FDR Explorer

Leo Freitas, Jim Woodcock

▶ To cite this version:

Leo Freitas, Jim Woodcock. FDR Explorer. Formal Aspects of Computing, 2008, 21 (1-2), pp.133-154. 10.1007/s00165-008-0074-7 . hal-00477905

HAL Id: hal-00477905
https://hal.science/hal-00477905

Submitted on 30 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOI 10.1007/s00165-008-0074-7
BCS © 2008
Formal Aspects of Computing (2009) 21: 133–154

Formal Aspects of Computing

FDR Explorer

Leo Freitas and Jim Woodcock

Department of Computer Science, University of York, York YO10 5DD, UK.
E-mail: leo@cs.york.ac.uk

Abstract. We describe: (1) the internal structures of FDR, the refinement model checker for Hoare's Communicating Sequential Processes (CSP); and (2) an application-programming interface (API) that allows users to interact more closely with FDR and to have finer-grain control over its behaviour and data structures. This API makes it possible to create optimised CSP code to perform refinement checks that are more space or time efficient, enabling the analysis of more complex and data-intensive specifications. The API can be used either by those constructing CSP models or by tools that automatically generate CSP code. We present examples of using our tool, including handling advanced FDR features such as transparent functions, which compress state spaces before checking. We also show how to transform FDR's graph format into a graph notation such as JGraph, enabling visualisation of labelled transition systems of CSP specifications.

Keywords: Refinement; Model checking; CSP; FDR; Labelled transition systems; Automata

1. Introduction

Complexity in hardware and software systems has increased the demand for reliability and correctness, most noticeably in high-integrity and safety-critical domains [BHW06]. One effective way of achieving required levels of assurance is by using formal methods of specification and verification, but the complexity arising from the concurrent interaction of different components means that tool support is imperative. Model checking tools have been shown to be useful, but the number of distinct behaviours in the finite model of a parallel system can easily reach an astonishing size making its analysis infeasible.

Our research objective is to enhance FDR [Gol05], the tool that automates refinement model checking for CSP [Ros97], and we assume in this paper previous knowledge of both CSP and FDR. FDR stands for *Failures Divergences Refinement*; it is a trademark and a commercial product of formal systems (Europe) Ltd, and it has been used successfully in both industry and academia for two decades. Provided the user has fluency with CSP, the FDR model checker is capable of automatically analysing quite huge systems. FDR's developers report using FDR on a system with the staggering figure of $7^{10^{1,000}}$ states [RGG95, p. 198]. Of course, this is not the actual number of states checked by FDR, neither is this a limit of the technology, but rather the total number of states of the combined system components. FDR was used in a compositional way to analyse this example, and checking a fraction of the state space was as good as checking the whole of it. A question usually arises in connection with this example: How easy is it to repeat this success if you are not one of FDR's developers?

Although model checking is usually described as push-button technology, the user needs to write the appropriate CSP for FDR, which usually requires creating abstractions to suitably bounded models. But most users are faced with the problem even in the various sources on FDR and CSP_M, such as [Ros97, Sca98, RSR01, Ros94, RGG95, MaH00], that there is no thorough and definite guidance on how best to transform models to tackle

Correspondence and offprint requests to: L. Freitas, E-mail: leo@cs.york.ac.uk

the state explosion problem effectively. Instead, the scholarly user has to sift through diverse and often unrelated sources of information to see how others have been successful. *FDR Explorer* can help the CSP-lterate 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.

To a first approximation, observable behaviours are characterised in FDR by labelled transition systems (LTSs) that represent the operational semantics of CSP specifications in the machine-readable version of CSP accepted by FDR, CSP_M [Sca98, Chap. 4]. In fact, FDR's data structure is far more complicated than a simple LTS. The concrete representation is a specialised bit-masked B-tree with highly optimised memory and time complexities. The best place to get more insight on the description of the abstract LTS implemented by the B-trees is in [ClH93]. Unfortunately, there is no published material on the implementation details.

FDR compiles two CSP processes into suitable LTSs in order to check whether one is a refinement of the other. These processes are usually an abstract specification S of a system or some particular property of interest, and a more concrete implementation I one wishes to check for refinement over a given semantic model m. We write this check as $S \sqsubseteq_m I$. FDR exhaustively searches for pairs of mutually reachable states from each compiled LTS. That is, it searches for the states reachable by traversing both LTSs on the same trace, for all possible traces of the implementation. So a pair is mutually reachable in the search if, and only if, it follows 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 has a behaviour not 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 of m enables the verification of different aspects of systems, such as safety properties in the traces model $\mathcal{T}$, nondeterminism in the failures model $\mathcal{F}$, and divergence in the failures-divergences model $\mathcal{FD}$. More details on refinement model checking can be found in [Ros94, FWC06]. When we mention the comparison of two LTSs for mutually reachable paths, this is an abstraction of the actual corresponding operation on B-trees. As those B-trees are hidden away and not available to the FDR user, it is enough to think of them abstractly as the GLTSs from [ClH93]. Directly comparing LTSs in the state space where FDR works would be computationally infeasible [RSR01, p. 135], and this shows the power and industrial strength of FDR as a tool.

FDR Explorer was developed by study, experimention, and reconstruction of available information: an exercise in software archaeology. The sources were FDR's manual [Gol05, Appendix C] and the available source code of a deadlock checker tool that directly interacts with FDR [Mar96]. As a result of archaeological investigations, hidden information from FDR's LTS and debugging information were revealed.

The lack of publication of crucial implementation details seems to be a tendency in our field. For instance, powerful indexing technology for theorem proving remains unpublished, and indeed is deemed unpublishable, as pointed out in [McC92, p. 2], and so the wheel gets reinvented by different tool-builders. With some detective work, we found the link with B-trees mentioned by one of FDR's author in [RSR01, p. 129]. The abstraction with LTSs, or rather a generalised LTS (GLTS), is also confirmed in [RSR01, Sect. 4.3], where a further pointer to [RGG95] is given on page 138. Following that pointer to, we find a citation to [Ros94] and [ClH93] on page 190. In [Ros94], an early, clear explanation of how FDR works is given. In [ClH93], some formal details of GLTSs are given, and we find that the best published source of (abstractly presented) information on what these internal, optimised B-trees are representing.

We use FDR version 2.82, and we list the most authoritative references for the background to our construction of *FDR Explorer*: the manual [Gol05], model checking algorithms [Ros94, MaH00], and internal automata theory data structures [ClH93]; additional information on FDR's transition system [RGG95]; a specialised deadlock-checking tool that invokes FDR in the background [Mar96]; a tool that compiles security protocol descriptions as optimised CSP [Low97]; a Ph.D. thesis on FDR [Sca98]; and a book that provides further insight on FDR's internal operation [RSR01, Chap. 4].

Perhaps *FDR Explorer* can be better exploited as a bridge between FDR and more high-level tools that write CSP code, such as [Mar96, Low97, TGV, Sri05]. An interesting example of this sort of tool is the one used by Motorola [CaS06]. It goes from informal requirements documents, where the input language is a restricted version of English, down to CSP_M descriptions of user and component views of the corresponding requirements; FDR is then used to verify refinement claims. *FDR Explorer* is used to discover how best to represent states in the CSP object code to make FDR's analysis feasible. Our tool has been used in a great extent for the work presented in [Fre05]. 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 *FDR Explorer*'s output into a specific format for the test-case generation tool TGV [TGV].

There is little related work in this area. Brief information in FDR's manual [Gol05, Appendix C] describes how one can use the Tcl/Tk script language [TCLTK] for direct interaction. These scripts mimic some of FDR's interfaces, hence allowing checks to be performed in batch mode or over the network. To the extent of our knowledge, there is only one tool that takes advantage of the scripting language FDR provides [Mar96]. It enables specialised checks for 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 original source of inspiration for our work. Furthermore, Valmari's approach [Val90] to exhaustively analysing LTSs in general could also be encoded/explored in FDR via Tcl/Tk scripts.

