Innovations Syst Softw Eng (2006)
DOI 10.1007/s11334-006-0021-9

ORIGINAL PAPER

Leo Freitas · Jim Woodcock · Ana Cavalcanti

State-rich model checking

Received: 21 October 2005 / Accepted: 8 December 2005 / Published online: ■■ 2006

Abstract In this paper we survey the area of formal verification techniques, with emphasis on model checking due to its wide acceptance by both academia and industry. The major approaches and their characteristics are presented, together with the main problems faced while trying to apply them. With the increased complexity of systems, as well as interest in software correctness, the demand for more powerful automatic techniques is pushing the theories and tools towards integration. We discuss the state of the art in combining formal methods tools, mainly model checking with theorem proving and abstract interpretation. In particular, we present our own recent contribution on an approach to integrate model checking and theorem proving to handle state-rich systems specified using a combination of Z and CSP.

Keywords Model checking · Theorem proving · Abstract interpretation · Formal method tools

1 Formal verification methods

The growing complexity of new hardware technologies and software systems inevitably increases the complexity in the functionality of these artifacts. When more complex functionality becomes necessary, the likelihood of subtle errors increases, pushing the demand for development techniques that foster reliability and safety [20].

The role of formal methods in computer science is analogous to the role of mathematical models and calculation in traditional engineering; see Table 1, which is based on information from [20]. Calculation can be used to predict the behaviour of a design while interacting with its environment. These calculations can be carried out in a series of steps towards the final implementation, guaranteeing the implementation of the original desired properties. This method creates programs that are correct by construction; it is known as stepwise refinement.

Table 1 Analogies of formal methods with respect to other fields

Field	Classical physics	Computing	Engineering
Example	Dynamics theory	Security protocol	Building
Design	Expressiveness	Message contents	Shape of walls
	Precision	Order of delivery	Placement of doors
Environment	Vacuum	Communication	Resistance of materials
	Gravity forces	Concurrency	Properties of fluids
Calculation	Differential calculus	Formal verification	Structural calculus
	Particle acceleration	Automated deduction	Differential equations

There are many kinds of computer systems and many corresponding ways to model and design them. Sequential programs are expected to terminate, producing some output from some input values. On the other hand, the behaviour of reactive and concurrent systems can be observed or altered at intermediate stable states; they are normally formed by several components. Reactive systems are expected to interact with an environment, exchanging data through communication, where the environment is normally another system component, or the final user. This interaction often occurs concurrently, and is expected to remain stable and active indefinitely.

For sequential systems, the enumeration of behaviours is not too hard to achieve. By selecting suitable test data, it is possible to check desired properties through direct execution. With sequential programs, it is fairly acceptable to reach a level of stability of a design without exploring all possible behaviours, despite the fact that this does not guarantee correctness.

For concurrent or reactive systems, however, due to the complexity of the interaction of the components, this enumeration must be done in a more reliable approach. A real explosion in behaviour complexity occurs when the components

Leo Freitas (✉) · Jim Woodcock · Ana Cavalcanti
Department of Computer Science,
University of York, York, UK
E-mail: leo@cs.york.ac.uk, jim@cs.york.ac.uk, alcc@cs.york.ac.uk

interact concurrently with each other or with their environment. In order to understand designs and to predict their properties, some way to comprehend all these behaviours is necessary [91].

Different kinds of systems demand different degrees of correctness. For instance, it might be acceptable that a spreadsheet program fails to print large files once in a while. Reactive systems, however, are normally related to critical functionality that must go neither wrong nor unstable. For example, a flight control system must not behave unexpectedly, and must always be responsive to the pilot and other components of an aircraft under all circumstances during a flight. Therefore, correct and predictable behaviour is essential.

The use of formal verification does not a priori guarantee correctness. Nevertheless, it can greatly increase the understanding of a system at early stages by revealing its inconsistencies, ambiguities, and incompleteness that would go unnoticed otherwise.

Formal methods can benefit system design in two different ways: (i) by providing mathematical concepts and notations to help reasoning development, and communication of ideas; and (ii) by exploring properties of our designs through mathematical modelling. Validating the faithfulness of the system and environment models, as well as the accuracy of calculations, are separate problems in formal verification [94].

So, to apply formal verification we need: (i) a mathematical model, (ii) a specification language, and (iii) a method of proof. If the language, the model, and the method of proof are properly represented, it is likely that formal verification techniques can be carried out automatically to a substantial extent.

A mathematical model must provide an expressive framework for capturing the behaviours of a system; its expressiveness should be sufficient to capture the design concerns. The framework is a theoretical account of what kinds of systems can be represented and analysed.

In order to apply any verification technique, the description of the system must be adequate. A specification language allows (possibly partial) characterisation of system behaviours through a comprehensible, concise, and unambiguous notation. It can be used to describe what can be computed by the system, instead of how this computation is carried out. A specification language enables the characterisation of the possible system inputs, and the environment assumptions that are necessary in order to guarantee correct behaviour.

A theory that predicts and calculates the observations (or behaviours) is required in order to make the model representing the specification meaningful. A method of proof is the theory that provides support for verification that specified properties are satisfied. The method of proof must be sound with respect to the mathematical model; otherwise, proof calculations could be in contradiction with the intended semantics by generating wrong conclusions. The verification method should also be complete: if there is a bug, then it should be found.

