Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 187 (2007) 19–34
www.elsevier.com/locate/entcs

FDR Explorer

Leo Freitas Jim Woodcock [1,2]

Department of Computer Science
University of York
YO10 5DD York, UK

Abstract

In this paper we describe the internal structures of FDR, the refinement model checker for Hoare's *Communicating Sequential Processes* (CSP), as well as an *Application Programming Interface* (API) allowing one to interact more closely with, and have fine grained control over, FDR's behaviour and data structures. With such information it is possible to create optimised CSP code to perform refinement checks that are more space/time efficient, hence enabling the analysis of more complex and data intensive specifications. This information is very valuable for both CSP users and tools that automatically generate CSP code, such as those related to security analysis generating test-cases as CSP processes. We also present a simple example of using the tool. Finally, we show how one can transform FDR's graph format into a graph notation (*e.g.*, JGraph), hence enabling visualisation of *Labelled Transition Systems* (LTS) of CSP specifications.

Keywords: refinement, model checking, CSP, FDR, Labelled Transition Systems, Application Programming Interface

1 Introduction

The soaring complexity in hardware and software systems has increased the demand for reliability and correctness, most noticeably in the high-integrity and safety-critical domains [1]. One effective way of achieving this goal is through the use of formal specification and verification. Nevertheless, no matter how useful those techniques might be, due to the sheer number of possible scenarios to investigate resulting from the concurrent interaction of different components, tool support is an imperative. For instance, the number of distinct behaviours to analyse can reach 10^7 distinct states on a parallel network of 10^{20} dinning philosophers and beyond [13].

In this paper, we are particularly interested in providing further tool support for refinement model checking the process algebra CSP [15] using its automatic tool FDR [7], hence we assume previous knowledge of both. CSP is a successful technology with industrial-strength tool support that has been used for two decades. In

[1] We are grateful to QinetiQ Malvern for their long-term support of our research group.
[2] Email: leo@cs.york.ac.uk, jim@cs.york.ac.uk

doi:10.1016/j.entcs.2006.08.042

this scenario, all observable behaviours are characterised by a *Labelled Transition System* (LTS) representing the (operational) semantics of CSP specifications [17, Chapter 4]. FDR is then used to exhaustively analyse those LTSs for various purposes, mainly refinement checking, determinism, and deadlock and livelock freedom. Due to the high-level of automation of model checking when compared with other formal verification techniques, such as theorem proving [2], the use of CSP and FDR became very attractive in both academia and industry.

FDR compiles two CSP specifications into suitable LTSs in order to check whether they from a refinement ordering ($S \sqsubseteq_m I$). These often are an abstract specification (S) of a system or some particular property of interest (represented as a CSP process), and the intermediate design or more concrete implementation (I) one wishes to check for refinement over a given model (m), which defines the granularity of detail. To perform a check, FDR exhaustively searches for pairs of mutually reachable states from each compiled LTS. That is, the states that one can reach by traversing both LTSs on the same trace, for all possible traces. More precisely, a pair is mutually reachable in the search if, and only if, they follow from the selection of a compatible transition on both LTSs with respect to the available visible events. An incompatible behaviour is characterised by the violation of at least one of the refinement criteria on the selected model. These criteria include information for each mutually reachable pair, such as traces, acceptable (or refused) behaviours, and divergence. Thus, if an incompatible pair is found, then the design (I) does not satisfy all the behaviours defined by the property or specification (S) on the given model (m), hence the proposed refinement does not hold and debugging information is available. The granularity on the chosen model (m) enables the verification of different aspects of systems, such as safety properties on the traces model ($\mathcal{T}$), nondeterminism on the failures model ($\mathcal{F}$), and divergent (or catastrophic/unpredictable) behaviour on the failures-divergences model ($\mathcal{FD}$). More details on (refinement) model checking can be found in [14,6].

Nevertheless, this kind of push-button technology, which enables automatic formal verification for correctness and refinement. This incurs quite some effort from the user in writing the appropriate CSP for FDR, which usually implies abstractions towards bounded models. The problem is there is no thorough and definite guidance source, to the extent of our knowledge. Instead, one needs to sift through many different (and unrelated) sources. Assuming the user has good knowledge of CSP, our tool can help the FDR user to generate more efficient CSP code, as well as to find the cause of some obscure execution errors, such as communication outside a channel data type. Perhaps it can be better exploited as bridge between FDR and more high-level tools which write CSP code.

