Taking our own medicine: applying the refinement calculus to state-rich refinement model checking

Leo Freitas, Ana Cavalcanti, and Jim Woodcock

Department of Computer Science
University of York, UK
{leo,alcc,jim}@cs.york.ac.uk

Abstract. In this paper, we advocate the use of formal specification and verification in software development for high-integrity and safety-critical systems, where mechanical proof plays a central role. In particular, we emphasise the crucial importance of applying verification in the development of formal verification tools themselves. We believe this approach is very useful to increase the levels of confidence and integrity of tools that are built to find bugs based on formally specified models. This follows the trend set out by a grand challenge in computer research for verified software.
In this direction, we present our experiences on a case study on the development process of a refinement model checking tool for *Circus*, a concurrent refinement language that combines Z, CSP, guarded commands, and the refinement calculus, with the Unifying Theories of Programming of Hoare and He as the theoretical background.
Keywords: model checking, theorem proving, formal verification.

1 Introduction

Increased complexity in hardware and software systems has created a demand for precision and reliability, particularly in the high-integrity and safety-critical domains [2]. One effective way of achieving this goal is through the use of formal specification and verification. When it comes to the development of formal tools, which ultimately will perform such verification, we see the use of formalism as essential. The same principles apply for critical systems.

A well-known programming technique is stepwise development through refinement, where correctness is guaranteed by construction. That is, starting from an abstract specification, the system is formally developed by the application of refinement laws that transform its representation to an artifact closer to a computer implementation (or program), where the properties of the specification are preserved, provided that generated proof obligations are discharged.

Specifications, intermediate designs, and concrete implementations are usually represented as mathematical models, where we are mostly interested in their data and behavioural aspects. The refinement laws enabling correct transformations are part of a refinement calculus of sequential programs [1, 15].

The problem with applying a refinement calculus is that it is a laborious task, and rigorous proof is needed. When the complexity and number of operations involved is high, the proofs become error prone, painstakingly long, and some sort of tool support would be very helpful, if not an imperative factor.

In this paper, we present our experience in using a refinement calculus in the development process of a model checking tool, where we have used the Z/Eves theorem prover [21] to mechanise the proof obligations. With the mechanised application of the refinement calculus, we strengthened the claims for correctness of the model checker.

The most important results obtained by this work are: (i) mechanical discharge of proof obligations; (ii) hints or counterexamples for failed proofs, hence invaluable suggestions for possible amendments in pre or postconditions, or loop invariants; (iii) hints about efficient use of Z/Eves that enables great reuse between different proof obligations, hence gained performance and productivity; and (iv) an informal strategy to translate Z and specification statements into JML [4], a modelling language that enables formal verification of Java code. This was developed while applying an extended version of the *Z Refinement Calculus* (ZRC) [5], the *Circus* refinement calculus [17], to an algorithm for refinement model checking of *Circus* [11, Chapter 4], which integrates model checking and theorem proving, in systems where data and behavioural aspects are combined.

The result of applying formal methods in the development of a model checker, was rewarding: since the encoding of the calculated algorithm, the code has not changed. Furthermore, as it is possible to (informally) translate these findings with some level of confidence to JML, we believe we have narrowed the gap between the concrete model of guarded commands and a Java implementation.

This sort of bootstrapping, where we have used *Circus* to specify and refine its own model checker, allows us to find early design flaws, as well as to ensure the algorithm is correct by construction. Thus, important properties, such as loop invariants, are precisely documented. That is, we took our own medicine to formally specify key aspects of the architecture, in order to enhance the consistency of the whole tool throughout the development process.

Whilst the model checker is still a prototype, various examples have been analysed and no bugs in the refinement algorithm have been found. We believe that for a formal verification tool, such an approach is essential for the assurance and credibility of any flaws they might find. This decision follows the trend set by a grand challenge in computer science research in verified software [2].

In the literature, there are few examples of such an approach, to the extent of our knowledge. The *Mural* theorem prover is one tool we know stepwise refinement was used throughout its development process [12] was used. Nevertheless, other tools, such as *Perfect Developer* [7], have applied verification of its own code after being developed.

In the next section, we present *Circus* and its refinement calculus by illustrating the application of some laws on simple examples. After that, Section 3 presents how we use Z/Eves to encode the proof obligations from the refinement

calculus. In Section 4, we present our case study. Finally, Section 5 presents conclusions and future work.

2 *Circus*

Circus is a concurrent language built for refinement, which allows the combination of different language paradigms [23]. It combines Z [22], CSP [9, 20], and specification statements found in refinement calculi. Other executable commands are also available, such as assignments, conditionals, and loops. This enables one to use *Circus* for both abstract specifications, intermediate designs, and actual code. Because its semantic model is based on the UTP [10], it is amenable for extension, and considerable work has already been done in this direction. The result is a unified programming language that can be used for developing concurrent programs through refinement.

Circus provides a refinement calculus that extends ZRC, hence we can formally specify and derive code not only for abstract data types, but also for concurrent programs. There is considerable effort in building a set of tools supporting the language. At the time of writing, there is a parser, a typechecker, a prototype refinement model checker, and the basis of a theorem prover.

One of the greatest challenges in combining different programming paradigms and notions of refinement is the provision of a suitable semantic model. The UTP model of *Circus* embeds all the features of Z and CSP. In this framework, we can guarantee that we can safely use both ZRC and CSP refinement laws, as well as new *Circus* laws.

In terms of data or behaviour dealt separately, one can apply either Z or CSP refinement laws. In situations where both paradigms cannot (or should not) be separated, new *Circus* refinement laws can be applied, and they are given in [17, App C]. This includes ZRC laws, CSP laws, and new *Circus* laws. For instance, using the *Circus* law (*C.141*) of interchange between alternation and guarded external choice, we could transform an alternation into an external choice.