In the next section, we present a motivating example that is simple yet quite interesting. Next, in Sect. 3 we describe FDR's architecture, its internal structures, and object model API that our tool interacts with. After that, Sect. 4 presents the functionalities we add by showing how it extends FDR's API. In Sect. 5, we present a brief overview of available compression techniques together with an example where *FDR Explorer* can be (somehow) naively used yet achieving great (compression) results. Section 6 describes how to transform the underlying FDR LTSs into a visual graph format with graph visualisation tool support [JGRAPH], and suggest how to provide further integration for FDR with graph tools. Finally, we conclude the paper and point to some future directions in Sect. 7.

2. Motivating example

To motivate the use of our tool, we provide a series of examples in the *FDR Explorer* distribution, as well as some extra help files and the complete class diagram model [Fre07]. The distribution includes a variety of examples from simple use of various CSP operators, to an industrial-scale example from [Law04], as well as examples on compression techniques mentioned in [Gol05, Chap. 5]. We have chosen a simple example to illustrate 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 [Ros97, p. 141], laws about this situation are given as

$$P \,\square\, \text{SKIP} = P \rhd \text{SKIP} = (\text{SKIP} \sqcap (P \,\square\, \text{SKIP})) = ((P \sqcap \text{STOP}) \,\square\, \text{SKIP})$$

How does FDR encode these processes? We load the specification to find out.

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

The result of inspecting processes `A` to `D` is given in Table 1. FDR's encoding of ($P \,\square\,$ 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` method on each event index. The `event` method returns for a given event index within the known alphabet its textual representation from the CSP script. In process A, node 0 reaches node 1 through event 1 (_tick) with one acceptances set containing events {1 2} (i.e., {_tick c}), where all nodes are divergence-free (0). Nodes with an empty set as one of its acceptances sets (i.e., { {} }) are either deadlocked (i.e., `STOP`) or have successfully terminated (i.e., `SKIP`). Every process `alphabet` contains "_tau" (τ) and "_tick" ($\surd$) representing internal communication and successful termination, respectively. Also, divergent nodes have empty acceptances

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

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			

sets (i.e., { }). The other processes can be interpreted similarly. To investigate the matter further, we have checked all combinations of refinement checks among these processes, where the ones that fail are

```
-- assert A[FD=B    assert A[FD=C    assert A[FD=D    fails!
```

Processes *B*, *C*, and *D* are equivalent in the failures-divergences model as they refine each other. Process *A* refines each of the processes *B*, *C*, and *D*, but is refined by none of them. That is because FDR uses Hoare's termination semantics in external choice, rather than Roscoe's [Ros97]. This shows the rationale for the most space-efficient encoding when termination (SKIP) occurs in an external choice.

One major change in the current version of *FDR Explorer* is the (recursive) computation of parts of composed processes, rather than just the **root**. This is useful in summarising the quality of the LTS produced in terms of space/memory needed for storage, as well as the amount of time required for its computation. We also included additional information about: (i) the number of nodes and transitions each LTS has; (ii) the number of transitions each node contributes; (iii) the time it takes to compile the indexed state machines (ISM); and so on.

3. 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.

3.1. 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 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, checks over a network, or external tools directly interacting with FDR [Mar96, Low97].

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, the FDR server methods are invoked 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), we fine-tune FDR for: (i) providing detailed investigation of individual witnesses and behaviours at different points in the LTS; (ii) creating more space-time efficient CSP specifications; and (iii) translating FDR's LTS format into graph formats of available libraries for layout and visualisation of processes, LTSs and debugging results. 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 structure not available on the other two interfaces.

The most important of these three points our tool covers is the insight provided on how to optimise CSP specifications. This is possible because FDR compiles [Arm07] separate CSP operators into low- and high-level operators, according to the shape of the LTS they generate. High-level operators are all those that can be represented as a compositional (and modular) process, which could be compressed with automata-theoretic techniques, such as bisimulation. After the application of their operational semantics rules, the resulting transition contains the same high-level operator on all its components. The most common high-level operators are hiding, parallelism, and renaming. On the other hand, low-level operators are all those that provide the core sequential language. Their operational semantics rules changes the "shape" of the process in the resulting transition. This usually, but not necessarily, leads to LTSs that are not very compression-sensitive. The most common low-level operators are prefixing, choices, sequential composition, primitive processes, and recursion. Thus, in frequent cases, the more high-level the process, the more amenable to compression the corresponding LTS will be.

Again, here we use LTS as an abstraction for what actually goes on underneath with more efficient B-tree data structures. More details on such separation and the compilation of CSP_M in FDR can be found in [Gol04]. Also, as many compression techniques rely on the presence of internal τ transitions, the earlier events are hidden the better are the chances of compression. Nevertheless, since event hiding could introduce divergence, it should be used with care. Obviously, it may be infeasible to provide the most compact LTS due to the structure of the process being described. Bearing such structuring in mind proves very useful when checking complex or data-intensive specifications [RSR01, Low97]. 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 [Low97, Sri05]. 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, as the example given in Sect. 5.2 shows.

3.2. Available functionality

FDR's object-model provides four main functionalities: (i) session management representing specification sources; (ii) ISMs representing compiled GLTSs which are amenable for manipulation by the 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 [Gol05, 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 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 information. It enables the compilation of CSP processes as ISMs, as well as the evaluation of mathematical expressions given using FDR's functional language constructs, and refinement assertions representing property checks.

Indexed 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. Nevertheless, if FDR allowed ISM compilation from a different (non CSP) source, it could be used to perform refinement model checking for other languages or CSP extensions. Unfortunately, this is not currently possible. Each ISM implements three functionalities: (i) LTS description; (ii) LTS structure; and (iii) LTS analysis. From the LTS description it is possible to retrieve the process name, its original `ASCII` script, and the alphabet of events for all involved processes in the script. 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 via factory methods called **refine** and **refinedBy**. Hypothesis objects represents an assertion about an ISM that is checked with the refinement check algorithms mentioned above. Thus, a hypothesis object can explicate the meaning of a refinement assertion. That is, it can provide 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, and which **parts** of the ISM structure will affect the check.

Debugging information. It has detailed descriptions of witness(es) for a refinement failure, and is the result of a check in the hypothesis object. This information is separated in three categories: (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 describes, 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 flawed 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.

3.3. FDR's API

With FDR's architecture in mind, understanding its API is quite straightforward. As the complete FDR object-model has not been publicly documented before, we give a detailed explanation of the available FDR server methods, which are summarised it in the UML class diagram of Fig. 1. Although this section looks like a manual or reference guide, since such a thorough explanation is new and it sheds light on how *FDR Explorer* is encoded (see the next section), we found appropriate to include it here. Moreover, knowing FDR's API can be very useful for understanding the information contained in debugging witnesses, or how LTSs are composed. In what follows, we explain the most relevant methods from the UML diagram.

`Session` This object represents an entry-point for the FDR server running under the Tcl/Tk interpreter and every other objects is (directly or indirectly) derived from it. To create a new `Session` object, one must first issue a `session` command in the Tcl/Tk command line. The script management methods are self-evident and return the set of **known** processes, declared **assertions** from a **source** file name respectively. Script evaluation methods are the most interesting and are enumerated below:

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

compile(Str, M): compiles the given process in the chosen semantic model. It also allow arbitrary (i.e., unnamed) processes based on the definitions in the script.

evaluate(Str): evaluates the given string as an expression like in functional languages.

In order to return an `ISM` object, the **compile** method requires that process names must include actual parameters for every formal parameter declared, or just the process name if no parameters are given. The available values for the models (**M**) are "`-t`" for traces ($\mathcal{T}$), "`-f`" for failures ($\mathcal{F}$), and "`-fd`" for failures-divergences ($\mathcal{FD}$) refinement. Loading a file through the session object may return some parsing errors if the CSP scripts contains problems or the file cannot be accessed. Both **load** and **compile** may generate parsing and compilation errors, respectively.