A thorough list of references for FDR (version 2.82) are: the manual [7], model checking algorithms [14,12], and internal automata theory data structures [3]; additional information on FDR's transition system [13]; specialised deadlock checking tool that invokes FDR in the background [11]; a tool that compiles security protocol descriptions as optimised CSP [10]; a PhD thesis on FDR [17]; and a book that provides further insight on FDR's internal operation [16, Chapter 4]. Provided one

has fluency with FDR's CSP, FDR is capable of automatically analysing quite huge systems. Furthermore, if one explores the compositional properties of CSP operators, as an example in [13, p.198] shows, the possibility of reasoning about systems reaches the staggering figure of $7^{10^{1000}}$ states. This is not the actual number of states checked by FDR, but the total number of states of the combined system components. It means that checking a fraction of the state space is as good as checking the whole of it. By dealing with such a huge number of states, FDR is able to model not only hardware, but also quite complex software designs, as shown in [16].

Therefore, the main aim of the exploratory tool we present is to provide better guidance for the FDR user, and better integration with other external CSP tools. This was achieved through thorough study, experimentation, and reconstruction of available information from two sources in a sort of "software archaeology". These sources were FDR's manual [7, Appendix C], and the available source code of a deadlock checker tool that directly interacts with FDR [11]. As a result of such "archaeological" investigations, hidden information from FDR's LTS and debugging information was revealed. Consequently, complex specifications can be analysed in acceptable time scales, and improved operability for tools automatically generating CSP scripts is provided. This tool integration trend follows the principles set out in a grand challenge in computer research [1].

Our tool has been used in a great extent for the work presented in [4]. We also know it has been used in a project in Brazil for the development of specialised counter-example generation from automatically generated test-cases from various testing techniques encoded in CSP. Yet another project in Brazil transforms the output FDR *Explorer* into a specific format for the test-case generation tool TGV [20].

In the next Section, we describe the internal structures and object model of FDR, which includes FDR's API our tool interacts with. After that, Section 3 presents the functionalities we add by showing how it extends FDR's API. Next, in Section 4, we present a running example. Section 5 describes how to transform the underlying FDR LTSs into a visual graph format with graph visualisation tool support [8], and point some future directions on how to provide further integration for FDR with graph tools. Finally, we conclude the paper and point to some future directions in Section 6.

Related work.

There is little related work in this area. Brief information in FDR's manual [7, Appendix C] describes how one can use the Tcl/Tk script language [19] for direct interaction. These scripts mimic some of FDR's interfaces, hence allowing checks to be performed in the background or over the network. To the extent of our knowledge, there is only one tool that takes advantage of the scripting language FDR provides [11]. It provides specialised forms of deadlock and livelock freedom based on various strategies for recognition of patterns in graphs representing CSP specifications. This deadlock checking tool opens a connection with the FDR server and uses available Tcl/Tk scripts to compile CSP specifications and perform specialised checks, where debugging information is not processed. These scripts were the orig-

inal source of inspiration for our work. Furthermore, Valmari's approach [21] to exhaustively analysing LTSs in general could also be encoded/explored in FDR via Tcl/Tk scripts.

2 FDR's object model

In this section we explain FDR's architecture by describing its object model, which comprises LTSs, file management, refinement algorithms, and debugging.

2.1 FDR's architecture

FDR's architecture is divided into two layers. The top layer is either a *Graphical User Interface* (GUI) or a direct batch interface, where both are written using an object-oriented version of Tcl/Tk. The FDR server at the bottom layer is a Tcl/Tk interpreter written in `C++`. The interpreter has an object-model preloaded that provides: (i) parsing and compilation of machine readable CSP known as CSP_M; (ii) implementation of various refinement model checking algorithms; and (iii) thorough debugging information about refinement flaws.

From the GUI, the user loads a specification, adds/performs refinement checks, and investigates debugging information visually. From the batch interface, the user could perform the same operations, but with textual feedback logged to the standard output. The batch interface can be useful for noninteractive checks, or checks over a network.

These functionalities that the top layer interfaces implement are Tcl/Tk scripts arranged in such a way that the object-model methods in the underlying FDR server are called appropriately. That is, with the right number of parameters, in the right order, and at the right time. Therefore, by fiddling with these Tcl/Tk scripts (or creating new ones), it is possible to fine-tune FDR for: (i) detailed investigation of individual witnesses and behaviours at different points in the LTS; (ii) how to create more space-time efficient CSP specifications; and (iii) translation of FDR's LTS format into graph formats of available libraries for layout and visualisation. That is what our exploration tool does. It acts as another interface at the top of the batch interface, which allows extended control over available debugging information, as well as access to FDR's LTS, both not available on other two interfaces.