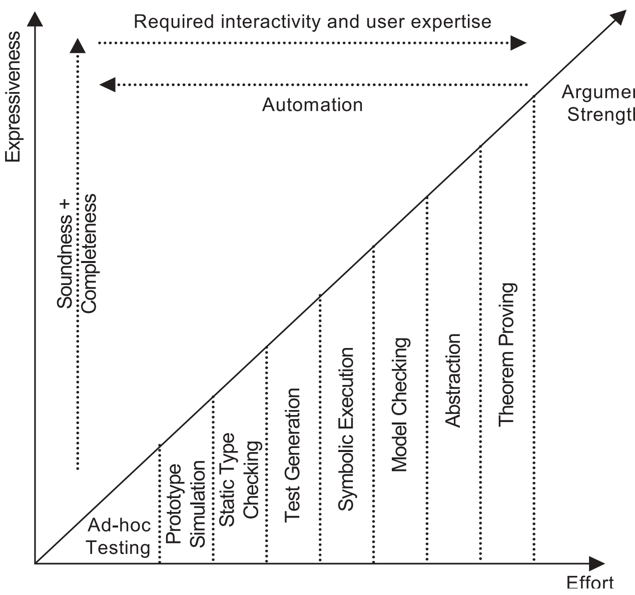

Fig. 1 Selecting formal verification techniques

Nevertheless, there are techniques that compromise soundness or completeness for higher-level automation, such as Java extended static checking with JML [Query1] [14], or Microsoft's device-driver analysis with SLAM [Query2] [65]. That is possible because their target domain of application is not safety-critical. Compromising soundness requires careful inspection of results as there can be false bugs. On the other hand, compromising completeness implies that a flawless check does not guarantee a flawless system, since the technique may not be able to identify existing bugs.

Two main concerns of formal verification are the readability of the obtained result, and the level of interaction with the user in order to achieve the desired goals of calculating, interpreting, and reasoning about the results. These elements vary according to the degree of complexity of the system, the kind of properties to verify, and the chosen verification method itself. The criteria for the selection of a method are based on two main factors: effort and expressiveness; see Fig. 1, which is based on a similar one in [100]. The account of the effort is measured by the amount of interaction required from the user. The expressiveness is measured by the variety of properties and systems one can check. The reward is analysed most often with respect to the strength of the argument claimed regarding soundness and completeness.

For example, on one side of the graph one can find ad hoc tests on system prototypes, which do not give a good reward: there is no guarantee of either system correctness or reliability. At the other extreme, mechanical proof via theorem proving of a formally specified design can be used to guarantee the satisfaction of desired properties, but at the expense of high expertise and interactivity from the user. In between come various (mostly finite-state) methods.

The last three techniques on the right aim at formal verification of properties, where correctness is the main concern.

The other techniques aim at refutation (or testing); they can be useful for finding problems, but not for guaranteeing that such problems will not happen [94].

There are, however, some tools and techniques that work with refutation by trying to find witnesses of failed property checks [45,73]. Such tools do not guarantee correctness, but still they help to find and eliminate flaws at early stages. This happens because they provide a strong argument in terms of soundness and correctness. Although completeness is not a major concern, the required degree of user expertise and interactivity is usually low.

The main advantages of model checking are its high level of automation, and the debugging information returned for reasoning purposes, provided enough resources are available. Model checking involves an exhaustive enumeration of behaviours, which are checked to establish whether they satisfy a given property of interest. If a behaviour that breaks the property is found, then it is used to provide debugging information.

Due to the exhaustive enumeration of behaviours, it is not always possible to model check infinite (or unbounded) systems due to the state explosion problem that usually occurs when different components of a reactive system interact. Clever (and compact) data structures that can represent (possibly infinite) behaviours efficiently and techniques to simplify models have been of primary interest. In any case, significant machine power in terms of processor, memory, and disk availability are needed for checking large system descriptions. Details on the relation between model checking and the techniques mentioned on the graph can be found in [21, Chap. 1], and [64, Chap. 1].

Deductive systems (or theorem proving) [52,92,96] use inference rules and axioms of a theory in order to prove correctness. One of its main advantages is the fact that it can handle verification of infinite-state systems. It is one of the most powerful techniques in use in formal verification. Nevertheless, it demands expertise from its user and is normally a time-consuming task; generally, no accurate limits or deadlines can be easily placed.

In this paper we discuss the state of the art in formal verification techniques. Special attention is given to combination of methodologies, where model checking plays a central role due to its importance in achieving correctness with higher levels of automation, and as few compromises in soundness and expressiveness as possible. This choice is based on major recent industry and academic interest.

In the next section, we define model checking, and discuss its variations and different interpretations. Next, in Sect. 3 we summarise the main approaches of classical and refinement model checking with examples of their application in industry. Sect. 4 discusses the state explosion problem. Afterwards, we present two state-of-the-art solutions for this problem in Sect. 5, where we focus on state-rich reactive systems that are modelled using a combination of languages to specify data and reactive behaviour. Finally, we summarise the discussions and draw our conclusions in Sect. 6.

2 What is model checking?

Model checking is a verification technique for finite-state systems (or components of systems), where reachable states of a program are systematically enumerated. If two of these systems interact, the final size of the combined state space can be the product of the original sizes, and this imposes restrictions on what kind of system can be modelled, what kind of property can be checked, and what kind of theory support is necessary [21,64,88].

The major advantages of model checking are its high level of automation, and the availability of precious debugging information when a failure is found. Highly automated tools are well accepted by users because they do not require much expertise from them in fields outside their own application domain. The debugging information returned is of high value in finding bugs, and reasoning about design flaws at early stages of the design.

The main challenge of model checking is to deal with the state explosion problem that occurs when components of a design interact concurrently. Such interactions can push the state space towards a number that is impossible to handle due to time and resource constraints. Therefore, in order to apply model checking, one needs to deal with this problem beforehand. That means finding a suitable and compact data structure to represent the model, together with an efficient search strategy in a setting that allows the highest levels of automation possible. This involves clever data-structure manipulations without compromising soundness [22,51], complexity analysis of involved search algorithms [63,104], reachability analysis between properties of the data structure with respect to the search algorithm [62], and so on. One successful approach used recently is the symbolic representation of the (possibly infinite or unbound) state space in order to represent richer models, which are frequently present in the design of software systems [34].

In practice, the model-checking approach for formal verification is not fully automatic, although it demands few user interactions when compared with other available approaches (see Fig. 1). For instance, some interaction from the user may be necessary in order to interpret the debugging information as counterexamples, or expertise in the tools themselves in order to properly fine-tune them and hence make checks meaningful (or even feasible). This leads to possible modifications on the original design. Hopefully, the user is asked to interpret information in their own application domain. This characteristic highly motivates users for selecting model checking as their verification methodology. Another task that often requires user interaction is to find a suitable finite-state representation of the system being modelled. Again, the solution must be given by the user through abstractions of the problem description (see Sect. 4).