`ISM`**.** This object represents a compiled CSP process as an LTS. They are based on the automata theory described in [ClH93], as explained in [RGG95]. Apart from the trivial methods about the textual script this object represents

(i.e., **name**, **shortname**, **describe**), there are three sets of methods related to LTS structures [Sca98, Chaps. 4, 8], refinement algorithms [Ros94, MaH00], and compression [RGG95]. The structural methods are given below.

root: returns the ($\mathbb{N}$) node index where refinement search starts.

alphabet: returns a set of event names (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 length of the **alphabet**. 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 (0) placed between the two node indexes. This event number can be used with 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. That is because the automata is normalised (i.e., made deterministic) collapsing nodes reached through τ events, and this acceptance set represents the event leading to these collapsed nodes as the empty set. 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 set of events from **alphabet** that are immediately available from a given node.

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`).

For those familiar with the Z notation [WoD96], the ISM state could be defined with a schema like

ISM_State

$$
\begin{array}{l}
root : \mathbb{N} \\
alphabet : \mathbb{F}\ NAME \\
events : \mathrm{iseq}\ NAME \\
trans : \mathrm{seq}\,(\mathbb{N} \leftrightarrow \mathbb{N}) \\
div : \mathrm{seq}\ BOOL \\
accs : \mathrm{seq}\,(\mathbb{F}\ \mathbb{N}) \\
initials : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\!+\ \mathbb{F}\ NAME \\
afters : \mathbb{N} \times NAME \twoheadrightarrow\!\!\!\!\!\!+\ \mathbb{F}\ \mathbb{N} \\
\ldots
\end{array}
$$

$$
\begin{array}{ll}
root \in \mathrm{dom}\ trans & (i) \\
alphabet = \mathrm{ran}\ events & (ii) \\
\mathrm{dom}\ trans = \mathrm{dom}\ div = \mathrm{dom}\ accs = \mathrm{dom}\ initials = \mathrm{dom}\,(\mathrm{dom}\ afters) & (iii) \\
\mathrm{dom}\,(\bigcup(\mathrm{ran}\ trans)) \subseteq \mathrm{dom}\ events & (iv) \\
\mathrm{ran}\,(\mathrm{dom}\ afters) \subseteq alphabet & (v) \\
\bigcup(\mathrm{ran}\ initials) \subseteq alphabet & (vi) \\
\forall n : \mathrm{dom}\ trans;\ e : alphabet \bullet initials\, n = \{\, i : \mathrm{dom}\,(trans\, n) \bullet events\, i \,\} & (vii) \\
\quad \wedge\ afters\,(n, e) = (trans\, n)(\!|\ \{events^{\sim}\ e\}\ |\!) & (viii) \\
\quad \wedge\ e \in initials\, n \Leftrightarrow afters\,(n, e) \neq \emptyset & (ix) \\
\ldots &
\end{array}
$$

assuming $NAME$ and $BOOL$ are defined as given sets. With such a schema, it is easy to see some of the invariants FDR ISM objects have. That is, from each numbered predicate in the schema we know that: (i) the **root** node index is always in the **transitions**; (ii) the **alphabet** is directly related to **events**, which are not repeated (iseq $NAME$); (iii) every node index in **transitions** has information about **divergences**, **acceptances**, **initials** and **afters**; (iv)/(v)/(vi) the event indexes in **transitions**, **afters**, and **initials** belong to the **alphabet**; (vii)/(viii) the initial events and target nodes are deduced from the transition systems; and (ix) every initial event in the transition system lead to at least one target node. We have used a variation of this Z model in order to formalise parts of FDR itself. This exercise enabled by the *FDR Explorer* tool has proved very useful in the design and implementation of another refinement model checker tool [Fre05, FCW06].

ISM objects also provide analysis methods enabling the user to choose a refinement algorithm to perform. They create assertions as `Hypothesis` objects as summarised below.

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**`")`.

These methods are triggered in the CSP_M scripts through the `assert` keyword. Furthermore, these operations could also be formalised in Z as operators over the *ISM_State* schema above. For instance, if we include the following command in the example provided in Sect. 2:

```
assert D :[ deadlock free [FD] ]
```

FDR can execute a deadlock freedom check for process `D` in the $\mathcal{FD}$ model. This `assert` is represented internally as a call to the **deadlockfree** method of the compiled ISM for `D` with a "`-fd`" parameter. 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 [Gol05, p. 60] and [RGG95], and further discussed in the example given in Sect. 5.2.

cheap: flags whether the **attribution** of a `behaviour` to the subcomponents of the process is: relatively straightforward, because the ISM node is not the location of a compressed ISM; or potentially expensive, since the contributing behaviours will need to be reconstructed by examining the uncompressed form of a compressed process.

composite: returns whether **this** high-level ISM is decomposable with **parts**.

parts: returns a set of subcomponent ISMs that the root node index represents. That means, if **composite** is *true*, **parts** is not empty.

operator: returns the high-level operator used to compose other **parts** of **this** ISM, where the possible values are `shared` for interleaving, `parallel` interface parallelism, `link` for piping, `rename` for renaming, and `hide` for hiding.

wiring: returns a set of **event** indexes associated with the **composite operator** in a **cheap** ISM, such as those of a synchronisation set, a renaming relation, or hiding set.

compress(Str): applies the named compression and returns a new ISM. This name must be known to the FDR server, and must be previously declared using the CSP_M `transparent` keyword.

apportion(Str): returns the ISM subcomponents' contribution to a `Behaviour` object. That is, with the `Behaviour` object name, the method returns the witness information related to the corresponding **part**.

In the FDR debugging window GUI, the **cheap** flag is used to determine whether to automatically unfold below a chosen node, and whether to draw the operator in black or red according to the resulting value. As high-level processes are easier to decompose and more amenable to compression, hence explore the modular structure of CSP operators, inspecting this flag can be useful to adjust complex or data intensive processes appropriately.

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 Sect. 2, we show how the LTS of some peculiar processes are structured, and in Sect. 5, we show how one can use our tool to gain useful insight about available compression techniques with great results and little effort.

`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 structures available and is to be used with the ISM **apportion** method. They contain information about (minimal) acceptances (or maximal refusals) after some trace has taken place. This is represented in FDR by the small window within the main debugging window that mentions observed and permitted events. In this case of deadlock freedom check for process `D` above, the `Hypothesis` object generated contains one witness with one debug tree and the following behaviour can be extracted:

Performs: $\langle$_tick$\rangle$
Accepts : {} **Could accept**: {_tick} {c}
Refuses : {_tick c} **Could refuse** : {c} {_tick}

That means, after performing the event _tick ($\surd$), `D` is refusing both _tick and c, whereas the specification for deadlock freedom (see *DF* in [Ros97, 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. *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 [JGRAPH]. 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 Sect. 6.

4.1. 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, are logged. Next, a given list of processes (with actual parameters if needed) is compiled into ISMs, and detailed information about their structure is 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 Fig. 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 for failures.

The helper methods provide documentation for every method of every class in FDR's object model from Fig. 1. As this can be quite verbose, the default output is just for `ISM`, `Hypothesis`, and `DebugContext` objects. Nevertheless, one could selectively call helper methods for other available classes directly.

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.

In the recent updated version, the user can also choose whether to compile composite **parts** of ISMs or not. That is, whenever the ISM method **parts** is not empty, we can recursively collect information about all of its elements. This is useful in understanding how FDR takes time building low-level processes and then cleverly just stores their composition patterns rather than the whole transition system, as can be inspected through the **wiring** and **operator** methods. This result had impacts beyond understanding how FDR or CSP_M works. It led to considerable design decisions throughout the development another model checker that handles both behavioural and data aspects of concurrent systems by combining refinement model checking techniques in the spirit of FDR with theorem proving and symbolic execution [Fre05].

4.2. *FDR Explorer* API

In this section, we detail below the most relevant methods for each category mentioned above.

Inspection methods

inspectProcs(File,List⟨Str⟩,Bool,Bool): inspects the given list of processes from a file handle pointing to the CSP script file. The first flag indicates whether we should (recursively) compile the inner **parts** of a (high-level) process, or just its root ISM. The last flag indicates whether the FDR objects created should be deleted or not. Inspecting the ISM object **parts** can be useful to spot possible places for compression, or make a full inspection of the whole process structure rather than just the root process. Obviously, because we need to compile more than one ISM, this choice incurs greater execution times. For execution in batch mode, the last flag should be set to *true* (or 1), whereas execution in interactive mode when further inspection by the is required, it should be set to *false* (or 0) in order to avoid garbage collection.

inspectParameterless (File, Bool, Bool): inspects all processes from the given file. It generates an error if any of the processes in the file have parameters. The flags are the same as those of **inspectProcs**.