The most important of these three points is the insight provided on how to optimise CSP specifications. This is possible because FDR separates CSP operators into low- and high-level operators, according to the shape of the LTS they generate. High-level operators are all those that generate compositional (and modular) LTSs, which could be compressed with automata-theoretic techniques, such as bisimulation. The most common high-level operators are hiding, parallelism, and renaming. Low-level operators are all those that provide the core sequential language, and generate LTSs that are not very compression sensitive. The most common low-level operators are prefixing, choices, sequential composition, primitive processes, and recursion. Thus, the more high-level the process, the more amenable to compression the corresponding LTS will be. Obviously, it is not always possible to provide the

most compact LTS due to the structure of the process being described. Nevertheless, bearing such structuring in mind proves very useful when checking complex or data intensive specifications [16,10]. This information might be interesting not only for the experienced CSP user handling complex specifications, but also for other tools that automatically generate CSP code, such as security analysis tools that use CSP for test-case generation [10,18]. By inspecting the object-model methods that are hidden in both top level interfaces available, we are able to tell exactly how, and under which circumstances, one can improve the compactness or efficiency of compiled CSP LTSs.

2.2 Available functionality

The object-model provides four main functionalities: (i) session management representing specification sources; (ii) *Interpreted State Machines* (ISMs) representing compiled LTSs with embedded refinement check algorithms; (iii) hypothesis objects allowing the check of refinement claims on ISMs; and (iv) debugging objects enabling precise interpretation of a failed refinement check. Apart from the brief explanation in [7, Appendix C], the object-model details are undocumented, as far as we know.

Session management.

It allows one to administer (a set of) loaded specification sources for a refinement check sessions. It implements two functionalities: (i) script management; and (ii) script evaluation. Script management allows file loading, and selective display of various kinds of information within a specification, such as the loaded CSP processes and channels, the assertions about refinement claims and property checks, the list of expressions used throughout the specification script, and so on. Once the specification has been loaded, script evaluation becomes the entry point for FDR's refinement algorithms and LTS. It enables the compilation of CSP processes as LTSs, as well as evaluation of mathematical expressions and refinement assertions representing property checks.

Interpreted state machines (ISM).

They represent a compiled state machine, and are the core functionality of FDR: refinement checks of LTSs compiled via a session object, usually from a CSP specification. That is, the underlying FDR server is generic enough to represent and model check not only CSP, but a particular category of LTSs. Obviously, the operational semantics of CSP fits into this category. Each ISM implements three functionalities: (i) LTS description; (ii) LTS structure; and (iii) LTS analysis. The LTS description is a database containing the process name, its original ASCII script, and the calculated alphabet of events used by the process it describes. More interesting is the LTS structure, which contains a detailed characterisation of the LTS, such as the refinement search root node, the initial events of each LTS node representing outgoing transitions, the next nodes reached through particular events, (minimal)

acceptances and divergence calculations for each node used during some refinement checks, advanced information about LTS compression, the way various CSP operators are treated as low-level or high-level, and so on. Finally, with LTS analysis one can select from the various embedded algorithms, such as refinement checking, deadlock and livelock freedom, or determinism characterisation of specifications.

Hypothesis objects.

Once one model checking algorithm has been selected for a compiled ISM, the FDR server returns a hypothesis object. It represents an assertion about an ISM. A hypothesis object generates debugging information (or success reports) allowing the investigation of the cause of a refinement failure (or successful check). It also contains simple state defining whether the check has been performed or not, what the checking status is, and which parts of the LTS structure will affect the check.

Debugging information.

It has detailed descriptions of witness(es) for a refinement failure. This information is separated in three functionalities: (i) debug context; (ii) debug tree; and (iii) behaviour of LTS nodes and their children. A debug context is the result of testing the assertion a hypothesis object represents, and is present in the FDR GUI as a separate debugging window. It contains three kinds of information: (i) participant processes; (ii) debug trees of each participant; and (iii) witness(es) containing the flawed behaviour of each participant. A debug tree represents the LTS of the flaw (or correct) process together with its characteristic behaviour. It is represented in the FDR GUI as tree views of the participant processes. Although debug contexts can represent successful checks, behaviour objects are always related to refinement failures, and they contain detailed information about the acceptances (and refusals) of a particular LTS after some trace has taken place. They represent the allowed behaviours a debug tree characterises. This appears in the FDR GUI as small windows with contrasting information regarding acceptances (or refusals) at particular debug tree nodes. For successful checks, no debugging information is available to the user.

2.3 FDR's API

Knowing FDR's API can be very useful for understanding the information contained in debugging witnesses, or how LTSs are composed (see Section 3).