In summary, any formal verification technique demands interaction from the user. The main advantage of model checking is the fact that the level of interaction is kept as low as possible. Moreover, the user is asked to interpret, and better understand the problems of its own design, inside their own application knowledge domain. This is a task the user is often more inclined to carry out.

3 Model-checking approaches

In the literature, the term *model checking* is almost always related to temporal logic; these works are known as traditional or classical model checking [20,21,64]. In [87], the term is interpreted in a different way; it means establishing an order between two different automata based on containment with respect to the properties they represent. This technique is known as refinement model checking. More details on automata theory can be found in [43]. Both approaches aim at formally checking properties of a design. What varies are the design notations and the way to represent and check desired properties.

In both approaches, the properties and the design are defined in a formal notation. There are three stages: (i) modelling, (ii) specification, and (iii) verification.

At the modelling stage, the design (or implementation) requirements are defined using a formal specification language accepted by the model checker. At this point, in order to deal with the state explosion problem, these requirements must be represented as a finite-state model, and as compactly as possible. Infinite-state (or unbound) representations must be abstracted in order to make their size feasible for model checking.

The specification stage defines the desired properties of the design. Once more, formal notation must be used. The choice of formal notation for both the requirements and properties, as well as the model-checking technique, sets the scene for what kind of properties can be checked, how they are checked, and what one can expect from those checks.

Finally, in the verification stage, the specified properties are verified against the model in order to guarantee correctness. The verification stage returns a meaningful answer to the user: either a successful report, or a counterexample. The former guarantees the claim for correctness. The latter is a very important debugging device used to reason about the failure of the property being checked. Many approaches for performing model checking are available. In what follows, we explain the most used in general terms.

We give a more detailed account of refinement model checking, as this approach has fewer references and no previous survey in the literature, to the extent of our knowledge.

3.1 Temporal logic

In this approach, the design is modelled using a language that is normally proprietary to the checking tool. The properties are defined using temporal logic formulae as assertion checks [60,61]. The verification is performed by exhaustively enumerating all the behaviours until a fixed point is reached [21,64], where variations of the traditional search algorithms exist [17,101]. In this context, model checking becomes simply a decision procedure. This led to its inclusion as part of some industrial-strength tools, such as satisfiability (SAT) solvers as part of the PVS [Query3] theorem prover [10,44].

This idea of using temporal properties for formal verification is of interest to the hardware design community. Hardware systems are finite-state and less dynamic compared to software modelling.

Temporal logic combines the usual propositional logic operators with *tense operators*, which are used to form statements about how conditions change in time. These operators can be used to state complex statements about the past, the present, or the future.

Different ways of interpreting such operators give rise to different versions of temporal logic. The properties of the operators are related to pre- and post-conditions of a Hoare triple [40], as well as a wide variety of temporal properties, such as responsiveness, safety, divergence, and so on. Each kind of logic can cover different aspects of desired properties according to the temporal order relation between different states in time.

When the temporal order is total, one is using linear time temporal logic (LTL). This is mainly applied to systems related to hardware design or security protocols [64, Chap. 1]. This kind of logic is also useful for modelling a *next-time* operator for synchronous digital hardware signalling [64, Chap. 2]. More details on a LTL model checker (called SPIN) can be found in [42].

If our interest is to model nondeterminism, such as the possibility of something happening, branching time temporal logic (CTL) is the appropriate choice. In this kind of temporal logic, the temporal order relation defines a tree which branches towards the future [64, Chap. 2]. More details on a CTL model checker (called NuSMV) can be found in [16]. The model of each different temporal logic is formally defined by a different set of axioms over the tense operators.

Clarke and Emerson coined the concept of *model checking* in the early 1980s [21]. It was first used for proving the validity of a property defined by a temporal logic formula against a finite-state model representing a design described using a specialised automaton called a *Kripke* structure. This allowed the proof procedure to be fully automatic, while maintaining the elegance of the formal specification. The model-checking algorithm proposed by Clarke and Emerson builds an automaton of the design from its formal specification in the source description language. The properties of the design to be checked are specified as temporal logic formulae. The truth of these formulae is established when the algorithm has successfully carried out an exhaustive search on the state space. Besides being fast and fully automatic, this technique produces a state sequence (or trace) when a formula being checked is $false$.

Since then, many other algorithms have been proposed in order to deal with the state explosion problem. The most successful one is based on the use of *ordered binary decision dia-*

grams (OBDDs) as the data structure to represent temporal logic formulae [12]. The use of OBDDs in a model checker is known in the literature as *symbolic model checking* [64]. OBDDs allowed industrial-scale checking on hardware design for the first time.

The symbolic approach is better suited for the validation of specific kinds of systems like digital hardware circuits and security protocols. This is because OBDDs are good at representing Boolean decisions, but not complex data types; they are tailored for checking very structured systems. In software modelling, typically, we have to handle asynchronous behaviour without a global clock, or complex data structures representing the system state. Model-checking software usually involves a much larger number of states to explore, making OBDDs lose their efficiency.

Another technique widely used that tries to minimise the state space is partial order reduction. This method explores the effect of the interdependence of concurrent events on the global state. Its application reduces the number of states to consider by interleaving components. In some cases, however, the addition of interleaving introduces nondeterminism and can actually increase the number of states. This technique itself is difficult to automate and tailored for very specific kinds of systems. More details can be found in [103], [21, Chap. 10], and [64, Chap. 9].

Due to the state explosion problem, a new trend is to use classical model checking itself in order to find suitable abstractions or approximated models to check properties of interest [101], and this has opened up a new field, as shown for example in [65]. We discuss abstraction and symbolic approaches below.

Examples in industry Bugs found in early stages of hardware design are a major cause of unexpected delays and maintenance problems. A successful case of using CTL and the model checker NuSMV to find failures in a circuit design is given in [13].