For the implementation of a model checker for *Circus*, we face many challenges: (i) how to represent Z schemas without loosing their characteristic abstraction; (ii) how to represent predicate calculus finitely in order to allow model checking; (iii) how to model check behavioural and data aspects of systems; (iv) how to maximise the levels of automation, while combining model checking with theorem proving, whenever theorem proving is required; *etc.*

3 Refinement calculus automation strategy in Z/Eves

Firstly, from a *Circus* specification we apply the refinement strategy proposed in [6]. As *Circus* specifications involve specification statements, guarded commands, and Z and CSP operators, we need to find a way to represent these structures in a theorem prover in order to enable mechanisation. Luckily, proof obligations can be described as specification statements mentioning a frame of variables that can be updated, as well as pre and postconditions with predicate

calculus, hence we can use theorem provers for discharging proof obligations that would otherwise require to be done by hand. That is, we have used Z/Eves not to apply refinement laws, but to discharge the proof obligations these laws generate. A further step, would be the construction of a refinement calculation tool for *Circus*, such as Refine [16] for ZRC, where the application of laws and the transformations they represent could also be done by formal tools.

From the *Circus* specification of the model checker, we decided to apply the refinement calculus to the refinement model checking algorithm in order to get to code. This choice was made because we believe this to be the crucial part of the whole architecture. By using the refinement calculus to reach the code from the abstract specification, we ensure that the algorithm is correct by construction, and important properties such as loop invariants are precisely documented. Furthermore, other parts of the architecture have also been formalised in Z/Eves, but no refinement calculation was performed.

Firstly, from the Z aspect of the *Circus* specification, we needed to prove simulation between the abstract and concrete models, which include the state and related operations. This was achieved through simulation laws and corresponding applicability and correctness proof obligations. After that, with the concrete model at hand, we started applying ZRC laws to transform the Z part of the specification into guarded commands. Moreover, proof obligations coming from the application of CSP and *Circus* laws can be discharged similarly. In this process, various properties of interest were discovered and altered, such as state and loop invariants. As postconditions of concrete specifications tend to be quite complex, with predicates often involving schema inclusions, predicate simplifications were also often important. Thus, whenever stepwise refinement or formal verification was applied and proof obligations were generated, we see mechanisation via theorem proving as an essential requirement for correctness.

For different problems, one can follow a similar strategy. With an abstract specification either in *Circus*, if concurrent aspects are relevant, or pure Z, if only data is under concern, the same ideas for of applying the refinement calculus hold. In this way, the strategy can be reused to refine specifications down to code, and into JML annotation.

Mechanisation with Z/Eves means encoding the proof obligations using the Z schema calculus. For that, we apply the ZRC law of basic conversion (bC) backwards from specification statements to schemas, and then syntactically rearrange the corresponding schema so that it is amenable for mechanical proof in Z/Eves. Obviously, during the first iterations of stepwise refinement, where one starts from schemas and usually goes to specification statements, this is not very helpful. Nonetheless, later on when specification statements are to be refined to the most common guarded commands, such as alternations and assignments, this strategy pays off as it allows proof obligations to be mechanically discharged, like the ones in our case study in the next section.

For instance, assuming square root is well-defined in Z/Eves with signature as $sqrt \in \mathbb{N} \rightarrow \mathbb{N}$, where domain checks were discharged. Let us illustrate how we encode the proof obligation generated by the application of strengthening the

postcondition (in the example from [15, p. 5])

$$y : [\, 0 \leq x \leq 9,\ sqrt\, y = x \,] \overset{sP}{\sqsubseteq} y : [\, 0 \leq x \leq 9,\ sqrt\, y = x \land y \geq 0 \,]$$

provided that $\forall x, y, y' : \mathbb{Z} \bullet (0 \leq x \leq 9) \land (sqrt\, y' = x \land y' \geq 0) \bullet (sqrt\, y' = x)$

which is true based on properties of *sqrt* defined in Z/Eves as one expected *sqrt* to behave, bearing in mind specification/mechanisations issues. Firstly, we encode the state variables carefully according in the *Frame* schema. Next, we encode the pre and post conditions as schemas *Pre*, *Post*, and *NewPost* coming directly from the specification statement predicates (via bC backwards).

$$\begin{array}{lcl} Frame & \widehat{=} & [\, x, y, y' : \mathbb{Z} \,] \\ Pre & \widehat{=} & [\, Frame \mid 0 \leq x \leq 9 \,] \\ Post & \widehat{=} & [\, Frame \mid sqrt\, y' = x \,] \\ NewPost & \widehat{=} & Post \land [\, Frame \mid y' \geq 0 \,] \end{array}$$

As the precondition does not mention the after state, we include a read-only (Ξ) version of *Frame* in *Pre*. Finally, we can discharge the proof obligation by proving the conjecture *posP*1 as a theorem

theorem posP1
$\forall\, Frame \mid Pre \land Post \Rightarrow NeqPost$

The complete set of translation strategies for the various ZRC laws used throughout the formal derivation of the refinement algorithm code can be found in Section 6 of [11, App. A]. In this reference, we also include extensive information on how to drive Z/Eves, so that one can achieve efficient and acceptable (or higher) levels of automation.

The use of the schema calculus to represent the ZRC proof obligations makes the proofs concise, elegant, and easier to follow. For example, in our case study, due to the sheer number and complexity of predicates involved, it soon became impossible to handle the proofs reliably, as they would easily spread across two or more A4 pages of mathematical formulae. In spite of some auxiliary lemmas needed to improve automation in Z/Eves, the mechanised result was much more tidy, organised, elegant, and reliable than the alternative by hand.

Some conventions for Z/Eves

For every declaration that might include undefinedness, such as type inconsistencies or partial functions called outside their domain, Z/Eves introduces proof obligations as *Domain Checks*. These are sufficient conditions for definedness one needs to prove, even if the definitions involved are not being used.

For most declared functions that might become involved in future domain checks or proof obligations, some housekeeping theorems should be included. Although these housekeeping theorems are usually obvious, quite repetitive, and

straightforward to prove, they do increase the levels of automation to a great extent. For instance, we can axiomatically define a total function f that nondeterministically creates sequences of size n as

$$f : \mathbb{N} \to \mathrm{seq}\ \mathbb{N}$$
$$\forall\, n : \mathbb{N} \bullet \#\,(f\ n) = n$$

In this case, Z/Eves introduces a trivial domain check as the proof obligation

$$f \in \mathrm{seq}\,\mathbb{N} \land n \in \mathbb{N} \land s \in \mathrm{seq}\,\mathbb{N} \Rightarrow n \in \mathrm{dom}\ f \land f\ n \in \mathrm{dom}\ \#$$

Discharging this proof obligation is important in order to ensure we have not given a definition that might introduce inconsistencies in the model whenever f is used. If f is not used, one ends up proving more than what is necessary.

