Connection*, which appeared in ACM Transactions on Software Engineering and Methodology, July 1997. The revised version is available at the Wright website.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, Shcool of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[FDR98] http://www.formal.demon.co.uk/FDR2.html

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[Wri98] http://www.cs.cmu.edu/~able/wright/index.html