As the complete FDR's object-model has not been publicly documented before, we provide it in the UML class diagram of Figure 1. In what follows we explain the most relevant methods for each of these classes.

`Session`

load(Str, Str): loads a CSP script from the given directory and file names.

compile(Str,M): compiles the given process on the chosen semantic model.

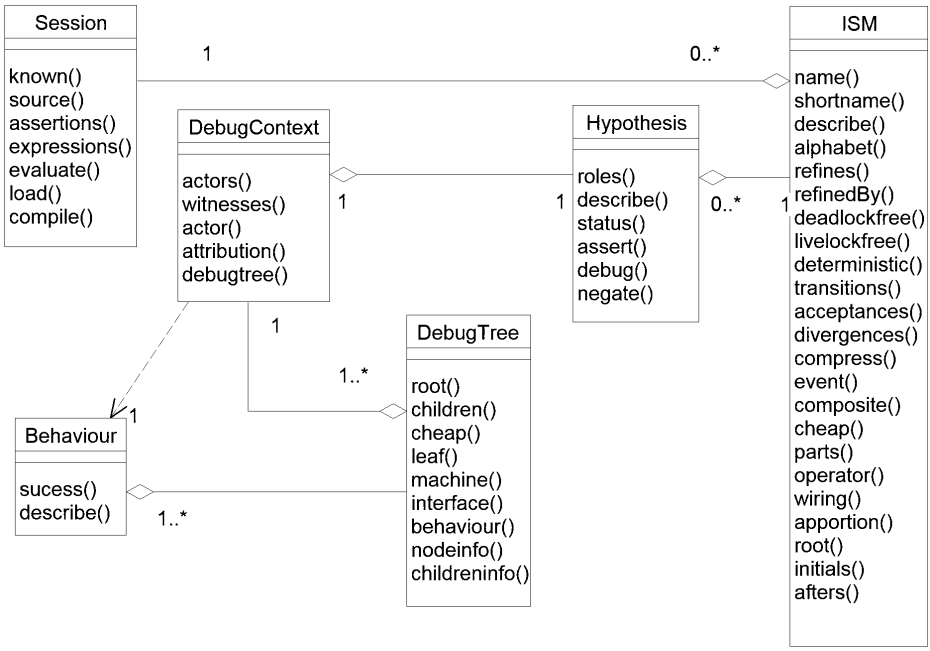

Fig. 1. UML class diagram of FDR object model

For returning an `ISM` object, the **compile** method requires that process names must include actual parameters for every formal parameter declared. The available models are "`-t`" for traces ($\mathcal{T}$), "`-f`" for failures ($\mathcal{F}$), and "`-fd`" for failures-divergences ($\mathcal{FD}$). It returns an `ISM` object.

`ISM`**.**

It represents a compiled CSP process as an LTS. They are based on the automata theory described in [3]. Apart from the trivial methods about the textual script this object represents, there are three sets of methods related to structure [17, Chapters 4, 8], algorithms [14,12], and compression [13]. The structural methods are given below:

root: returns the node index where the search starts.

alphabet: returns a list of full event names of the process (with channel name and values) represented by this ISM.

event(int): returns the corresponding name of an event from the process alphabet at the given index. It returns an empty string if the index is greater than the size of elements. Every process has two predefined events in its alphabet for internal communication (as "_tau" at index 0), and successful termination (as "_tick" at index 1).

transitions: returns the LTS transitions. Each transition is represented by three numbers between braces (*e.g.*, {`1 0 3`}), where the source node index (1) reaches the target node index (3) through an event number (0) placed between the two node indexes. This event number can be used in the `event` method to retrieve

the corresponding element from the process alphabet.

divergences: returns the divergences of each node index as a list of boolean values. Thus, if the LTS contains four nodes, all of which are not divergent, the method returns a list of 4 boolean values set to *false* (or 0). For checks outside the failures-divergences ($\mathcal{FD}$) model, this list is not calculated and every node index is assumed as not divergent.

acceptances: returns the (minimal) acceptances set for each node index mentioned in transitions. Each acceptance set looks like "{{4} {3}} {}", which means that the node index 0 has two acceptances sets, one containing the event labelled 4, and another labelled 3. Moreover, as node index 1 acceptances set is empty, it represents internal transitions taking place. By inspecting the corresponding divergence information for node 1, one can check whether this internal transition causes divergence or not. Finally, acceptances sets containing the empty set (*i.e.*, {{}}) represent deadlocked nodes. Once more, the `event` method can be used here to retrieve the event name.

initials(int): returns a list of alphabet elements immediately available for a given node index.

afters(int, Str): returns a list of target node indexes reached from the source node index through the `event` name in the process alphabet. As LTSs are not complete, this is a partial method because not every node has transitions through every event. Thus, in such (partial) cases, the method returns an empty set of nodes. The same empty result is returned for terminal nodes as well, such as those representing deadlock (*e.g.*, `STOP`), or successful termination (*e.g.*, `SKIP`).