We also introduce additional facts about the function's domain, and result maximal types, to increase automation of other definitions that depend on f. In Z/Eves syntax, these facts are introduced as named conjectures to be proved as theorems, where the names are preceded by modifiers used for controlling automation granularity.

theorem grule gFMaxType
$f \in \mathbb{P}\,(\mathbb{Z} \times \mathbb{P}\,(\mathbb{Z} \times \mathbb{Z}))$

theorem grule gFRelMaxType
$f \in \mathbb{Z} \leftrightarrow \mathbb{P}\,(\mathbb{Z} \times \mathbb{Z}))$

theorem rule rFResultMaxType
$\forall\, n : \mathbb{N} \bullet f\ n \in \mathbb{P}\,(\mathbb{Z} \times \mathbb{Z})$

theorem rule rFIsTotal
$\forall\, n : \mathbb{N} \bullet n \in \mathrm{dom}\ f$

In Z/Eves, assumptions (*grule*) rules are used by every tactic that rewrites the goal, hence it enables coarse-grained automation for commonly needed type consistency checks that often appear in later proofs where f is used. On the other hand, rewriting (*rule*) rules are used by only a few specialised tactics, hence they enable fine-grained automation for more specialised scenarios.

As we use the schema calculus throughout our case study, and in the proof obligations from the refinement calculus, it is sometimes necessary to inform Z/Eves about obvious facts regarding the (maximal) types of the schema components, depending on which components are used later on. For instance, in a schema such as

$$S \mathrel{\widehat=} [\, n : \mathbb{N};\ s : \mathrm{seq}\,\mathbb{N} \mid \forall\, i : \mathrm{dom}\ s \bullet s\ i < n \,]$$

Z/Eves includes a domain check about the application of s to i that is easily discharged, since $i \in \mathrm{dom}\ s$. Depending on how S might appear, perhaps with

s being created using f, one needs to include additional information about the type of s with theorems, such as

theorem frule fSsMaxType
$\forall S \bullet s \in \mathbb{P}(\mathbb{Z} \times \mathbb{Z})$

This kind of theorem, among other reasons, is useful to avoid the need to always expand the schema definitions in order to discharge goals where s is involved. This "use without expansion" is the most useful tool for higher degrees of automation and modularity of proofs when complex schema inclusions occur. That is, because we can surgically guide specific aspects of the goal without the need to expand (possibly a great amount of) unrelated assumptions from included schemas. This kind of usage is defined as a forward rule (*frule*).

More details about Z/Eves are beyond the scope of this paper, and are omitted here for space constraints. An extensive tutorial including detailed information on how to precisely drive Z/Eves, with higher levels of automation for a variety of scenarios, can be found in Section 1.1.1 of [11, App. A].

4 Case study: witness search model checking algorithm

In this section we briefly present the *Circus* model checker architecture, detailing its refinement checking module. From this module, we include parts of the abstract model, parts of the sequential algorithm derivation via forward simulation, and the complete refined code of a sequential algorithm from the concrete model. Moreover, we discuss our findings and present some benchmarks of the whole project of the model checker tool.

Model checker architecture overview

The architecture of the *Circus* model checker is inspired by FDR, the refinement model checker for CSP [8, 18], and it has four components: (i) a parser, (ii) a typechecker, (iii) a compiler, and (iv) a refinement checker, as shown in Figure 1. From a *Circus* specification in LaTeX, the parser creates an *Abstract Syntax Tree* (AST) that the typechecker annotates with type information (AST_+). The compiler then transforms the annotated using the operational semantics of *Circus* into a labelled transition system with predicates embedded on the arcs (PTS). These automata are analysed by the witness search algorithm we present here in order to find possible flaws.

From this architecture, an automaton theory (for PTS), the operational semantics of *Circus*, and the refinement checker module have been formally defined as *Circus* specifications, and mechanical proof of properties and proof obligations have been carried out using Z/Eves.

The refinement checker module takes two compiled automata representing the *Circus* specification and implementation sides of the refinement order, together with a criterion (or level of detail) to perform the search. The refinement

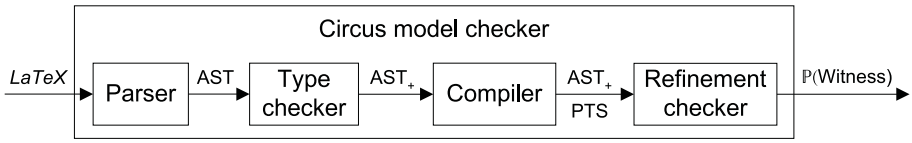

Fig. 1. *Circus* model checker architecture

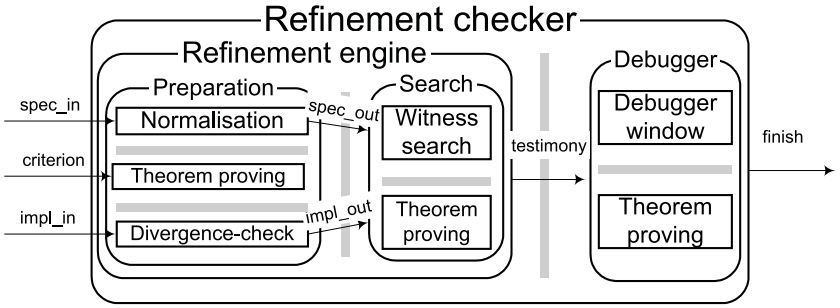

Fig. 2. *Circus* refinement checker

checker is defined in *Circus* by the parallel composition of various processes, as shown in Figure 2. For our case study, we detail the witness search process only. A full account of each of these processes, as well as the other components of the whole architecture is given in [11].

Witness search

Witness search establishes whether a specification S is refined by a design or implementation I, denoted by $S \sqsubseteq I$. If it does, a successful report is generated. If it does not, we provide sufficient debugging information that can be used to produce a suitable human-readable account of the failures as a set of witnesses. It works over PTS automata representing the state-rich aspects of *Circus*.