Symbolic model-checking algorithms have also been used to model-check the μ-calculus [49]. This is a logic that can express a variety of properties of transition systems, such as language containment, reachable state sets, state equivalence relations, and so forth. More details on classical model checking of properties described using the μ-calculus can be found in [64, Chap. 6], and [21, Chap. 7]. Other examples of the use of classical model checking related to cache coherence protocols can be found in [84].

Classical model checking is widely used, either with the support of a particular tool like NuSMV or SPIN, or as proof tactics or formulae SAT solvers within theorem provers like PVS. Furthermore, an alternative to BDDs [Query4] for symbolic model checking that uses SAT solvers is given in [11, 47, 106].

Further applications can be found. In [3] a CTL model checker that is based on exploiting modularity is presented. An example of the use of symmetry and induction to enable industrial-scale case studies is given in [19]. Hardware circuits were checked using CTL in [13]. The use of bounded model checking for refutation and verification is discussed in [72,73]. Communication protocols and control flow systems have been checked [29,39]. In [26], symbolic model checking with CTL is used for verification of VHDL [Query5] descriptions of the design of a RISC [Query6] microprocessor.

In the same direction of industrial scalability, new techniques are trying to use model checking itself to find suitable (model-checkable size) abstractions. The use of classical model checking itself as a way to simplify and abstract the problem domain can be found in [101]. A combination of data and behaviour in classical model-checking data structures to enable better description of software systems is explored in [17].

3.2 Automata theoretic methods

There is another approach to model checking based on automata theoretic methods [87]. The idea is to represent both the design and the properties as automata. In this case, the verification to be performed is characterised by some relation between these automata.

For example, this relation can be containment between the language each automaton represents. In other words, the automata represent all the system behaviours as sequences of a language. Therefore, if the specification automaton contains the language of the implementation automaton, then the implementation satisfies (or refines) the specification. More details about language containment over automata can be found in [22]. Related topics in automata theory itself used in model checking are also important. For instance, compression techniques useful for the automata theory used by the CSP model checker failures–divergences refinement (FDR) [Query7] are detailed in [90,103], while new ideas for automata theory to handle combination of techniques, such as theorem proving, and paradigms, such as data and behaviour, have been investigated in [33,104]. Other properties, such as nondeterminism and divergence, can also be expressed similarly via different notions of containment between automata [88, Chap. 8]. These notions of containment are discussed below.

This procedure of ordering automata via containment with respect to specific properties of interest can be performed several times in a stepwise development cycle that is very attractive for both software and hardware. Depending on the way one constructs and interprets the automata, different results can be observed and analysed. Each time, the design analysed in the previous cycle becomes a specification for a more concrete model. This follows the idea of refinement [7,68], where it is possible to systematically derive more detailed implementations from an early abstract specification in a stepwise fashion. Each cycle aims at establishing a refinement ordering during the development process. Since the refinement ordering is transitive, the final, detailed implementation still satisfies the very first abstract specification. The process should proceed up to a point where one finds a satisfactory design, close to the final implementation code or circuit, while still satisfying the original abstract specification.

The refinement model-checking algorithm is usually a variation of *breadth first search* (BFS) that investigates mutually reachable nodes of a pair of automata, one modelling the specification and another the design or implementation system. A variation frequently used is to provide the specification as a property, rather than an abstract version of the design or implementation [95]. A property to be checked against the specification is defined as an even more abstract program. For instance, for an abstract version of a security protocol, a property specification could be a system which accepts any event except those that incur a security flaw, such as wire tampering. If the abstract specification is proved to be a refinement of this property, that implies the specification is safe with respect to the specific flaw; this is different from an implementation that includes new functionality on the original abstract specification. Thus, property-oriented specification allows checks for specific features, such as deadlock and divergence freedom, as well as variations on the search algorithms for efficiency purposes. For instance, the most successful property-oriented specification checking is deadlock freedom, where a series of clever manipulations on the traditional checking algorithms are available in [62].

The BFS starts at the initial nodes of each automata, representing the root of the search. The search algorithm looks for incompatible node pairs that are reachable via a given trace from each automaton. At each node pair, the properties under investigation are checked. If a node pair is compatible, the search looks for successor nodes from each element of the pair, hence leading to the set of mutually reachable node pairs to be searched next. Otherwise, if an incompatible node pair is found, a witness is generated by building the automaton backwards from the failed pair to the root of the search. Moreover, the algorithm could carry on trying to find witnesses through alternative paths available, if the user so requires.

Breadth first search (BFS) is used because, if a witness is found, then it is always the cheapest possible in terms of necessary time and work effort for performing the search [37, 87]. It also enables efficient parallelisation [63]. More details on BFS algorithms and their variations for refinement model checking can be found in [34, Chap. 4].

Counterexamples provided using refinement model checking are more informative than those generated using classical model checking. Although improvements for classical model checking have been discussed in [101], these improved witnesses from refinement model checking containing more than the trace of a failure often enable more detailed investigation and reasoning about flaws. On the other hand, in order to judge information from witnesses accurately, it is imperative that either the path for its occurrence is deterministic or additional information about nondeterminism is stored. The former allows more compact witnesses but requires one of the two automata in the refinement relation to be deterministic. The latter do not impose restrictions on the automata, but require more memory to perform the search.

The idea of refinement is very attractive. It enables not only property checking, but also consistent evolution from abstract specifications, passing through intermediate design, and up to the final program code in what is known in formal software development as correctness by construction [4, 38, 107].

In what follows, automata theoretic methods used to establish refinement through model checking are discussed. All automata theoretic methods can be viewed as variations of interpretations over some theory of finite automata, as given in [43]. The language and the properties each automaton represents reflects the expressiveness of the formal notation.

CSP The CSP [88] community uses refinement model checking as a verification technique; tools can be found in [32, 36, 56, 97]. For that, a formal model for the specification and the implementation is derived as a pair of automata, where extra information on the nodes and arcs about refusal sets and divergence is added. This additional information is used in order to model more than just language containment by traces.

Language containment is established based on the (denotational) semantic models of CSP represented on the arcs and nodes of the automata. The CSP automata are built using an operational semantics. Following the denotational semantics of CSP, one can establish refinement in three models; that is, automata containment with respect to three different perspectives:

1. The traces model, which deals with safety properties.
2. The stable-failures model, which deals with nondeterminism.
3. The failures–divergences model, which deals with both nondeterminism and divergence.