The results are logged into separate files for each process in the given list (or file name). 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 log. 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 logging the results on the given file handle. Parameterised processes must be instantiated, otherwise the FDR server crashes. It returns a compiled ISM object to the user.

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

Extraction methods. There is one extraction method for each class in Fig. 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 for each class as well. They log a description of the role that each method of each class has. We also group commonly used helper methods together.

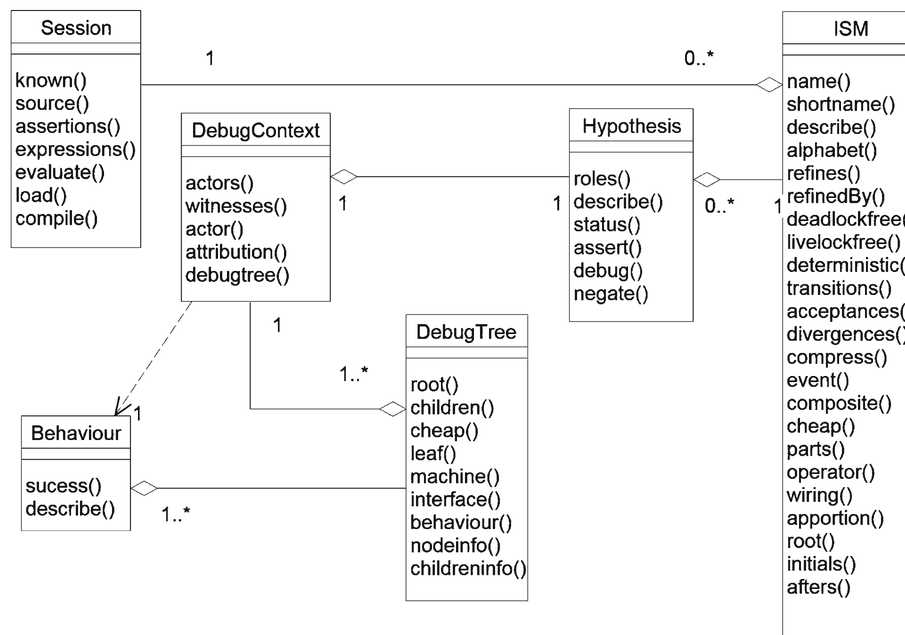

Fig. 1. UML class diagram of FDR object model

Auxiliary methods

FDRExplorerHelp: provides a thorough 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.

Finally, to integrate our tool into FDR, we hook the *FDR Explorer* API scripts into FDR's Tcl/Tk `main` method. This is illustrated in the next subsection below.

4.3. Calling the APIs

Now, let us show how one could perform the check explained in the example of Sect. 2 using the *FDR Explorer* API. Assuming that FDR is installed at `$FDRHOME`, we start the FDR server with the following command from the shell prompt:

```
venice$ $FDRHOME/bin/fdr2tix -insecure -nowindow
```

Now the FDR server is running and the Tcl/Tk interpreter is loaded, assuming *FDR Explorer* is installed at `$FDREXPLORER_HOME`, which points to "`./FDRExplorer`", we start its interface using the next command

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

As Tcl/Tk variables are accessed using the dollar sign in the FDR server in batch mode, to access the environment variable `$FDREXPLORER_HOME`, one can use the undocumented function `$env(VAR)`, where `VAR` is the environment variable without the dollar sign. This is how one can work into the interactive batch mode of FDR.

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 summarised in Table 1 with neither recursively inspecting ISM **parts**, nor deleting the generated objects using the commands

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

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

```
% inspectParameterless "./skip.csp" 0 0
```

As a result of executing the inspection methods, four files are created and named `skip.X.csp.exp`, where `X` will be either `A, B, C,` or `D`. They contain detailed information about each process ISM, as well as the three 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 instance name to use is the one FDR returns, and we could type a command, such as

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

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

5. Tackling state explosion

State explosion is one of the greatest problems in model checking, as it forbids acceptable running time and compromises scalability, where parallel interaction among processes affords evidence to its occurrence. To aid this problem in the CSP scenario of FDR, one requires great skill with CSP_M that often includes coding tricks and patterns. To acquire such skills demands time and expertise from the designer, which could compromise the whole verification exercise due to inappropriate scripts leading to unacceptable running times. As we shall see in the example below, one can use *FDR Explorer* to avoid such demand.

One way around this problem is either theoretical via abstraction techniques [Ros97, Chap. 12], or practical through automata compression methods [RGG95]. The former, usually requires the user to know about the underlying semantic models thence propose abstract versions of it, whereas the latter solely relies on the tool to provide the functionality thence the user just needs to know when to use it.

5.1. Compression functions

FDR provides a series of LTS compression functions that enable the user to dramatically decrease the number of states and transitions needed to represent the underlying processes without compromising their semantics. As mentioned in FDR's manual the most challenging aspect in compressing ISMs is to preserve the structured debugging facilities available for the uncompressed LTS. To achieve this, as it happens with process compilation, debugging information is generated on-the-fly as the user requests it from each part of the top-level process where a failure had been found. The downside of this requirement is that processes with long checking times could require twice as much time if debugging follows from such checks. On the other hand, luckily, the result of applying compression functions shall be twice as efficient in a similar scenario.

For compression to work effectively, however, there are two general rules: (i) bind communicating processes as early (and with the smallest low-level processes) as possible; and (ii) hide as much of the events, and as close to the low-level processes, as possible. That is, it is better to put small processes in parallel early rather than composing sequential process together first for parallelisation later. Also, the earlier the events are hidden, the more internal transitions that could be compressed one shall have. The only worry on such attempts is that hiding may introduce divergence, hence invalidate a refinement check in the failures-divergences model. Nevertheless, if the property of interest is a safety (traces) or nondeterministic (stable-failures) property, or else the process is know to be divergence-free by construction or through other methods, hiding events does not invalidate refinement checks and should be done as much as possible in order to take advantage of the many compression techniques available.

FDR's user manual provides a good theoretical explanation that accounts for the various compression techniques available, and when/where to apply then from a denotational semantics point of view. It does not include, however, an illustrative (and intuitive) example on the various options. It also does not mention one quite interesting (undocumented) compression function named `chase`, which chases internal (τ) transitions as much as possible, which is briefly explained in [Ros97, Sect. 15.4]; it is also mentioned in [RSR01]. Differently from the other compression functions, as it can select from a set of possible nondeterministic paths to follow, `chase(P)` might change the value of a process `P`, and should only be used when one knows it does not. This certainty is given whenever `P` is deterministic. Anyway, it is always true that

$$P \sqsubseteq_{FD} \text{chase}(P)$$

because `chase` can only perform transitions that the implementation is free to choose. Nevertheless, `chase` does not remove implicit nondeterminism, as in an external choice of overlapping events, such as

```
a -> P [] a -> Q
```

since it does not involve τ-transitions. This discussion is useful, yet curiosity plays a useful part in understanding how it actually affects the compiled transition systems. Originally, this curiosity came from cryptic CSP scripts found through work with highly optimised CSP code developed by one of our industry collaborators.

FDR Explorer is particularly useful in pursuing this productive curiosity about compression techniques. By inspecting appropriate processes that exploit the benefits of each compression function, the CSP user can learn quite a lot about how to nicely lay out processes by recognising the different patterns the (originally) generated (and later compressed) automata have.