Witnesses are characterised as a nonempty joint path of node pairs coming from both automata, since a witness is the result of a search that found at least one incompatible node pair.

$$\begin{array}{lcl} JointPath & == & \{\, SNP : \mathrm{iseq}\ NodePair;\ SCL : \mathrm{seq}\ \mathbb{N} \mid \#SCL = \#SNP \,\} \\ Witness & == & JointPath \setminus \{\, (\langle\rangle, \langle\rangle) \,\} \end{array}$$

A joint path is formed by a pair of sequences, where the first element is an injective sequence of node pairs, and the second element is a sequence of layers

of the *Breadth First Search* performed. It enforces that both sequences must have the same size, hence both node pairs, and their corresponding search levels, are accessed at the same index. Injectivity of node pairs is important because it ensures no pairs are searched twice. Nevertheless, different pairs can searched at the same level. Search levels are important for memory efficient extraction of debugging information from witnesses.

Next, we define the conditions for a valid witness: (i) the last element of the node pair sequence of a witness (sN, iN) must be valid ($NodePairInv$), but no information about their compatibility is known, since it is the current pair being checked; (ii) on the other hand, every node pair in the *front* of a witness must be valid ($NodePairInv[dn/sN, n/iN]$) and compatible ($\neg GenVl[dn/sN, n/iN]$); (iii) there must exist a trace ($wtsTrace$) from both automata of S (normalised nf) and I (ip) corresponding to each node pair sequence that is part of a witness; and (iv) the search level of each node pair recorded strictly increases.

WitnessInv

$$
\begin{array}{l}
m : Criterion;\ nf : NFPTS;\ ip : IPTS;\ w : Witness;\ NodePairInv \\
\hline
(sN, iN) = last\,(w.1) \\
\forall\, dn : DNode;\ n : Node \mid (dn, n) \in \mathrm{ran}\,(front\ w.1) \bullet \\
\qquad NodePairInv[dn/sN, n/iN] \land \neg GenVl[dn/sN, n/iN] \\
\exists\, T : \mathrm{seq}\ \Sigma \bullet T = wtsTrace\,(nf, ip, w) \\
\forall\, i : 1 \mathinner{..} (\#\, w.2 - 1) \bullet w.2\,(i) \leq w.2\,(i+1)
\end{array}
$$

The existence of a trace in S (np) and I (ip) from the node pairs in the current witness (w) establishes the nodes that are mutually reachable, while searching for new successor pairs. That means, if one can create a valid non-empty sequence of node pairs from the two automata, then it must be possible to retrieve the unique trace related to such witness. The trace of events is unique because of the deterministic property of the normalised automaton of S. Finally, it ensures that lower level nodes must appear before higher level ones. As we do not store the whole trace a witness represents, these levels allow memory efficient representation of flaws. This consistency on the levels information is important for the debugger to provide accurate information while rebuilding the transition system from the failed pair up to the root of the search. Many of these properties were found due to failed proofs while mechanising.

The abstract model. Witness search is responsible for finding whether all the behaviours of I are allowable by at least one behaviour of S, such that they have a trace in common. The behaviours of interest depend on the selected criterion to establish refinement, which in turn has specific violation criterion. Due to space restrictions, we present only the relevant parts.

The general violation criterion is defined next. Regardless of the criterion being traces, nondeterminism, or divergences, every node pair from S and I must be valid ($NodePairInv$), and checked for traces violation ($TrVl$). For the traces criterion (tr), this is enough. Other different criteria, such as nondeterminism

(*sfl*), can also be checked for stable-failures violation (*SFlVl*). Finally, the divergence violations (*DvVl*) are checked only for the failures-divergences criterion (*fldv*). The violation of each criterion is defined as a Z schema that establishes the relationship between node pairs from the automata of S and I.

GenVl

$$m : Criterion;\ NodePairInv$$

$$TrVl \lor (\neg\, m = tr \land (SFlVl \lor (m = fldv \land DvVl)))$$

In this way, we separate concerns at the specification level.

The abstract state includes the refinement search parameters (*RSParams*), and the set of witnesses found (*wts*). The invariant of the abstract state (*RSState*) guarantees that: (i) the number of witnesses searched ($\#wts$) does not exceed the amount requested (*wr*); (ii) the automata involved after the transformations occurred during normalisation and divergence checking are valid with respect to the operational semantics (*enabled*) (see arrows in Figure 2); and (iii) witnesses that have been found, must satisfy the witness invariant (*WitnessInv*), and have the last node pair violating some compatibility criteria (*GenVl*).

RSState

$$RSParams;\ wts : \mathbb{P}\ Witness$$

$$wts \in \mathbb{F}\ Witness \land \#wts \leq wr$$
$$\forall\, sN : DNode;\ iN : Node;\ a : \mathbb{P}_1\ \Sigma \mid NodePairInv \land \neg\, GenVl \land a \in enabled\,(ip.ts, iN) \bullet \neg \bigcup (enabled\,(nf.ts, sN)) \cap a = \{\}$$
$$\forall\, w : Witness \mid w \in wts \bullet \exists\, sN : DNode;\ iN : Node \bullet WitnessInv \land GenVl$$

That is, for consistency, if a node pair $((sN, iN))$ is valid (*NodePairInv*), compatible ($\neg GenVl$), and has visible events ($a \neq \varnothing$) immediately available (*enabled*) in the implementation I (*ip*), then there must be some event in common with the normalised specification S (*nf*). Otherwise, either the operational semantics, or the model checking compatibility criteria, would have been wrongly specified. These consistency elucidations are due to mechanical proof.

Next is the signature of refinement search operations. It establishes that the search parameters that are part of the state do not change (Ξ), and that the set of witnesses (*wts*) may increase (to wts'), but previously found witnesses are not lost ($wts \subseteq wts'$).