With these methods, one is able to understand how FDR represents CSP processes as LTSs. This can be useful for finding adequate CSP specification patterns for FDR, as well as to understand obscure issues of the operational semantics. For instance, in Section 4, we show how the LTS of some peculiar processes are structured. Next, there are the methods enabling the user to choose a refinement algorithm to perform. They create `Hypothesis` objects as assertions:

refines(Str,M) : `Hypothesis("Str [M= this").`

refinedBy(Str,M) : `Hypothesis("this [M= Str").`

deadlockfree(M) : `Hypothesis("deadlock-free[M] this").`

livelockfree(M) : `Hypothesis("livelock-free[M] this").`

deterministic(M) : `Hypothesis("deterministic[M] this").`

Finally, there are more advanced methods, which describe how FDR encodes low- and high-level processes, as well as entry points for compression techniques described in [7, p.60] and [13].

cheap: returns whether the root node corresponds to a low- or high-level process. As high-level processes are easier to decompose, hence explore the modular structure of CSP operators, this flag can be useful to adjust processes appropriately.

This is particularly useful when handling complex or data intensive specifications.

operator: returns the high-level operator name used as the composite represented by the root node, for instance, hiding or parallelism.

wiring: returns a set of `event` numbers associated with the composite `operator` in a `cheap` ISM, such as those of a hiding set, or a synchronisation set.

parts: returns a list of subcomponent ISMs of the composite ISM the root node index might represent.

`DebugContext`

actors: returns the number of nodes involved in a witness.

witnesses: returns the number of witnesses found.

actor(int): returns the process name of an actor index, which is bound by the number of involved actors and points to a node index.

attribution(int,int): returns the behaviour object of the chosen witness at the selected actor index.

debugtree: returns a debug tree containing behaviour information about a refinement failure.

Debug contexts also contain the behaviour of the search root node, together with the involved process names and found witnesses. They describe FDR's graphical interface debugging window.

`Behaviour`

Instances of this class represent the most detailed level of debugging information available. They contain information about acceptances (or refusals) after some trace has taken place.

3 FDR *Explorer*

With this knowledge of how FDR works, we built an extended API as Tcl/Tk script files to be loaded in FDR's server. At the moment, we have a user friendly output that is just plain text, and a translation from FDR LTS transitions to a graph notation language with graph visualisation support [8]. The former is intended for FDR's users or automatic CSP code generating tools, whereas the latter is to be passed to a graph visualisation tool, as explained in Section 5.

3.1 Available functionality

The API we devised contains methods divided in five categories: (i) process inspection; (ii) process compilation; (iii) information extraction; (iv) helper methods; and (v) auxiliary methods.

The inspection methods are the main methods one usually calls at the beginning of a refinement session. Firstly, the current session details, such as known processes and assertions, is logged. Next, a given list of processes (with actual parameters if needed) is compiled into ISMs, and detailed information about their structure are logged. After that, three hypotheses for determinism, and deadlock and livelock freedom are automatically generated and checked. Finally, if these hypotheses are *false*, then information about the debug context they contain is logged. This includes not only the debugging context, but also all behaviours and debug trees in case of a refinement failure. If the script contains assertions about refinement checks, or if the user wants to perform any specific refinement, then the created hypothesis objects can be inspected in the same way. Alternatively, if the script has no parameterised process that demands actual parameters instantiation, no process list is needed and all processes from the current session are inspected automatically.

The compilation methods can be used to generate ISMs in a chosen CSP model for a list of processes one wants to inspect. They are called by the inspection methods, but the user (or another tool) could use them to perform specific compilation tasks.

The extraction methods log information from all Tcl/Tk methods available for each class from the FDR object-model (see Figure 1). That is, the LTS structure of ISM objects, the debug context (if any) of the three automatically generated hypothesis objects mentioned above, the debug trees of each debug context, and the corresponding behaviours detailing the reason of failures.

The helper methods provide messages for every method of every class in FDR's object model of Figure 1. As this can be quite verbose, the default output is just for `ISM`, `Hypothesis`, and `DebugContext` objects. Nevertheless, the user (or another tool) could selectively call helper methods for available FDR classes.