More details on this can be found in [22, 87].

Sometimes it can be a bit awkward to define relatively simple properties related to predicative formulae in CSP, if compared with the simplicity of temporal logic formulae, as pointed out in [95]. Even so, the witness information returned from a failed check is far more detailed than those available in temporal logic, as it includes not only traces, but also additional information from the models chosen to perform the check, such as refusal sets and divergence information.

A similar approach for refinement is taken by the *Circus* community [110]: an integrated refinement language that combines CSP, Z [102], and the refinement calculus [68].

CCS[Query8] Milner takes a different approach in modelling the CCS process algebra [66]. Instead of collecting a set of sequences of traces like in CSP, processes are compared in CCS by capturing external behaviours using a tree of observations.

The notion of correctness in CCS is established by an *observational equivalence* between a pair of processes. In this view, two processes are equivalent if an observer cannot distinguish between them by any experiment. Another form of equivalence is defined in order to capture internal actions; it is called *behavioural equivalence*. This is important when one needs to consider internal actions and nondeterminism. These forms of equivalence can be proved by establishing a *bisimulation* relation between the two processes.

Bisimulation equivalences were also studied by Cleaveland and Hennessy [22]. In fact, the few available descriptions of the CSP model checker failures–divergences refinement (FDR) [36] often refer to [22] as a main source of information for the construction of its automata theory and search algorithm. That is because it gives an automata theoretical account for the data structures used in FDR. Additional references to the bisimulation approach used for model checking can be found in [28,77,85].

Examples in industry A successful industrial-scale example of refinement model checking is in the area of security protocol design and verification [95]. In fact, the initial motivation for the available refinement model-checking tools was exactly to solve this problem. The experiment was such a success that it progressed to a set of commercial and academic tools of great value. Other related tools are also available, such as a CSP parser [97], Java libraries for CSP [6,35], translators [71], a refinement checker specialised in deadlock detection [62], a concurrent programming language [46], and so on. Currently, there are two main tools: one is the refinement model checker FDR [36], and the other is the CSP animator process behaviour explorer (ProBE) [32]. FDR and ProBE have been used to model-check a wide variety of industrial-scale systems, such as the example in [89]. Another refinement model checker for CSP is ARC[Query9]; it is still a prototype [80]. Benchmark analyses comparing FDR with ARC and other tools, and a detailed description of ARC's algorithm can also be found in [79].

According to [90], on particular kinds of systems FDR can check 10^{1000} distinct states and beyond. With the aid of a clever combination of techniques and properties of the CSP language, such as monotonicity of some operators, this number could in fact grow to the staggering figure of $7^{10^{1000}}$. This is not the actual number of states checked by FDR, but the total number of states of the combined system components, which means that checking a fraction of the state space is as good as checking the whole of it. By dealing with such a huge number of states, FDR is able to model not only hardware, but also quite complex software designs.

In the domain of security protocols, these tools were used for a variety of property checks, such as secrecy [99], authentication [55,98], or non-repudiation [95]. The success of the approach produced yet another tool called: compiler for the analysis of security protocols (CASPER) [56]. This tool accepts a formal notation tailored for the description of security protocols. The tool then compiles the protocol description into CSP code to be used by FDR for checking the desired properties.

Since CASPER is a compiler, the problem mentioned earlier of the complexity of simple property descriptions involving predicates in CSP is completely hidden from the user. That is, the user only needs to interact with a quite simple interface, and many requirements and tricks in dealing with FDR are entirely abstracted via CASPER. This is a good example of a mature interaction between formal tools.

Many protocols have already been model-checked with these tools, and a considerable amount of expertise and knowledge is already available in this field. For example, the CyberCash [57], and the TMN[Query10] [58] protocols were successfully checked with the support of FDR, ProBE, and CASPER. For some necessary domain abstraction, and infinite property checks, the theorem prover PVS [92] was used.

Another example that involves exploration of the monotonicity of CSP operators in a field different from security protocols is given in [69]. This work presents a formal model of the Brazilian satellite SACI-1 in CSP-Z [31]. The author uses FDR to show that there was a flaw in the design of the communication facilities between the satellite and the Earth. Unfortunately, the verification was carried out when the system was already in use, and it was too late: the Brazilian space agency lost communication with the satellite after a few initial interactions. Other successful examples in the application of CSP and FDR, its variations, and future directions can be found in [1].

A prototype model checker for *Circus* has just been completed [34]. It enables refinement model checking of systems that combine behavioural aspects defined in CSP with data aspects defined using Z and guarded commands.

4 Overcoming the state explosion problem

We have already mentioned the state explosion problem for model checking; it happens whenever different components of systems interact, producing a massive number of states to be analysed. Suggestions for handling this problem are given by Clarke [21], Roscoe [87], Goldsmith [90], Shankar [101], Valmari [104], and others [48]. These are not new methods, but clever manipulations of the model-checking data structures and algorithms in order to reduce the state space, and improve the efficiency of the whole model checking task. In the sequel, we discuss some of these possibilities; they can be applied to both classical and refinement model checking.

Symmetry reduction minimises the number of possible states by the identification of replicated behaviour in the composition of nontrivial components. It allows the simplification of both the temporal logic formula, and the specification. Nevertheless, its application often requires replication of the system components, a fairly observable aspect in hardware, as well as concurrent and reactive software systems. More details about this technique for classical model checking can be found in [64, Chap. 14].

Induction is a technique applicable to families of finite-state systems. The method finds an invariant process describing the behaviour of the entire family of processes, which allows one to check the whole family at once. The main disadvantage lies in the fact that finding families of processes is normally an undecidable procedure, and is hence harder to automate. An interesting discussion of the use of symmetry and induction techniques for classical model checking is given in [19]. More details about induction can also be found in [64, Chap. 7], and [21, Chap. 15].