RSOps

$$\Xi\, RSParams;\ \Delta\, RSState$$

$$wts \subseteq wts'$$

In the general violation criterion (*GenVl*), we factor the searching for witnesses with respect to each violation criterion. This allows a modular combination of

criteria within the different aspects of the compatibility check. Thus, for each criterion, we define an operation to search for witnesses. The set of witnesses found (wts') must be a subset of the set containing all valid witnesses (w) according to the witness invariant ($WitnessInv$), and related violation criteria for traces ($TrVl$) for a pair of nodes coming from the specification (sN) and the implementation (iN) automata.

$$
\begin{array}{|l}
\hline
TrWtsSearch \\
RSOps \\
\hline
m = tr \land wts' \subseteq \{\, w : Witness;\ sN : DNode;\ iN : Node \mid \\
\qquad\qquad WitnessInv \land TrVl \bullet w \,\} \\
\hline
\end{array}
$$

As we do not need to necessarily find all witnesses, but a specific number requested (wr), the value of wts' is a subset of, rather than equal to, the entire space of witnesses. Similarly, for the other criteria, each violation schema is disjoined to form the other sets of witnesses

$$
\begin{array}{ll}
WitnessInv \land (TrVl \lor SFlVl) & \text{for stable-failures} \\
WitnessInv \land (TrVl \lor SFlVl \lor DvVl) & \text{for failures-divergences}
\end{array}
$$

Finally, we define a total operation for finding witnesses, regardless of the criteria, as the disjunction of the witness search operations.

$$FindWitnesses \ \widehat{=}\ (TrWtsSearch \lor SFlWtsSearch \lor FlDvWtsSearch)$$

This modular approach gives room for future extensions in a precise fashion. For instance, one could encode one of the extended failures models for CSP defined in [3] as an additional violation schema, with the corresponding criterion flag and schema characterising the space of witnesses to search for. Moreover, for all available operations in the abstract model, we have proved applicability theorems about the operation preconditions.

The concrete model. We applied (a trivial) data refinement over the state of the abstract model ($RSState$), so that the concrete model has additional components: (i) two injective sequences for pending (pd), and already checked node pairs (ck); (ii) the node pair (wnp) currently being searched; (iii) a sequence of working levels used to register at which level of the search each (working) node pair appeared; and (iv) the abstract refinement search parameters. The sequential state invariant gathers properties about the variables (pd, ck, lvl, wnp, $swts$) used in the algorithm's code, instead of scattered in postconditions of later specification statements. This decision was taken in order to minimise and modularise the complexity of proof obligations generated, as the first s (harder) attempt to discharge the proof obligations when predicates were scattered showed. The first predicates are about finiteness of witnesses, and an equivalence for wnp used for better automation. Next we have the number witness we can search

($\#swts \leq wr$). The consistency between progress of node pairs from the implementation (ip) and the normal form (nf) as defined by the operational semantics ($enabled$) comes next, and it is similar to the abstract state ($RSState$), but mentioning the working node pair (wnp). A series of predicates establishing that the working node pair, and the pending and checking sequences elements are valid in the product automata (PA), are also included.

SeqRSState

$$
\begin{array}{l}
RSParams;\ swts : \mathbb{P}\ Witness;\ ck, pd : \mathrm{iseq}\ NodePair;\ lvl : \mathrm{seq}\ \mathbb{N} \\
wnp : NodePair;\ wsN : DNode;\ wiN : Node;\ wl : \mathbb{N} \\
\hline
swts \in \mathbb{F}\ Witness \land wnp = (wsN, wiN) \land \#swts \leq wr \\
\forall\, a : Arc \mid \neg\, GenVl[wsN/sN, wiN/iN] \land \neg a = \{\} \land \\
\quad a \in enabled\,(ip.ts, wiN) \bullet \neg \bigcup (enabled\,(nf.ts, wsN)) \cap a = \{\} \\
\#ck = \#lvl \land wnp \in PA\,(nf, ip) \land NodePairInv[wsN/sN, wiN/iN] \\
ck \in \mathrm{iseq}\,(PA\,(nf, ip)) \land pd \in \mathrm{iseq}\,(PA\,(nf, ip)) \\
\forall\, sNck : DNode;\ iNck : Node \mid (sNck, iNck) \in \mathrm{ran}\ ck \bullet \\
\quad NodePairInv[sNck/sN, iNck/iN] \land \neg GenVl[sNck/sN, iNck/iN] \\
\forall\, sNpd : DNode;\ iNpd : Node \mid (sNpd, iNpd) \in \mathrm{ran}\ pd \bullet \\
\qquad NodePairInv[sNpd/sN, iNpd/iN] \\
\mathrm{ran}\ pd \cap \mathrm{ran}\ ck = \{\} \\
\forall\, i : 1 \,..\, (\#lvl - 1) \bullet lvl\,(i) \leq lvl\,(i+1) \\
\forall\, j : 1 \,..\, \#lvl \bullet lvl\,(j) \leq wl \\
\forall\, w : Witness \mid w \in swts \bullet \mathrm{ran}\ pd \cap \mathrm{ran}\ w.1 = \{\} \\
\forall\, w : Witness \mid w \in swts \bullet WitnessInv[wsN/sN, wiN/iN] \land \\
\qquad GenVl[wsN/sN, wiN/iN]
\end{array}
$$

Next, comes the property that pending and checked pairs are disjoint, hence the search is closed under the elements of these sequences. This is important to establish the main loop variant, and hence guarantee that the whole search terminates. Finally, we include a series of properties regarding search levels useful for debugging, together with information about how witnesses relate to pending and checked pairs, which are further detailed latter. Many of these were discovered through formal proof.

This is possible by the application of forward simulation rules with a quite trivial retrieve relation: the set of witness from the abstract world equals the set of witnesses used in the concrete world. Thus, we have an operational refinement, rather than data refinement. At first we have used an injective sequence to represent $swts$, and the retrieve schema as $wts = \mathrm{ran}\ swts$, but it increased the complexity of the proofs in a great extent, because the Z toolkit does not have great automation for this data type. Fortunately, this was not a problem for the proof obligations related to the injective sequence of pending and checked node pairs. Finally, as Java and JML support sets, this choice did not become an implementation issue.

At this stage, in order to establish refinement, we needed to prove that, for every available operations of the abstract model, the corresponding concrete

version satisfies the two proof obligations of applicability and correctness generated [17, Law C.4]. In particular, we have done this for the entire *Circus* specification. In here we want to emphasise that the refinement algorithm simulates the abstract specification.

$$FindWitnesses \preceq SeqWitnesses$$

Finally, Like in the abstract model, we calculate the preconditions of all concrete operations to ensure their applicability as well.