Finally, the auxiliary methods provide help on how to use the FDR explorer API itself, file loading, garbage collection of Tcl/Tk objects, and a main method that hooks the tool into the FDR server.

3.2 FDR Explorer API

We detail below the most relevant methods for each category mentioned above.

Inspection methods

inspectProcs(File,List⟨Str⟩,Bool): inspects the given list of processes from a file handle pointing to the CSP script file. The flag indicates whether the FDR objects created should be deleted or not. For batch execution, such as those done by tools, it should be set to *true* (or 1). For execution where further inspection is required, it could be set to *false* (or 0).

inspectParameterless(File,Bool): inspects all processes from the given file. It generates an error if any of the processes have parameters.

The results are logged into separate files for each process in the given list. Thus, if a file named `spec.csp` and a list of processes {`P(0)` `Q`} are given, two files named

`spec.P(0).exp` and `spec.Q.exp` are created, each containing the corresponding process inspection. Moreover, if such files already exist, they are truncated (*i.e.*, cleared) before being written.

Compilation methods

compileProcInModel(File,Session,Str,M): compiles the given process in the given model from a FDR `Session` object printing the results on the given file handle. Parameterised processes must be instantiated, otherwise the FDR server crashes. It returns the created ISM object to the user.

compileProcs(File,Session,List⟨Str⟩): compiles the list of processes in the failures-divergences model.

Extraction methods.

There is one extraction method for each class in the diagram of Figure 1. They log the result of calling each method of the corresponding FDR class into the given output file. They receive the file handle for logging the output, and the corresponding FDR object. Moreover, there are some additional methods for `ISM` objects used to create default hypotheses about determinism, and deadlock and livelock freedom.

Helper methods.

Similarly, there is one helper method defined for each FDR class as well. They log a textual description of the role each method of each FDR class has. We also group commonly used helper methods together.

Auxiliary methods

FDRExplorerHelp: provides a description about the FDR *Explorer* API.

load(Str): creates a FDR `Session` object and loads the given full file name.

deleteAll(File,List): releases the allocated memory from the objects in the given list logging the results into the given file.

Finally, to integrate our tool into FDR, we hook the FDR *Explorer* API scripts into FDR's Tcl/Tk `main` method (see Section 4.1).

4 Running example

To illustrate the use of our tool, we provide a series of examples in the distribution, as well as some extra help files and the complete class diagram model[5]. This includes examples with both low- and high-level operators, as well as an industrial-scale example from [9].

In this paper, we have chosen a simple example to show how the knowledge of FDR's LTS structures can give insight into how FDR works. The purpose is to

understand how FDR encodes the presence of termination (`SKIP`) in an external choice. In [15, p.141], laws about this situation are given as

$$P \,\Box\, SKIP = P \triangleright SKIP = (SKIP \sqcap (P \,\Box\, SKIP)) = ((P \sqcap STOP) \,\Box\, SKIP)$$

But how does FDR encode these processes? To find out, let us load the machine readable version of these CSP processes from the `skip.csp` file in the tool distribution package:

```
channel c
P = c -> P
A = P [] SKIP
B = P [> SKIP
C = SKIP |~| (P [] SKIP)
D = (P |~| STOP) [] SKIP
```

The result is given in Table 1. It is clear that FDR's encoding of $(P \,\Box\, SKIP)$ is more efficient than the other versions, as the number of nodes and transitions are different in each representation. Process A has 3 transitions and 3 nodes, processes B and C have the same LTS with 6 transitions and 5 nodes, and process D has the worst LTS with 7 transitions and 5 nodes. As the alphabet of these processes are the same, so are the results from `event`. In process A, node 0 reaches node 1 through event 1 (_tick) with one acceptances set containing events 1 and 2 (_tick c), where all nodes are divergence-free. The other transitions can be interpreted similarly.

To investigate the matter further, we added assertions checking how these different representations relate to each other. Processes B, C, and D are equivalent in the failures-divergences model as they refine each other. Process A refines both processes B, C, and D, but is refined by neither of them. This shows the rationale for the most space-efficient encoding of termination (`SKIP`) on external choices to use in FDR.

Furthermore, as we execute by default a deadlock freedom check, let us inspect the output for process D. The hypothesis object contains one witness with one debug tree and the following behaviour:

Performs: ⟨_tick⟩

Accepts : {} **Could accept**: {_tick} {c}

Refuses : {_tick c} **Could refuse** : {c} {_tick}

That means, after performing the event _tick, D is refusing both _tick and c, whereas the specification for deadlock freedom (see *DF* in [15, p.375]) could refuse either event but not both. From FDR's debugging window, this can be viewed by pressing the button labelled Acc. (or Ref.).