5.2. Example: interleaved buffers

To illustrate this scenario about the advantages *FDR Explorer* might bring to the user in terms of learning/understanding of what goes on "under the bonnet", we have encoded a set of interesting processes where the effects of compression functions can be observed. The intention is to show how a naive attempt at inspecting a CSP_M script with *FDR Explorer* can be quite productive. For that, we have chosen the well-known one-place buffer example

$$COPY = left?x \rightarrow right!x \rightarrow COPY$$

We use a variation of the script simple.csp available from FDR's distribution, which provides an abstract specification of a chain of such buffers and a multiplexed implementation. Firstly, we define a set of values to be communicated over the channels with a data type of fruits is defined.

```
datatype FRUIT = apples | oranges | pears
channel left,right,mid : FRUIT
channel ack
```

Next, the processes are defined as follows

```
-- abstract specification as a chain of one-place buffers
COPY    = left?x -> right!x -> COPY
BUFF(n) = [right <-> left] i: <1..n> @ COPY

-- impl.  as a chain of n copies of two processes communicating over mid and ack
SEND           = left?x -> mid!x -> ack -> SEND
REC            = mid?x -> right!x -> ack -> REC
SYSTEM         = (SEND [| {| mid, ack |} |] REC) \ {| mid, ack |}
SYSTEM_BUFF(n) = [right <-> left] i: <1..n> @ SYSTEM
```

The abstract specification is defined by `COPY` and `BUFF` as a chain of one-place buffers, and the remaining four processes as a chain of `n` parallel implementations of two processes communicating over the *mid* and *ack* channels using the CSP piping operator ($\gg$), which is given in CSP_M by linking the channels to be plumbed together. In order to better explore the compression techniques over τ-transitions, such as `chase`, we introduce further nondeterminism by interleaving (`m`) such (`n`) buffer chains.

```
INT_BUFF(m, n)     = ||| i: {1..m} @ BUFF(n)
INT_SYSTEM0(m, n) = ||| i: {1..m} @ SYSTEM_BUFF(n)
```

That is, we have `m` interleaved buffers of `n` chains of `COPY` and `SYSTEM`. Although there may be other interesting buffer processes we could have analysed, our choice was due to its suitability to illustrate a naive attempt at exploiting FDR's compression functions. To see how the experiments would scale, we define global constants for the number of processes and chains as

```
PROCESS_COUNT = 2
BUFFER_COUNT  = 2
```

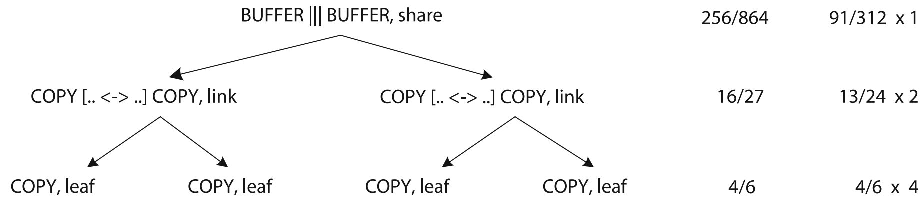

Fig. 2. Abstract interleaved buffer LTS structure and ISM information

where the interleaved buffers are instantiated accordingly as

```
INT_BUFFER = INT_BUFF(PROCESS_COUNT, BUFFER_COUNT)
INT_SYSTEM = INT_SYSTEM0(PROCESS_COUNT, BUFFER_COUNT)
```

In this setting, the following refinements (or rather equivalences) hold

$$
\begin{array}{llll}
COPY & \sqsubseteq_{FD} SYSTEM & SYSTEM & \sqsubseteq_{FD} COPY \\
BUFFER & \sqsubseteq_{FD} SYSTEM_BUFFER & SYSTEM_BUFFER & \sqsubseteq_{FD} BUFFER \\
INT_BUFFER & \sqsubseteq_{FD} INT_SYSTEM & INT_SYSTEM & \sqsubseteq_{FD} INT_BUFFER
\end{array}
$$

As soon as we start increasing the number of processes and buffers, the checks become more time/space consuming for `INT_BUFFER` and `INT_SYSTEM`. Using the compression functions with a variation of these two global constants achieves great compression rates. In fact, as one would expect, the greater the parameters, the better the compression, since our system is highly nondeterministic and contains a large amount of τ-transitions.

In this kind of example, one goal is to use *FDR Explorer* to spot possible places for compression via informed inspection of *FDR Explorer*'s log. This can be clearly observed by using *FDR Explorer* to inspect the structure of the compressed automata with smaller values, and later comparing the number of nodes and transitions with bigger values for those global constants. Let us illustrate this in practice by examining the node and transition count information gathered after running plain and compressed versions of both `INT_BUFFER` and `INT_SYSTEM` using *FDR Explorer*. As before, to do that we need to start FDR in batch mode assuming the processes are typed into a file named `simple.csp` in the current directory

```
venice$ $FDRHOME/bin/fdr2tix -insecure -nowindow
        % source "./FDRExplorer/FDRExplorer.tcl"
          % set lprocs { INT_BUFFER INT_SYSTEM }
       % inspectProcs "./simple.csp" $lprocs 1 0
```

They compile the two processes and their corresponding parts to recursively generating two files with the detailed ISM information we are looking for. From the knowledge about how FDR structures CSP in low- and high-level processes, one can see the obvious candidates for compression are the processes with interleaving and piping, as their (ISM) **parts** are formed by high-level processes, and the ISM **composite** method result is *true* (1). The effect is quite nice as the order of magnitude of the achieved compression is proportional to the number of processes and chains; that is, the greater the value of `PROCESS_COUNT` and `BUFFER_COUNT` the greater the order of LTS compression achieved. This is a direct consequence of the modularity and structural composition of CSP operations mentioned above. We discuss how to get to this result in the sequel.

In Fig. 2, we draw the process tree for the the `INT_BUFFER` process with 2 processes and 2 chained buffers. Each element of the tree corresponds to an ISM object. Firstly, the result of the ISM **describe** method gives the CSP_M script it represents. Next, the ISM **operator** method gives the way ISMs are composed. Finally, the node-count/transition-count for uncompressed and compressed ISMs with their corresponding number of instances completes the figure. For instance, the root node is described by process verb `BUFFER ||| BUFFER` and is composed using the `shared` operator. The uncompressed version has 256 nodes and 864 transitions, whereas the compressed version has 91 nodes and 312 transitions, and this ISM is repeated once. The other **composite** ISMs are laid out similarly. Not surprisingly, as the `leaf` (`COPY`) processes are formed by low-level CSP operators and have no parts, it cannot be further compressed. Likewise, the implementation is presented in the Fig. 3. Not surprisingly, process `INT_SYSTEM` benefits more from ISM compression, since it has two further high-level ISM components with the hiding and parallelism inherited from `SYSTEM`. The same good result repeats itself if we increase the values of the global constants.

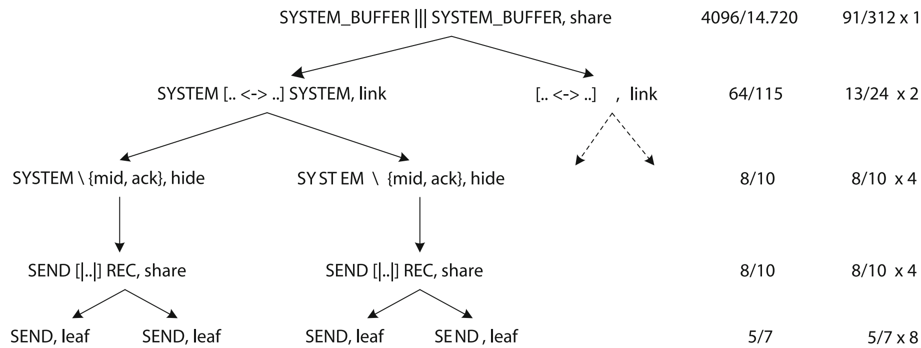

Fig. 3. Interleaved buffer implementation LTS structure and ISM information

Interestingly for this example, it was not through our CSP_M experience in compression techniques, but through inspection of the logged results from *FDR Explorer* from both `INT_BUFFER` and `INT_SYSTEM` that we found out where to apply the compression functions to achieve the results mentioned. That is, we inspected the ISM **parts** and **operator** method results to conclude which ISMs were **composite**, hence more susceptible to compression. We then compressed both root nodes, as they are clearly high-level processes, since they are formed by a parallel operator (i.e., interleaving). Once the place for compression has been spotted, now we need to choose which compression technique to use. As the processes are highly nondeterministic, the `chase` compression function would be a good choice. Unfortunately, it is not quite useful in this case as it changes the intended behaviours and invalidates the refinement between the processes. One good choice is the (semantic-preserving) `model_compression` function as it exploits weak (bisimilarities) equivalences, which happens to occur often in these processes. If one cannot spot the appropriate compression method to use, one naive strategy is to inspect every available one with *FDR Explorer* and see which ones works best. In time, one gets the feeling of when to use what. That is a real gain that *FDR Explorer* can give to the user through empirical knowledge of (rather complex denotational) compression techniques. It can be also helpful for automatic CSP_M scripting tools that generate the necessary combinations of possible compression functions and processes depending on the values of ISM method results. The use of the `model_compress` function on the root processes is defined below. To use any compression function, we first must declare its name with the `transparent` keyword.

```
transparent model_compress
MC_INT_BUFFER = model_compress(INT_BUFFER)
MC_INT_SYSTEM = model_compress(INT_SYSTEM)
```