The algorithm. It defines how node pairs are checked for compatibility, as well as how new pairs are found. To give an overview of the algorithm we provide the entire derived code in Figure 3, which is written in *Circus*. This code has been derived using ZRC, and action refinement laws for *Circus* [17, Appendix C]. Although our algorithm is similar to the algorithm of FDR presented in [19, 14], the mechanised proof effort precisely exposed loop invariants, and a great amount of hidden information that is interesting for the understanding of the witness search problem for refinement model checking in general.

The algorithm is divided into two stages: (i) compatibility check; and (ii) successor node pairs search. In the compatibility check, we first assign to the working node pair (wnp), update the pending pairs (pd), and increment the working level of the search accordingly. If wnp is incompatible ($GenVl$), then it must be included as a new witness in $swts$. It is formed by the previous checked pairs, together with the offending working node pair at the working level. Otherwise, if wnp is compatible, then the sequence of checked pairs and search level are updated likewise, and the next stage of finding successor pairs starts. The search is performed while there are pending pairs ($pd \neq \langle\rangle$) to be searched, and witness to be found ($\#swts < wr$).

While searching for successors, the arcs immediately available for communication in the implementation are retrieved through the *enabled* function representing all events immediately available from a given node. For each of those arcs, one needs to progress appropriately in the automata of S and I, according to the loop invariant. In order to exhaust all enabled implementation arcs ($arcS$), we choose the specification node successor (sN), and select all available implementation successor nodes ($iN \in iNS$) on the same arc. If it is a silent or internal transition, here specified as an empty arc, it represents nondeterminism (from an internal choice, for instance) being resolved in I. Since after normalisation the automaton of S is deterministic and has no silent transitions left, there is no successor node for S in this case. Otherwise, in the case of visible communication, the selection of successors follows from the *arcStep* function. It determines the set of nodes we can reach through a given arc at a particular node. These two functions represent the formally specified operational semantics of *Circus* [11, Chapter 3].

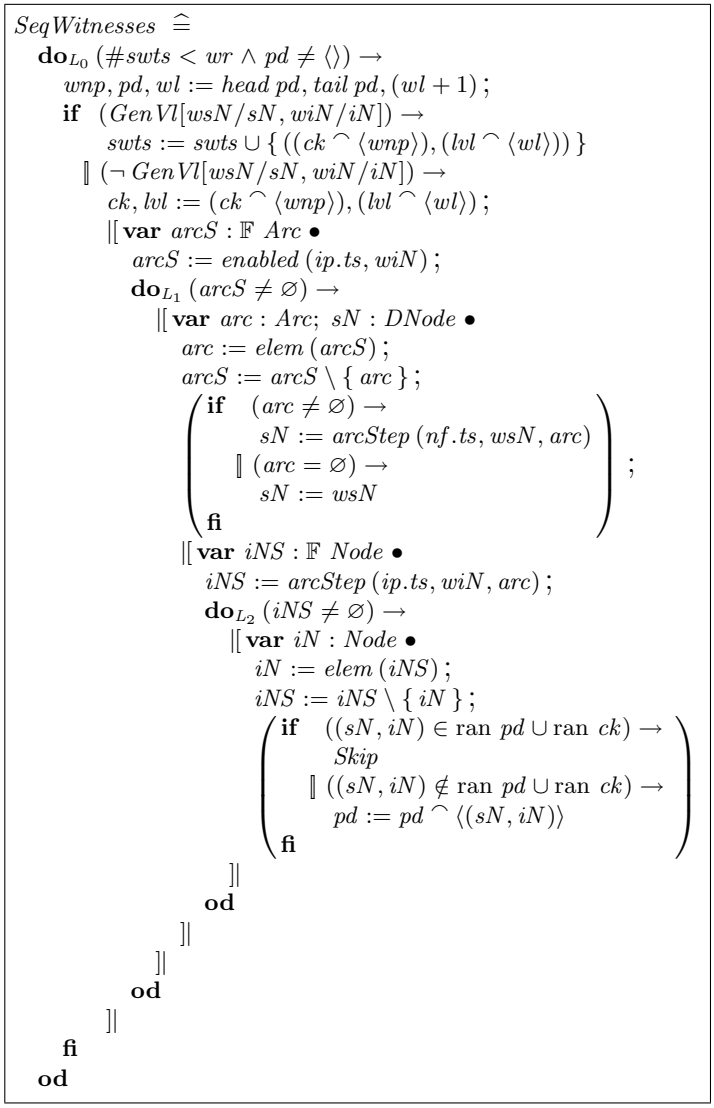

Fig. 3. Sequential witness search algorithm

Interesting properties we have discovered

Although this way of building up the witness from the sequence of checked pairs comes from FDR's algorithm, some properties to enable us to derive the code are not documented, to the extent of our knowledge. In FDR's algorithm description [19], it is mentioned that the elements of pd and ck are disjoint. Because of the mechanisation of proof obligations, these well-known and some other facts must be formally specified. For instance, we need to precisely include obvious facts not mentioned in [19], such as: (i) all node pairs in ck and pd are valid (or are part of) the automaton of S and I; (ii) all node pairs in ck are compatible in the chosen model ($GenVl$); (iii) node pairs from ck and pd can only come from the product automata of S and I, and not any valid node pair from other automata; and so on.

There are, however, some not entirely obvious facts as well. They must be clearly stated, otherwise the proof obligations cannot be mechanically discharged. We see this as a very interesting contribution to the field of refinement model checking. These facts are mostly related to the normalisation of S that occurs at the preparation process (see Figure 2), the various relationships between the witnesses found and the data structures used in the sequential search, and about loop invariants.

Normalisation properties. During normalisation, the automaton of S is transformed to become deterministic and free of silent transitions. Among other reasons, this is useful because it makes the sequence of node pairs unique, and hence it enables memory-efficient representation of the search space without compromising its results. Nonetheless, as witness search and normalisation are independent *Circus* processes, we must record that the automata received were built by the operational semantics. Another example is that, since the normal form of S is a deterministic automaton, and elements of pd and ck are disjoint, when the search finishes, the union of elements from pd and ck must be the size of the product automata of S and I. In this way, we ensure that all node pairs are checked, hence a precise characterisation of search exhaustiveness is given.