4.1 Calling the APIs

Now, let us show how one could perform the check explained above using the FDR *Explorer* API. Assuming that the directory `$FDRHOME/bin` is in the Unix `$PATH`,

LTS for process A

alphabet	=	_tick c	`event(0)`= _tau
transitions	=	{0 1 1} {0 2 2} {2 2 2}	`event(1)`= _tick
acceptances	=	{{1 2}} {{}} {{2}}	`event(2)`= c
divergences	=	0 0 0	

LTS for process B

alphabet	=	_tick c
transitions	=	{0 0 1} {0 0 2} {1 1 3} {1 2 4} {2 1 3} {4 2 4}
acceptances	=	{} {{1 2}} {{1}} {{}} {{2}}
divergences	=	0 0 0 0 0

LTS for process C

alphabet	=	_tick c
transitions	=	{0 0 1} {0 0 2} {1 1 3} {1 2 4} {2 1 3} {4 2 4}
acceptances	=	{} {{1 2}} {{1}} {{}} {{2}}
divergences	=	0 0 0 0 0

LTS for process D

alphabet	=	_tick c
transitions	=	{0 0 1} {0 0 2} {0 1 3} {1 1 3}
		{1 2 4} {2 1 3} {4 2 4}
acceptances	=	{} {{1 2}} {{1}} {{}} {{2}}
divergences	=	0 0 0 0 0

Table 1
FDR LTS for processes `A`, `B`, `C`, and `D` from file `skip.csp`

and that `FDRExploerer.tcl` is in `$FDRHOME/lib`, we start the FDR server with the following command from the shell prompt:

```
venice$ fdr2tix -insecure -nowindow
```

Now the FDR server is running, we load the FDR *Explorer* interface using

```
% source lib/FDRExplorer.tcl
```

After that, the extended API is available and we can start inspecting processes.

Assuming the file `skip.csp` is in the current directory, we could ask for the LTS structures of all 4 processes without deleting the generated objects using

```
% set lprocs {A B C D}
% inspectProcs ‘‘./skip.csp’’ $lprocs 0
```

As no process in the file has parameters, we could also use the command

```
% inspectParameterless ‘‘./skip.csp’’ 0
```

As a result of executing the inspection methods, four files are created and named `skipX.csp.exp`, where `X` will be either `A, B, C,` or `D`. They contain detailed information about each process ISM, as well as the 3 default hypothesis about determinism, and deadlock and livelock freedom already checked. If any of those checks fail, additional information about debug contexts, debug trees, and behaviours are also logged. Finally, if one wants to perform operations over the FDR objects returned, it can be done directly by manually calling methods. The object name to use is the one FDR returns. Thus, we could type a command, such as

```
% session__1 compile D -t
```

if we wanted FDR to recompile process `D` on the traces model on the current session. This will result in a new `ISM` that one can call any of the other available methods in a similar way.

5 Graph visualisation

At the moment, the translation strategy from FDR LTS transitions to an available graph format [8] is indeed very simple. For every transition from FDR's LTS, we swap the event and target node indexes, so that we have source and target node indexes followed by the event number, instead of the format presented in Table 1, where the event number is between the node indexes. The purpose of this extension to our tool is to prove the concept that one can visualise CSP LTSs and debugging information generated by FDR, hence improving the user friendliness for CSP related tools. To perform such operation, one needs to load the script from the `fdr2jgraph.tcl` file into the FDR server, and call the `fdr2jgraph` method passing the input CSP file name, the process name and an output file to generate the JGraph compatible code.

An open possibility is to develop a more thorough translation strategy taking advantage of the various features many graph visualisation libraries have. For instance, graph layout algorithms for rendering big LTSs, or annotations support on nodes for inclusion of acceptances and divergence information.

Another interesting possibility would be to manipulate the generated graph, trying to find specific patterns, hence allowing a deeper understanding of the behaviour of the represented machine, or suggestions for further compression that FDR

couldn't foresee. Later on, with such information, one could try adjusting/adapting the original CSP script in order to perform quicker and smoother checks. Such a strategy of fiddling with the CSP script has been successfully achieved through different CSP specification patterns via the CASPER tool [10], which translates security protocol notation into highly optimised CSP code. These investigations could also serve as the basis for a diagrammatic tool that would formally represent the CSP semantics, hence allowing users to formally draw concurrent processes!