Repeating the same trick through inspection of inner (composite) ISMs, we reach the best possible compression result using `model_compress`, by compressing the piped buffers, as shown in the processes below.

```
-- compressed abstract buffer
MC_BUFFER       = model_compress(BUFF(BUFFER_COUNT))
INT_MC_BUFFER    = ||| i: {1..PROCESS_COUNT} @ MC_BUFFER
MC_INT_MC_BUFFER = model_compress(INT_MC_BUFFER)

-- compressed buffer implementation
MC_SYSTEM       = model_compress(SYSTEM_BUFF(BUFFER_COUNT))
INT_MC_SYSTEM    = ||| i: {1..PROCESS_COUNT} @ MC_SYSTEM
MC_INT_MC_SYSTEM = model_compress(INT_MC_SYSTEM)
```

Table 2 presents compression results as node/transition counts for processes with global constants set to 2 interleaved processes and 2 chained buffers. For instance, the process `INT_SYSTEM` has the most dramatic improvement. The compressed `MC_INT_MC_SYSTEM` has 221/496 nodes/transitions representing a 19.58/30.41 times improvement from the original `INT_SYSTEM` that has 4,328/15,086 nodes/transitions. Furthermore, these results are the summation of the nodes/transitions given in Figs. 2 and 3.

The ultimate goal is to use *FDR Explorer* to gain insight into the precise (and formal) description of the transformation done by those compression functions; however, a brief (informal) source of explanation can be found in [Ros97, App. C.2]. That is, with the knowledge of the operational semantics of CSP_M, one can foresee

Table 2. Node and transition counts (NC/TC) with relative compression gain

Configuration →	2P × 2B	
Processes ↓	**NC**	**TC**
INT_BUFFER	304	942
INT_MC_BUFFER	139	390
NC_INT_MC_BUFFER	133	384
INT_SYSTEM	4,328	15,086
INT_MC_SYSTEM	323	678
MC_INT_MC_SYSTEM	221	496

the structure of the automata based on the low- and high-level process constructs, hence take appropriate design choices to avoid state explosion. For the compression functions, this can also be achieved by understanding the transformations that take place at the automata level. Furthermore, using *FDR Explorer* to inspect the result of ISM methods, such as **composite** and **parts**, gives the clue to whether compression functions would be useful or not. For instance, the use of `normalise` is not appropriate here because the nondeterministic patterns makes the automata grow rather than shrink in this case. Similarly, as mentioned before, the use of `chase` would also be unfortunate as it changes the original behaviours of processes and the above refinement checks would fail.

This example clearly shows how the use of compression can dramatically impact the effectiveness of model checking. Obviously, one can use further abstraction or semantic oriented techniques, but those require some sort of expertise from the user, which could be avoided at first by using *FDR Explorer*. The tool also serves as a learning aid for those compression techniques. Finally, since we could automatically distinguish low- and high-level parts of CSP processes, another user interface tool could be built on the top of *FDR Explorer* to provide "compression hints" to the user or automatic script generation.

6. Graph visualisation

At the moment, the translation strategy from FDR LTS transitions to an available graph format (named JGraph [JGRAPH]) is indeed very simple. For every transition from FDR's ISM, 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 and extracted via *FDR Explorer*, hence improving the user friendliness of 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. Also, more powerful graph formats using XML could also be used to encode other interesting features in a better way from the software engineering point of view.

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 could not 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 for fiddling with the CSP script has been successfully achieved through different CSP specification patterns via the Casper tool [Low97], 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!

7. Conclusion

In this paper we present a new interface to the CSP [Ros97] refinement model checker FDR [Gol05], 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 [Fre05]. 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.[1]

The main contribution of *FDR Explorer* is enabling better integration between CSP script generation tools [Sri05], 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, such as the use of compression functions. These facilities are not available from the original FDR GUI. This follows the trend of tool integration set out by a grand challenges in computer research [BHW06].

We 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. We also show an extensive example on how to take advantage of *FDR Explorer* inspection mechanism to learn quite complex CSP compression techniques rather easily and naturally.

Finally, we explain how we transformed the available CSP LTS transitions into a graph notation format with visualisation tool support [JGRAPH]. 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. This integrated drawing environment could also hint possible CSP LTS compression opportunities based on the LTS structure *FDR Explorer* exposes.

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. Unfortunately, this is not yet possible since the FDR server only allows CSP LTS creation through CSP_M scripts. 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 [Val90]. Furthermore, from a presentation of an early version of this paper to people involved in the development of FDR, it became clear their interest in integrating the ideas behind *FDR Explorer* in the next major FDR release, which was something quite rewarding and exciting.

Acknowledgments

Most helpful comments and pointers to right references were given in three different occasions by Michael Goldsmith, one of FDR's authors. We would also like to thanks Alexandre Motta from UFPE in Brazil for using the tool in an industry-scale scenario for Motorola. We are also grateful Jim Davies for his encouraging comments about the purposefulness of the tool and its great potential. Finally, we are grateful to QinetiQ Malvern for their long-term support of our research group.

Appendix: Interleaved buffers CSP_M code

```
-- Simple demonstration of FDR2
-- A single place buffer implemented over two channels
-- Original by D.Jackson 22 September 1992
-- Modified for FDR2 by M.  Goldsmith 6 December 1995
-- Modified for FDR Explorer by Leo Freitas 10 January 2007

-- First, the set of values to be communicated
datatype FRUIT = apples | oranges | pears

transparent chase
transparent sbsim
```