Witness properties related to the sequential state. As pd and ck belong to the product automata of S and I, and the normalisation guarantees the search paths to be unique, node pairs from witnesses already found can never appear as pending. Furthermore, valid node pairs in the product automata that have not yet being searched (*i.e.,* they are neither pending nor checked), can never be part of any witnesses that have already been found. These facts are included in the last predicates of *SeqRSState*.

Properties of the main loop. Let us explain the invariant, guard, and variant of each labelled loop from Figure 3. With application of appropriate laws and further simplifications, the main loop (L_0) invariant is reduced to the sequential state invariant already presented in schema *SeqRSState*. That is not surprising as we moved the algorithm main variables to the state on purpose at the beginning,

in order to have the main loop invariant clear from the state invariant itself. This is crucial for concentrating the proof effort at one hard/difficult point, whereas the remaining proofs become simpler. The main loop guard defines the termination condition for the algorithm as

$$\#swts < wr \land \neg pd = \langle\rangle$$

which means that either enough witnesses have been found, or there are no more pending pairs to be checked. It has been previously introduced by strengthening the postcondition right after initialisation of the corresponding variables via assignment. The main loop variant is defined as

$$(PS\,(nf, ip) - \#ck) + (wr - \#swts)$$

because only ck or $swts$ will increase at each iteration but not both, as every valid node pair being searched is either compatible or not. As a loop variant is an integer expression whose value is strictly decreased by the loop body, we need to find the boundaries for both ck and $swts$. The checking sequence is bound by the product size of both automata ($PS\,(nf, ip) \in \mathbb{N}_1$), as we can never check more than what is available, whereas the set of witnesses is bound by the number requested on wr. Moreover, pd cannot be used in the variant because it may not vary at every iteration.

Properties of loop L_1. The next loop encodes the search for successor pairs from a compatible node pair. As each arc from the set of enabled arcs ($enabled$) is being explored, we need to establish via the assignment that the following properties about arcs hold

$$arcS \subseteq enabled\,(ip.ts, wiN) \land \\ (\forall\, a : Arc \mid a \in arcS \bullet \neg arcStep\,(ip.ts, wiN, a) = \varnothing\,)$$

That is, subset containment with respect to the operational semantics ($enabled$) guarantees that exploring new arcs ($arcS$) preserves the amount remaining to be searched, and valid normal form nodes have no silent transitions. This forms part of the loop invariant. Moreover, after the assignment on $arcS$, we also establish the new properties about a compatible working node pair (wnp)

$$\neg wnp \in \text{ran } pd \land wnp \in \text{ran } ck \land \\ (\forall\, w : Witness \mid w \in swts \bullet \neg\, wnp \in \text{ran } w.1)$$

That is, wnp is not pending, has already been checked, and cannot be part of any witnesses previously found. This is also important for re-establishing the main loop invariant, as well as make both loops L_0 and L_1 work. They are dischargeable because the normal form is unique, the injective sequences (pd and ck) are disjoint, and because of the witness properties related to the sequential state and witness invariant mentioned above. Finally, the invariant of L_1 is given in four parts: (i) the guard from the alternation ensuring the working node pair

is compatible ($\neg\, GenVl[wsN/sN, wiN/iN]$); (ii) a simplified version of the state invariant ($SeqRSState$) with information about well-formed witnesses removed, as to search for successors it is irrelevant; (iii) the new properties of the working node pair after the update of ck; and (iv) the properties of $arcS$ just mentioned. Also, since we use the cardinality of $arcS$ as the variant, $arcS$ must be finite. The loop guard is given as ($arcS \neq \varnothing$), and the variant is $\#\, arcS$.

Properties of loop L_2. The final part of the algorithm is the possible inclusion of new successor pairs as pending, whenever they have not been already checked. We need to iterate over the set of reachable implementation nodes (iNS) to form new node pairs (sN, iN), where the normal form node (sN) has already been fixed. Loop L_2 has the invariant of L_1 conjoined with the property about iNS

$$iNS \subseteq arcStep\,(ip.ts, wiN, arc)$$

which is dischargeable as the nodes in iNS are reaches from $arcS$ *enabled* by the operational semantics. Finally, the guard of L_2 is ($iNS \neq \varnothing$), whereas the variant is $\#\, iNS$.

These, and other properties that were found throughout the mechanical formalisation process, have proved the whole idea of applying formal methods in the development of formal tools worthwhile for this case study. Although mechanisation can incur some burden and time constraints, in the longer run, we believe it to be indispensable in discovering information that is crucial for correctness, and a better understanding of the problem at hand.

Translation into JML

At last, we translate the various predicates representing different properties of the algorithm into JML notation. They appear as comments in the Java code that implements the algorithm in Figure 3.

As the proof obligations normally come from specification statements, it is usually straightforward to translate, because JML allows pre and postconditions on methods as special predicates directly, as the *requires* and *ensures* clauses of JML annotations, respectively. Similarly, the JML *assignable* clause is a direct representation of the specification statement frame.

It is more challenging to translate the loop invariants. At the time when this case study was performed, the JML documentation and language support for encoding loop invariants was not as thorough as it is today, where the available annotations are better documented and supported by the JML tools. Because of that, we needed to provide an intermediate solution: the predicate's annotation and the algorithm's code were scattered into various methods of Java inner classes, so that the frame, pre and postconditions could be precisely specified at each different stage. Nonetheless, although this specifies/documents the problem precisely, it unfortunately complicates the code. Furthermore, the lack of Z toolkit definitions within the available JML data structures, such as injective sequences, also limited this translation effort altogether.