Modular structure exploration allows the refinement model checking of larger systems by exploring the properties of the operators of the language under consideration. For instance, the CSP parallel operator is monotonic with respect to refinement for deadlock freedom. Therefore, if we prove that a process P_1 is refined by another process P_1', and similarly for P_2 and P_2', then we can conclude that the parallel composition of P_1 and P_2 is refined by the parallel composition of P_1' and P_2'. For instance, take the example of the Brazilian satellite SACI-1; it uses the monotonicity of a restricted CSP parallel operator with respect to deadlock freedom in order to reduce the number of states from 201, 168 to 3426, and the number of transitions from 1, 705, 581 to 13, 680 on the checked automata structures. This is a very important result because it allows the direct checking of systems with a huge number of states and transitions, which would otherwise be very costly, if possible at all.

Assumption containment is another technique that is similar to modular structure exploration. The main difference is the presence of assumptions when mutual dependencies between components of the specification arise. In this approach, when verifying a property of a component, one makes assumptions about the behaviour of other components. These assumptions are expected to be discharged at some convenient point during the development process. More details on this technique can be found in [82,67].

Abstract interpretation is a very important industry-oriented technique for tackling the state explosion problem by minimising the state space, which is used by both the classical and the refinement model-checking approaches. It simplifies the state space and yet guarantees that the original properties are preserved. The major advantage of the technique is to enable the possibility of model-checking infinite systems, as well as making the verification process simpler by means of domain reduction. Nevertheless, it is normally necessary for the user to provide an accurate abstraction.

Finding such abstraction is a difficult task that often requires theorem proving in order not to compromise the desired properties being checked by an oversimplification of the original system. It is necessary to find a balance between the benefits of abstraction, and the major effort that it takes to be proved consistent. An algorithm to find an abstraction for a combination of Z with CSP is given in [70]. A similar approach is also available for data-independent CSP processes [51]. In [101], the same idea is used for CTL in classical model checking. More details regarding abstract interpretation techniques for model checking and theorem proving can be found in [30,23].

Finally, under particular circumstances, it is possible to make *compressions* over the representation of models without semantic loss [90]. By compression we mean manipulation at the level of the automata representing the systems being analysed, such as subset construction [43, Chap. 2], bisimulation [22], and other methods [48,103]. In fact, it is recommended that the user try to describe models by having such possibilities in mind. An interesting discussion on this topic related to algorithm complexity is given in [104].

5 Iterated integrated analysis

The answer to overcoming the state explosion problem for more complex software systems lies in combining effective and expressive verification techniques, with the highest degree of automation and the lowest compromise in soundness possible. The user expects something useful, powerful, effective, and expressive enough with the smallest amount of effort possible. The technique that makes the best balance of these issues is model checking. Nevertheless, under some circumstances where expressive abstractions, or reasonably sized system specifications are not easily available, theorem proving is the choice that remains for achieving correctness.

In this section we present how techniques are being combined to overcome the state explosion problem. These combinations have increased the interest in formal verification from both academia and industry because of their greater expressive power, and yet considerably high level of automation. These are focused mainly on the combination of model checking and theorem proving. The former due to its high level of automation and the ability not only to present counterexamples for failures, but also to reason mechanically about those failures; the latter due to its expressiveness, and ability to handle infinite or unbounded models (see Fig. 2, which is similar to an approach in [93]). When blended, those techniques push the edge of mechanisation towards a more expressive and less interactive direction, which is very attractive to users.

Thus, the combination of model checking and theorem proving have become the state of the art in terms of practical and theoretical development for formal verification techniques. For instance, the Spark Ada toolset combines automatic static checking with automated theorem proving for a subset of Ada tailored for high-integrity and safety-critical systems [9]. In [34], one can find recent developments on a symbolic refinement model checker for *Circus*, a state-rich refinement language [110].

Whenever possible, model checking is preferable due to its higher level of automation. Nevertheless, due to the state explosion problem mentioned earlier, theorem proving is needed in order to perform formal verification over infinite or unbounded systems. In this combined approach, theory and tools must provide a balance between interaction effort and reward. There are two main issues: (i) the decidability of

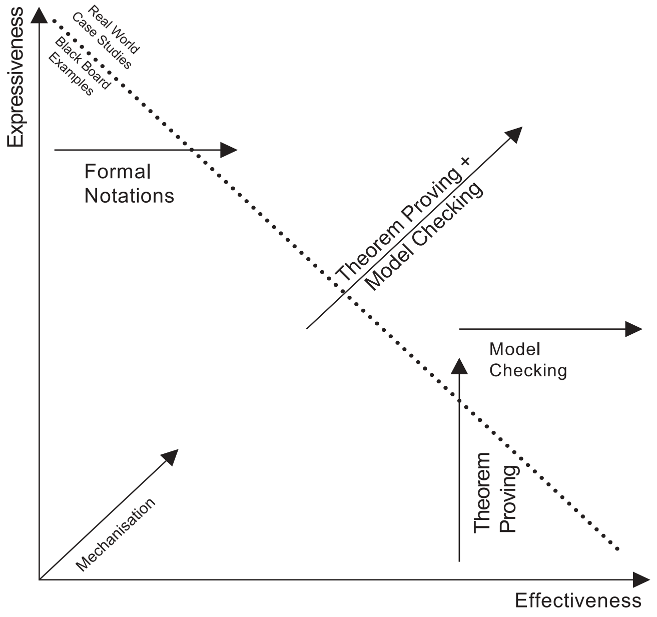

Fig. 2 Combining model checking with theorem proving

the theories being used, and (ii) the ability to use automatic proof tactics and efficient decision procedures. When interaction is unavoidable, it ought to be as close as possible to the user's domain of expertise.

5.1 Finding abstractions

Figure 3, which is based on [100,21], depicts an iterative cycle to combine model-checking and theorem-proving techniques in order to: (i) find abstractions to enable model checking; and (ii) interpret debugging information from model checking mechanically. Basically, we have an interactive stepwise method in which theorem proving is used to prove the correctness of an abstraction that enables model checking. The user takes advantage of three effective formal methods: model checking, theorem proving, and abstract interpretation.