[1] See the Circus web site http://www.cs.york.ac.uk/circus for more details on these works.

```
transparent diamond
transparent normalise
transparent explicate
transparent model_compress
transparent tau_loop_factor
squidge(P) = normalise(diamond(P))

-- Channel declarations
channel left,right,mid : FRUIT
channel ack

-- The specification is simply a single place buffer
COPY(in, out) = in ? x -> out ! x -> COPY(in, out)

-- N one-place buffers chained together, 1 is too few, 2 is interesting, 3 is too much
PROCESS_COUNT = 3
BUFFER_COUNT  = 3

BUFF(n, in, out) = [out<->in] i: <1..n> @ COPY(in, out)
INTBUFF(n, m, in, out) = ||| i: {1..n} @ BUFF(m, in, out)

BUFFER    =  BUFF(BUFFER_COUNT, left, right)
INTBUFFER  =  INTBUFF(PROCESS_COUNT, BUFFER_COUNT, left, right)

-- interleaved model compressed buffer
MC_BUFFER = model_compress(BUFF(BUFFER_COUNT, left, right))
INT_MC_BUFFER = ||| i: {1..PROCESS_COUNT} @ MC_BUFFER
MC_INT_MC_BUFFER = model_compress(INT_MC_BUFFER)

COPY_INT = ||| i: {1..PROCESS_COUNT} @ COPY(left, right)
CHASE_COPY_INT = chase(COPY_INT)
NORMALISE_COPY_INT = normalise(COPY_INT)
EXPLICATE_COPY_INT = explicate(COPY_INT)

-- Apply some compression methods to the buffer

CHASE_BUFFER = chase(BUFFER)
SBSIM_BUFFER = sbsim(BUFFER)
DIAMOND_BUFFER = diamond(BUFFER)
NORMALISE_BUFFER = normalise(BUFFER)
EXPLICATE_BUFFER = explicate(BUFFER)
MODELCOMPRESS_BUFFER = model_compress(BUFFER)
TAULOOPFACTOR_BUFFER = tau_loop_factor(BUFFER)
SQUIDGE_BUFFER = squidge(BUFFER)

CHASE_INTBUFFER = chase(INTBUFFER)
SBSIM_INTBUFFER = sbsim(INTBUFFER)
DIAMOND_INTBUFFER = diamond(INTBUFFER)
NORMALISE_INTBUFFER = normalise(INTBUFFER)
EXPLICATE_INTBUFFER = explicate(INTBUFFER)
MODELCOMPRESS_INTBUFFER = model_compress(INTBUFFER)
TAULOOPFACTOR_INTBUFFER = tau_loop_factor(INTBUFFER)
SQUIDGE_INTBUFFER = squidge(INTBUFFER)

-- The implementation consists of two processes communicating over
```

```
-- mid and ack
SEND(in) = in ? x -> mid ! x -> ack -> SEND(in)
REC(out) = mid ? x -> right ! x -> ack -> REC(out)

-- These components are composed in parallel and the internal comms hidden
SYSTEM(in, out) = (SEND(in) [| {| mid, ack |} |] REC(out)) \ {| mid, ack |}

-- N systems as chained buffers
SYSTEM_BUFF(n, in, out) = [out<->in] i: <1..n> @ SYSTEM(in, out)
SYSTEM_BUFFER = SYSTEM_BUFF(BUFFER_COUNT, left, right)

-- interleaved model compressed system buffer
MC_SYSTEM_BUFFER = model_compress(SYSTEM_BUFF(BUFFER_COUNT, left, right))
INT_MC_SYSTEM_BUFFER = ||| i: {1..PROCESS_COUNT} @ MC_SYSTEM_BUFFER
MC_INT_MC_SYSTEM_BUFFER = model_compress(INT_MC_SYSTEM_BUFFER)

INT_SYSTEM_BUFF(n, m, in, out) = ||| i: {1..n} @ SYSTEM_BUFF(m, in, out)
INT_SYSTEM_BUFFER = INT_SYSTEM_BUFF(PROCESS_COUNT, BUFFER_COUNT, left, right)

-- Apply diamond elimination to the system buffer version
CHASE_SYSTEM_BUFFFER = chase(SYSTEM_BUFFER)
DIAMOND_SYSTEM_BUFFER = diamond(SYSTEM_BUFFER)
EXPLICATE_SYSTEM_BUFFER = explicate(SYSTEM_BUFFER)
NORMALISE_SYSTEM_BUFFER = normalise(SYSTEM_BUFFER)

CHASE_INT_SYSTEM_BUFFFER = chase(INT_SYSTEM_BUFFER)
DIAMOND_INT_SYSTEM_BUFFER = diamond(INT_SYSTEM_BUFFER)
EXPLICATE_INT_SYSTEM_BUFFER = explicate(INT_SYSTEM_BUFFER)
NORMALISE_INT_SYSTEM_BUFFER = normalise(INT_SYSTEM_BUFFER)

FILTERED_CHAOS(x) = CHAOS(diff({|left, right|}, { left.x, right.x }))
COPY_CHAOS0(x)    = left.x -> right.x -> COPY_CHAOS0(x)
COPY_CHAOS1(x)    = COPY_CHAOS0(x) ||| FILTERED_CHAOS(x)
COPY_CHAOS        = [] x: FRUIT @ COPY_CHAOS1(x)

BUFF_CHAOS(n)     = [right<->left] i: <1..n> @ COPY_CHAOS
INTBUFF_CHAOS(n,m) = ||| i: {1..n} @ BUFF_CHAOS(m)
BUFFER_CHAOS   = BUFF_CHAOS(BUFFER_COUNT)
INTBUFFER_CHAOS = INTBUFF_CHAOS(PROCESS_COUNT, BUFFER_COUNT)

CHASE_COPY_CHAOS = chase(COPY_CHAOS)
NORMALISE_COPY_CHAOS = normalise(COPY_CHAOS)
MODEL_COMPRESS_INTBUFFER_CHAOS = model_compress(INTBUFFER_CHAOS)
BUFF_NORM_CHAOS(n)     = [right<->left] i: <1..n> @ NORMALISE_COPY_CHAOS
INTBUFF_NORM_CHAOS(n,m) = ||| i: {1..n} @ BUFF_NORM_CHAOS(m)
BUFFER_NORM_CHAOS   = BUFF_NORM_CHAOS(BUFFER_COUNT)
INTBUFFER_NORM_CHAOS = INTBUFF_NORM_CHAOS(PROCESS_COUNT, BUFFER_COUNT)

NORMALISE_BUFFER_NORM_CHAOS = normalise(BUFFER_NORM_CHAOS)
NORMALISE_INTBUFFER_NORM_CHAOS = normalise(INTBUFFER_NORM_CHAOS)

-- Checking "SYSTEM" against "COPY" will confirm that the implementation
-- is correct.

MODEL_COMPRESS_INTBUFFER = model_compress(INTBUFFER)
```

```
MODEL_COMPRESS_INT_SYSTEM_BUFFER = model_compress(INT_SYSTEM_BUFFER)


--assert INTBUFF(PROCESS_COUNT,1,left,right) [FD= COPY_INT
--assert COPY_INT [FD= INTBUFF(PROCESS_COUNT,1,left,right)

assert COPY(left, right) [FD= SYSTEM(left, right)
assert BUFFER [FD= SYSTEM_BUFFER

assert INTBUFFER [FD= MODEL_COMPRESS_INTBUFFER
assert INTBUFFER [FD= MC_INT_MC_BUFFER

assert MODEL_COMPRESS_INTBUFFER [FD= MC_INT_MC_BUFFER


assert INTBUFFER [FD= INT_SYSTEM_BUFFER
assert INTBUFFER [FD= MODEL_COMPRESS_INT_SYSTEM_BUFFER
assert INTBUFFER [FD= MC_INT_MC_SYSTEM_BUFFER

assert MODEL_COMPRESS_INTBUFFER [FD= INT_SYSTEM_BUFFER
assert MODEL_COMPRESS_INTBUFFER [FD= MODEL_COMPRESS_INT_SYSTEM_BUFFER
assert MODEL_COMPRESS_INTBUFFER [FD= MC_INT_MC_SYSTEM_BUFFER

assert MC_INT_MC_BUFFER [FD= INT_SYSTEM_BUFFER
assert MC_INT_MC_BUFFER [FD= MODEL_COMPRESS_INT_SYSTEM_BUFFER
assert MC_INT_MC_BUFFER [FD= MC_INT_MC_SYSTEM_BUFFER

-- In fact, the processes are equal, as shown by
assert SYSTEM(left, right) [FD= COPY(left, right)
assert SYSTEM_BUFFER [FD= BUFFER

assert MODEL_COMPRESS_INTBUFFER [FD= INTBUFFER
assert MC_INT_MC_BUFFER [FD= INTBUFFER

assert MC_INT_MC_BUFFER [FD= MODEL_COMPRESS_INTBUFFER


assert INT_SYSTEM_BUFFER [FD= INTBUFFER
assert MODEL_COMPRESS_INT_SYSTEM_BUFFER [FD= INTBUFFER
assert MC_INT_MC_SYSTEM_BUFFER [FD= INTBUFFER

assert INT_SYSTEM_BUFFER [FD= MODEL_COMPRESS_INTBUFFER
assert MODEL_COMPRESS_INT_SYSTEM_BUFFER [FD= MODEL_COMPRESS_INTBUFFER
assert MC_INT_MC_SYSTEM_BUFFER [FD=MODEL_COMPRESS_INTBUFFER

assert INT_SYSTEM_BUFFER [FD= MC_INT_MC_BUFFER
assert MODEL_COMPRESS_INT_SYSTEM_BUFFER [FD= MC_INT_MC_BUFFER
assert MC_INT_MC_SYSTEM_BUFFER [FD= MC_INT_MC_BUFFER

--assert COPY_CHAOS(left,right) [FD= COPY(left,right)
--assert COPY_CHAOS [FD= COPY(left,right)
--assert COPY_CHAOS [FD= CHASE_COPY_CHAOS

--assert BUFFER_CHAOS [FD= BUFFER
--assert INTBUFFER_CHAOS [FD= INTBUFFER
```

```
--this one fails!
--assert CHASE_COPY_CHAOS [FD= COPY(left,right)

--assert NORMALISE_COPY_CHAOS [FD= COPY(left,right)
--assert NORMALISE_BUFFER_NORM_CHAOS [FD= BUFFER
--assert NORMALISE_INTBUFFER_NORM_CHAOS [FD= INTBUFFER


-- tb.csp=transparent.buffer.csp
-- $FDRHOME/bin/fdr2tix -insecure -nowindow
-- source "/usr/leo/fdr/ok/FDRExplorer.tcl"
-- set lprocs { INTBUFFER CHASE_INTBUFFER SBSIM_INTBUFFER DIAMOND_INTBUFFER
--    NORMALISE_INTBUFFER EXPLICATE_INTBUFFER MODELCOMPRESS_INTBUFFER
--    TAULOOPFACTOR_INTBUFFER SQUIDGE_INTBUFFER SYSTEM_BUFFER
--    CHASE_SYSTEM_BUFFFER DIAMOND_SYSTEM_BUFFER
--    NORMALISE_SYSTEM_BUFFER COPY_INT }
--
-- set lprocs { INTBUFFER CHASE_INTBUFFER NORMALISE_INTBUFFER EXPLICATE_INTBUFFER
--    SYSTEM_BUFFER CHASE_SYSTEM_BUFFFER NORMALISE_SYSTEM_BUFFER
--    EXPLICATE_SYSTEM_BUFFER COPY_INT CHASE_COPY_INT NORMALISE_COPY_INT
--    EXPLICATE_COPY_INT }
-- set lprocs { INTBUFFER CHASE_INTBUFFER NORMALISE_INTBUFFER INT_SYSTEM_BUFFER
--    CHASE_INT_SYSTEM_BUFFFER NORMALISE_INT_SYSTEM_BUFFER }
-- set lprocs { SYSTEM_BUFFER CHASE_SYSTEM_BUFFER NORMALISE_SYSTEM_BUFFER }
-- set lprocs { BUFFER BUFFER_CHAOS CHASE_BUFFER CHASE_BUFFER_CHAOS }
-- INTBUFFER_CHAOS  crashes! 7,870,000 transition
-- set lprocs { INTBUFFER NORMALISE_INTBUFFER INTBUFFER_NORM_CHAOS
--    NORMALISE_INTBUFFER_NORM_CHAOS }
-- set lprocs { INTBUFFER_NORM_CHAOS NORMALISE_INTBUFFER_NORM_CHAOS }

-- set lprocs { INTBUFFER MODEL_COMPRESS_INTBUFFER MC_INT_MC_BUFFER INT_SYSTEM_BUFFER
--    MODEL_COMPRESS_INT_SYSTEM_BUFFER MC_INT_MC_SYSTEM_BUFFER }
-- interactiveInspectProcs "/usr/leo/fdr/ok/examples/tb/tb.csp" $lprocs 1
-- source "/usr/leo/fdr/ok/fdr2jgraph.tcl"
-- fdr2jgraph "/usr/leo/fdr/ok/examples/tb/tb.csp" $lprocs "view"
-- interactiveInspectProcs "/usr/leo/fdr/ok/examples/tb/tb.csp" "SYSTEM_BUFFER"
```