Some benchmarks

In total, the whole formalisation effort in the development of the *Circus* model checker is summarised in Table 1. It includes: (i) an extended Z toolkit to handle finiteness and injections better; (ii) the automata theory for *PTS*; (iii) the normalisation, divergence checking, refinement search, and debugger *Circus* specifications; and (iv) the refinement proofs for the derivation of the sequential witness search algorithm. The complete process took one person working full-

Formal item	Ext. Z toolkit	PTS theory	Normal form	Div. check	Ref. search	Debugger	Total
Abbrev.	2	15	0	0	3	2	**22**
Given sets	0	2	0	0	0	1	**3**
Free types	0	2	0	0	4	0	**6**
Ax. defs.	0	16	6	0	6	1	**29**
Gen. defs.	10	7	0	0	0	0	**17**
Schemas	0	8	4	6	108	8	**134**
Z/Eves rules	81	191	18	5	90	12	**397**
Lemmas	14	24	0	0	73	0	**111**
Theorems	11	44	5	1	25	1	**87**
Proof scripts	103	259	23	6	214	1	**606**
Domain checks	3	25	7	0	43	0	**78**
Channels	—	—	3	3	6	2	**14**
Actions	—	—	3	7	25	9	**44**
Variables	—	—	0	0	14	1	**15**
Total	**224**	**593**	**69**	**28**	**611**	**38**	**1563**

Table 1. Summary of formal declarations for *Circus* model checker

time for around one whole year. For the algorithm derivation alone, we applied around 101 refinement laws, which generated 42 proof obligations, including: (i) 14 trivial proofs discharged directly by Z/Eves; (ii) 12 easy proofs with (possibly lengthy) straightforward manipulations; (iii) 10 hard proofs usually depending on case analysis and Z/Eves rules; and (iv) 6 difficult proofs that exposed most of the inconsistencies in the automata theory, and in the formal definitions.

Although the number of Z/Eves automation theorems is high, they are repetitive and straightforward to prove. Also, many of the given theorems are in fact specification statements and proof obligations encoded as schemas by our automation strategy. To the extent of our knowledge, this code derivation is the biggest case study in the application of ZRC, and one of the few related to the development of a formal tool.

5 Conclusion

We expect the experiences shown in this case study to motivate the application of formal specification and verification, both in theory and practice, for tools aimed at formal verification, as well as computer systems in general. We advocate the use of mechanical proof throughout formal specification and verification.

We believe that formalisation plays a crucial role in increasing the integrity levels of the model checker through a combination of techniques. Together with the refinement algorithm and the architecture, the operational semantics and the underlying automata theory are also formally defined. Throughout the development process, Z/Eves was used to discharge proof obligations from the algorithm derivation, animate the operational semantics, prove properties of the theory of automata, and so on. In this process we presented a recipe showing how to use a theorem prover to discharge proof obligations generated by the application of the *Circus* refinement calculus.

This sort of bootstrapping, where we have used *Circus* to specify and refine its own model checker, allows us to find early design flaws, as well as to ensure the algorithm is correct by construction. For instance, the various properties about witnesses, pending, and checking sequences, enabled us to properly understand why some witnesses could not be properly interpreted by the debugger process due to lack of information. Moreover, to bridge the gap between formal specification and actual code, we use JML annotations to document our findings at the level of the Java code. Thus, important properties, such as loop invariants, are precisely documented and amenable to further verification. This exercise of taking our own medicine shows how one can go from an abstract formal specification to code mechanically, hence gathering the knowledge to step forward on the roadmap for building formal verification tools formally. In this process, we found not only bugs, but also unkonwn/undocumented important properties.

Foreseeable extensions to the work are a formal derivation from the abstract specification of witness search to a parallel refinement model checking algorithm. Apart from concurrency complexity, there is a further burden while integrating theorem proving and model checking in a parallel setting, such as dependencies between their results. Another interesting work is to extend the JML type system to include most of the Z toolkit, hence enabling more Z specifications to be translated and analysed by JML tools. At this point, an automated translation tool could be created, in the spirit of another tool that already partially converts B to JML [13].

References

[1] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Text in Computer Science. Springer-Verlag, 1998.

[2] 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.

[3] Christie Bolton and Gavin Lowe. A Hierachy of Failures-Based Models: Theory and Application. *Theoretical Computer Science Journal*, June 2004.
[4] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, Electronic Notes in Theoretical Computer Science, pages 73–89. University of Nijmegen, Elsevier, March 2003.
[5] A. L. C. Cavalcanti and J. C. P. Woodcock. *ZRC—A Refinement Calculus for Z*. Formal Aspects of Computing Journal, 10(3):267–289, 1999.
[6] A. L. C. Cavalcanti and A. C. A. Sampaio and J. C. P. Woodcock. *A Refinement Strategy for Circus*. Formal Aspects of Computing Journal, 15(2-3):267–289, 2003.
[7] Escher Technologies. *Perferct Developer User's Guide, v.3.0, 2004*. Available online at `www.eschertech.com/product_documentation/UserGuide.htm`
[8] Michael Goldsmith. *FDR2 User's Manual version 2.82*. Formal Systems (Europe) Ltd., June 2005.
[9] C. A. R. Hoare. *Communicating Sequential Process*. International Series in Computer Science. Prentice-Hall, 1985.
[10] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. International Series in Computer Science. Prentice-Hall, 1998.
[11] Leonardo Freitas. *Model Checking Circus*. PhD thesis, Univeristy of York, October 2005.
[12] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. **Mural***: a Formal Development Support System*. Springer-Verlang, 1991. ISBN: 3-540-19651-X.
[13] Petra Malik and Mark Utting. CZT: A Framework for Z Tools. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B: 4th International Conference of B and Z Users, Guildford, UK*, pages 13–15. Springer-Verlag, April 2005.
[14] Jeremy M. R. Martin and Yvonne Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. *Communicating Process Architectures*, 2000.
[15] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1994.
[16] M. V.M. Oliveira and M. A. Xavier and A. L. C. Cavalcanti. *Refine and Gabriel: Support for Refinement and Tactics*. In J. R. Cuellar and Z. Liu editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310–319. IEEE Computer Society Press, 2004.
[17] Marcel Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, University of York, 2006.
[18] Peter Ryan, Steve Schneider, Bill Roscoe, Michael Goldsmith, and Gave Lowe. *Modelling and Analysis of Security Protocols*. Addison Wesley, 2001.
[19] 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.
[20] A. W. Roscoe. *The Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1997.
[21] Mark Saaltink. *Z/Eves 2.0 User's Guide*. ORA Canada, 1999.
[22] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice-Hall, 1996.
[23] J. C. P. Woodcock and A. L. C. Cavalcanti. *The Semantics of Circus*. In D. Bert and J. P. Bowen and M. C. Henson and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, number 2272 in Lecture Notes in Computer Science, pages 184–203, Springer-Verlag, 2002.