Firstly, it is usually the responsibility of the user to propose a suitable abstraction. This makes the technique a bit expensive, as a considerable amount of theoretical and technical skills in different subjects might be needed.

After that, in order to maintain correctness, theorem proving is used to guarantee that the chosen abstraction fulfills the original requirements. Unfortunately, finding such approximations and proving their correctness using a theorem prover can be hard. Under particular circumstances, however, these tasks of finding an abstraction and proving its correctness are possible to automate.

Once the simplified domain is available, one is able to use model checking to validate the desired property claims using the approximated model. The debugging information of model checking can then be further analysed by the theorem prover in order to draw the final conclusions, possibly automatically [91].

Finally, the idea of the iterated approach is to repeat this cycle by strengthening (or weakening) the invariant of the abstraction until, eventually, a property-preserving abstraction is found that allows one to verify aspects of the concrete design as required. Nevertheless, this use of under- and overapproximation of abstractions can generate "dubious witnesses or spurious counterexamples" [101]. This technique has been encoded as decision procedures in PVS.

Another approach uses simplified debugging information and an expressive translation from ANSI-C (including pointers, bit vectors, unions, etc.) to propositional logic, where the compromise made for higher automation is soundness; that is, false positives (or dubious witnesses) can be generated. This approach is used by Microsoft's SLAM toolset for formal verification of Windows device drivers [24,50]. It allows both a wide variety of systems to be analysed, as well as a higher degree of automation to be achieved, with as little user interaction as possible. On the other hand, the approach is limited in the scope of properties it can check.

As pointed out in [91], enumerating the behaviour of the abstracted version of the system is a better debugging method than exploring some of the behaviours of the original one, as it simplifies the irrelevant details for the property being checked. When one proves a given abstraction to be faithful with respect to the original domain, any property claim on the abstracted system should be valid on the original one as well.

Proposing abstractions For example, in [100], an abstraction for basic integer arithmetic is proposed: the integers ($\mathbb{Z}$) are represented by the smaller set $\{0, +1, -1, \top\}$, and the following interpretation is chosen:

$$[\![0]\!] = \{0\}, \quad [\![+1]\!] = [0, \infty),$$
$$[\![-1]\!] = (-\infty, 0], \quad [\![\top]\!] = \mathbb{Z}$$

With these new abstracted definitions, operations like "$+$" and "$-$" can then be lifted to "$\hat{+}$" and "$\hat{-}$", as shown in Table 2. In this way, we have a finite representation for the original infinite-state space. The next step is to prove that, with such simplifications, the new design preserves the original properties (soundness). This is done by establishing a *Galois connection* between the concrete and the abstract domains [25]. This is the task that normally requires a theorem prover.

Once the new design is proved correct, one can feed it to the model checker in order to perform the desired property checks. The returned results can then be used either as a final answer to the user, or as input for the definition (and proof) of a new abstraction, if the one provided is not yet sufficient because it might still be inadequate for model checking. For instance, this might happen if the abstracted domain is still too large, and hence further reduction is needed. This approach is quite similar to Z data refinement [108, Chap. 16], another well-known technique used in formal verification.

This sort of guessing is prone to be unsound at first choice in a complex scenario, and that is why the use of theorem

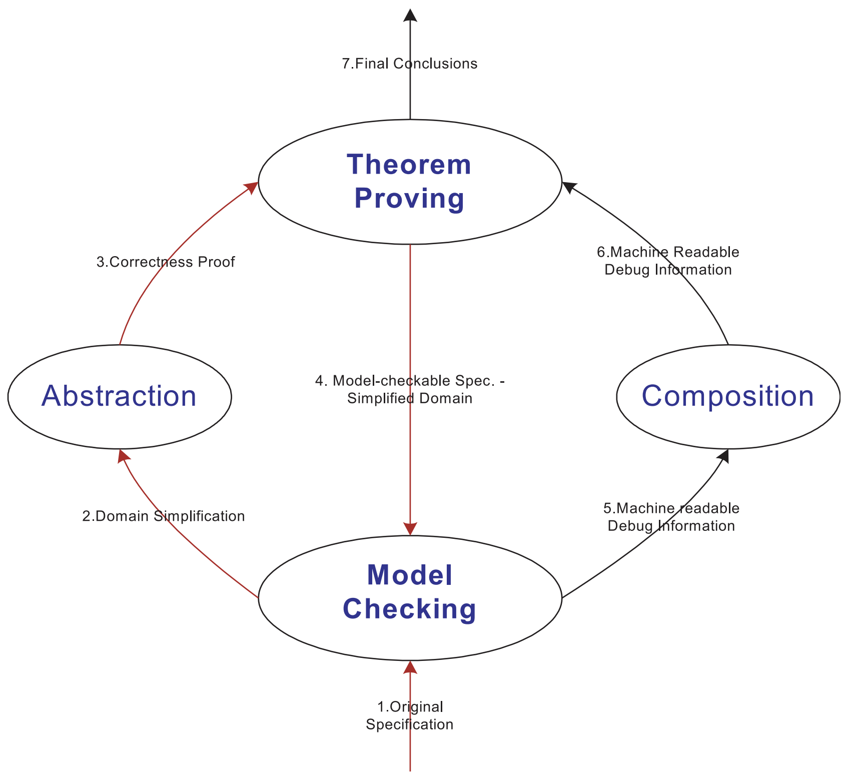

Fig. 3 Iterated integrated analysis

Table 2 Integer arithmetic operations abstraction example

$\hat{+}$	**0**	**++1**	**−1**	$\top$
0	0	+1	−1	$\top$
+1	+1	+1	$\top$	$\top$
−1	−1	$\top$	−1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$
$\hat{-}$	**0**	**+1**	**−1**	$\top$
0	0	−1	+1	$\top$
+1	+1	$\top$	+1	$\top$
−1	−1	−1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

proving for correctness is needed. In any case, even if the abstraction is not yet correct, bugs found in the approximated model are likely to indicate problems in the original system. The current trend is to compromise soundness for higher levels of automation whenever possible [14,101], or lower (yet acceptable) automation and expressiveness for high assurance and correctness by construction [4,107]. The choice depends on the problem domain. For instance, formal verification is being used to prove the SmartCard APIs [Query11] in JML [83], where high levels of automation are of greater priority. On the other hand, for high-integrity or safety-critical systems, such as flight control or banking systems [38], one cannot afford this form of compromise and other techniques are used.