References

[Arm07] Armstrong P (2007) Interacting with the CSP compiler. Oxford University Press, Oxford

[BHW06] Bicarregui J, Hoare C, Woodcock J (2006) The verified software repository: a step towards the verifying compiler. FACJ 18(2):143–151

[CaS06] Cabral G, Sampaio A (2006) Formal specification generation from requirement documents. In: Brazilian symposium in formal methods, SBMF.

[ClH93] Cleaveland R, Hennessy M (1993) Testing equivalence as a bisimulation equivalence. FACJ 5(1):1–20. Springer, Heidelberg

[ClW96] Clarke E, Wing J (1996) Formal methods—state of the art and future directions. ACM Comput Surv 28(4):626–643

[Fre05] Freitas L (2005) Model Checking Circus. PhD Thesis, University of York

[Fre07] Freitas L (2007) FDR Explorer v0.4. http://www.cs.york.ac.uk/~leo

[FCW06] Freitas L, Cavalcanti A, Woodcock J (2006) Taking our own medicine: applying the refinement calculus to the development of a model checker. Formal Methods and Software Engineering (8th ICFEM), LNCS 4260, pp 697–716

[FWC06] Freitas L, Woodcock J, Cavalcanti A (2006) State-rich model checking. Innov Softw Eng NASA J 2(1):49–64

[GMR03] Goldsmith M, Moffat N, Roscoe B, Whitworth T, Zakiuddin I (2003) Watchdog transformations for property-oriented model-checking. In: Proceedings of the 12st international FME symposium, Pisa, Italy. LNCS 2805, pp 600–616. Springer, Heidelberg

[Gol04] Goldsmith M (2004) Operational semantics for fun and profit. In: The first 25 years of communicating sequential processes, London, UK. LNCS 3525, pp 265–274. Springer, Heidelberg

[Gol05] Goldsmith M (2005) FDR2 user's manual, version 2.82, Formal Systems (Europe) Ltd

[JGRAPH] JGraph user's manual (2006) http://www.jgraph.com/pub/jgraphmanual.pdf

[Law04] Lawrence J (2004) Practical application of CSP and FDR to software design. In: Abdallah AE, Jones C, Sanders J (eds) 25 years of CSP, FACJ
[Low97] Lowe G (1997) Casper user manual. Oxford University, London
[MaH00] Martin J, Duddart Y (2000) Parallel algorithms for deadlock and livelock analysis of concurrent systems. In: Welch P, Bakkers A (eds) Communicating process architectures. IOS Press, pp 1–14
[Mar96] Martin J (1996) The design and construction of deadlock-free concurrent systems. PhD Thesis, University of Buckingham
[McC92] McCune W (1992) Experiments with discimination-tree indexing and path indexiing for term retrieval. J Autom Reason 9(2):147–167
[RGG95] Roscoe B, Gardiner P, Goldsmith M, Hulance J, Jackson D, Scattergood J (1995) Hierarchical compression for model checking CSP or how to check 10^{20} dining philosophers for deadlock. First TACAS in LNCS Springer, 1019(1)
[Ros94] Roscoe B (1995) Model checking CSP in a classical mind: essays in honour of C. A. R. Hoare. In: International series in computer science, Chap. 21. Prentice-Hall, Englewood Cliffs, pp 353–378
[Ros97] Roscoe B (1997) The theory and practice of concurrency. International Series in CS. Prentice-Hall, Englewood Cliffs
[RSR01] Ryan P, Schneider S, Roscoe B, Goldsmith M, Lowe G (2001) Modelling and analysis of security protocols. Addison Wesley, Reading
[Sca98] Scattergood J (1998) The semantics and implementation of machine readable CSP. PhD Thesis, Oxford University, The Queen's College
[Sri05] Srivatanakul T (2005) Security analysis with deviational techniques. PhD Thesis, University of York
[TCLTK] Tcl/Tk: Tool command language, 2006. http://www.tcl.tk/
[TGV] The test sequence generator TGV. http://www-verimag.imag.fr/~async/TGV
[Val90] Valmari A (1990) A stubborn attack on state explosion. In: Proceedings of 2nd international conference in computer-aided verification. LNCS 531. Springer, Heidelberg, pp 156–165
[WoD96] Woodcock J, Davies J (1996) Using Z: Specification, refinement, and proof. International series in computer science. Prentice-Hall, Englewood Cliffs

Received 15 January 2007
Accepted in revised form 23 February 2008 by B. K. Aichernig, E. A. Boiten, M. J. Butler, J. Derrick, L. Groves and C. B. Jones
Published online 3 April 2008