6 Conclusion

In this paper we present a new interface to the CSP [15] refinement model checker FDR [7], which extends one of the available user interface APIs. It allows extended control over debugging information, as well as investigation of hidden features of the LTS data structure used to represent compiled CSP specifications for refinement model checking. With this tool it was possible to carefully study the operational semantics of CSP, hence develop an operational semantics for a concurrent language similar to CSP [4]. It has also been used by other people in test case generation using CSP and FDR, and Java code generation tools for this new concurrent language.

The main contribution of the FDR *Explorer* API is that it enables better integration between CSP script generation tools [18], as well as improved information to the user. This appears as the ability to investigate witness information at different points of the LTS, or reasoning about more space-efficient representations of CSP processes. These functionalities are not available from the original FDR interfaces. This also follows the trend of tool integration set out by one of the UK *Grand Challenges in Computer Research* [1].

We also show an example of running the tool for finding out how FDR represents the presence of termination (`SKIP`) in an external choice, which we found quite illuminating.

Finally, we explain how we transformed the available CSP LTS transitions into a graph notation format with visualisation tool support [8]. This is the first step towards integration with a visualisation tool for CSP. Going further, one could provide the translation the other way round, hence enabling drawing graphs that would formally represent CSP specifications and could be directly passed to FDR for refinement checks.

As future work, we envisage to provide an object-oriented version of the Tcl/Tk script. This would enable to provide integration of the FDR *Explorer* API into FDR's GUI. Another interesting idea is to provide support from graph notation formats back to FDR LTS format. This would enable one to characterise general graphs (or graph patterns) as interesting CSP processes amenable for refinement checking. Furthermore, with such a bi-directional link with graph notations, it might be possible to integrate FDR with clever set-theoretic compression techniques over LTS structures, as presented by Valmari in [21].

References

[1] J. C. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The Verified Software Repository: a Step Towards the Verifying Compiler. UK Grand Challenge for Computer Research, Steering Committee, 2004.

[2] Edmund M. Clarke and Jeannette M. Wing. Formal Methods—State of the Art and Future Directions. *ACM Computer Surveys*, 28(4):626–643, December 1996.

[3] Rance Cleaveland and Matthew Hennessy. *Testing Equivalence as a Bisimulation Equivalence*. FACJ, 5(1):1–20. Springer-Verlag, 1993.

[4] Leonardo Freitas. *Model Checking Circus*. PhD thesis, University of York, October 2005.

[5] Leo Freitas. FDR *Explorer* v0.3. http://www.cs.york.ac.uk/~leo

[6] Leo Freitas, Jim Woodcock, and Ana Cavalcanti. *State-rich model checking. Innovations in Software Engineering—a NASA Journal*, 2(1):49–64, March 2006.

[7] Michael Goldsmith. *FDR2 User's Manual version 2.82*. Formal Systems (Europe) Ltd., June 2005.

[8] JGraph User's Manual, 2006. http://www.jgraph.com/pub/jgraphmanual.pdf.

[9] Jonathan Lawrence. Practical Application of CSP and FDR to Software Design. In Ali E. Abdallah, Cliff Jones, and Jeff Sanders, editors, *25 Years of CSP*. FACJ, 2004.

[10] Gave Lowe. *CASPER User Manual*. Oxford University, 1997.

[11] J. M. R. Martin. *The Design and Construction of Deadlock-free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.

[12] J. M. R. Martin and Y. Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. *Communicating Process Architectures*, 2000.

[13] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. *First TACAS in LNCS Springer-Verlag*, 1019(1), 1995.

[14] A. W. Roscoe. *Model Checking CSP* in *A Classical Mind: Essays in Honour of C. A. R. Hoare*. International Series in Computer Science. Prentice-Hall, 1994. Chapter 21, pages 353–378.

[15] A. W. Roscoe. *The Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1997.

[16] Peter Ryan, Steve Schneider, Bill Roscoe, Michael Goldsmith, and Gave Lowe. *Modelling and Analysis of Security Protocols*. Addison Wesley, 2001.

[17] J. B. Scattergood. *The Semantics and Implementation of Machine Readable CSP*. PhD thesis, Oxford University, The Queen's College, 1998.

[18] T. Srivatanakul. *Security Analysis with Deviational Techniques*. PhD thesis, University of York, 2005.

[19] Tcl/Tk: Tool Command Language. http://www.tcl.tk/, 2006.

[20] The test sequence generator TGV. http://www-verimag.imag.fr/~async/TGV

[21] A. Valmari. *A stubborn attack on state explosion* in Proceedings of 2^{nd} International Conference in Computer-Aided Verification. LNCS 531, pages 156–165. Springer-Verlag, 1990.