Systems with more-complex state spaces have created the demand for new techniques. The most successful in this direction is the use of symbols while encoding the formal model to perform a check. Being applicable to both classical and refinement-based model checking, the symbolic approach enables the encoding of certain types of infinite (or unbounded) systems. This is possible through the representation by symbols of their characteristic features in some suitable logic, rather than by enumeration of all possible states. That is, representing distinct characteristics (classes or properties) of states, rather than each individual state.

For instance, a counter based on the infinite set of natural numbers ($\mathbb{N}$) can be represented by the proposition $x \geq 0$ or the set $\{\ v : \mathbb{Z} | v \geq 0\ \}$ with one single state, rather than one state for each individual positive number. That is, what matters is the fact that we do not consider negative numbers.

Classical model-checking example Classical model checking explores the idea of finding abstractions automatically. Although the user is forced to give the initial approximated abstraction, it can be either very coarse or very strict. Tools are used to make sure that the approximated abstraction is neither too weak nor too strong. A weak (or under-) approximation needs to be strengthened in order to avoid missing possible flaws; it is related to completeness. On the other hand, a strong (or over-) approximation needs to be weakened in order to avoid catching problems not present in the concrete domain originally; it is related to soundness [101].

The approach taken is to use model checking itself together with predicate abstraction [8], in order to find an effective approximation. The difference is that it uses symbolic characterisations of the problem domain, such as the example given earlier of the natural number counter, rather than explicit state enumeration. A nice side-effect of the use of symbols is that the returned witnesses now carry more information than just the usual traces. This information is the symbols characterising the approximations taken during the abstraction process.

This approach is available, for instance, in the SAL [Query12] framework embedded into the PVS theorem prover [10,74]. Another example of the use of model checking for finding suitable abstractions is the work in [65]. The user does not need to provide an initial abstraction. It is calculated directly from the given concrete domain [24]. It starts from an empty abstraction (or under approximation) that is completed as the check goes along.

Refinement model-checking example As already mentioned, an algorithm to find domain abstractions for data-independent CSP-Z processes is given in [70], with further extensions and optimisations provided in [30]. In this approach, the abstraction does not need to be given by the user. Instead, the algorithm starts from an empty domain and the search looks for necessary strengthening predicates; this is similar to [24]. The search terminates when both automata from the refinement relation are exhaustively searched.

As the automaton contains a specific encoding of the Z part of the CSP-Z specification using a functional language similar to Haskell, some verification conditions pop up to the user to prove as theorems. This means that the user needs to guide the algorithm by choosing which parts of the search to prune according to his answers to the presented verification conditions.

Other related efforts towards finding and proving domain abstractions automatically for CSP are given in [105].

5.2 Symbolic refinement model checking

A completely different instantiation of the iterative combination of techniques for formal verification has been developed recently [34]. It combines refinement model checking and theorem proving in a way that allows not only property checks and stepwise development, but also automated calculation of abstractions via data refinement.

The symbolic refinement model-checker prototype works for a subset of *Circus*, whose semantic background is based on Hoare and He's unifying theories of programming (UTP) [41].

The objective of the design of *Circus* was to enable the specification of both data and behavioural aspects of concurrent and reactive systems, as well as to support correct-by-construction stepwise development through refinement. It provides a method of program development that is based on refinement laws, and is calculational in style [75].

The motivation for a new tool arises from existing research in the specification of systems using *Circus* [5,109]. Due to the lack of adequate tools, adaptations and simplifications have been carried out in order to make the descriptions suitable for analysis using FDR. This motivated the work on theory and tools for model checking *Circus*, such as the model checker and the basis of a theorem prover for the UTP [76].

The *Circus* model checker can be used similarly to the way in which FDR is used for CSP. Nevertheless, due to the possible complexity introduced through the use of Z and guarded commands, some user interaction might be required in the form of interactive theorem proving. The main goal is to have a sound tool that exhaustively checks for refinement with the highest level of automation possible, and which gives accessible (human-readable) counterexamples in the case of failure. In practice, the way one is expected to use the tool is depicted by the cyclic process presented in Fig. 4.

Firstly, a *Circus* specification in LaTeX is given to the compiler, which implements an operational semantics that generates finite automata representing *Circus* specifications on-the-fly, as the checking progresses through the witness search algorithm. Finiteness is achieved with the use of symbolic representation of characteristic sets of states.

On-the-fly model checking is a well-known technique [81]. The direct benefit is that, since model checking often finds problems early in the design, there is no need to calculate the whole automaton up front. Due to the state explosion problem that often arises, keeping track of the entire automata is not efficient, and hence avoided.

In [33], a suitable theory of automata that encodes predicate logic through set comprehension on automata arcs is presented. It is called the *predicate transition system* (PTS), and is the output of the *Circus* compiler. A thorough account of possible variations, as well as what one can gain from theoretical exploration of algorithm complexity and automata theory for the benefit of the symbolic model checking approach, can be found in [104].

The refinement search is carried out by the refinement engine. It receives the partially compiled transition system and performs the main activities involved in witness search: (i) the validation of the refinement search criteria for a current node pair; (ii) the identification of new successor nodes to check; and (iii) the generation of counterexamples, if necessary.

Refinement criteria validation and search for successor nodes demands evaluation of predicates and expressions, which in turn might require automated theorem proving. While searching for new nodes, the refinement algorithm turns back to the compiler for further compilation, now of different programs corresponding to the available nodes after the refinement criteria check. If a counterexample is found during the search, the debugger comes into play to properly cast the refinement search information into a human-readable format, which is presented as textual output.

The predicates and expressions generated are verification conditions for another component to evaluate. The user does not need to provide any suitable (and correct) formulae or abstraction, as soundness is guaranteed by construction, but the discharge of proof obligations may be needed. Obviously, the more complex the system description, the harder the proof.

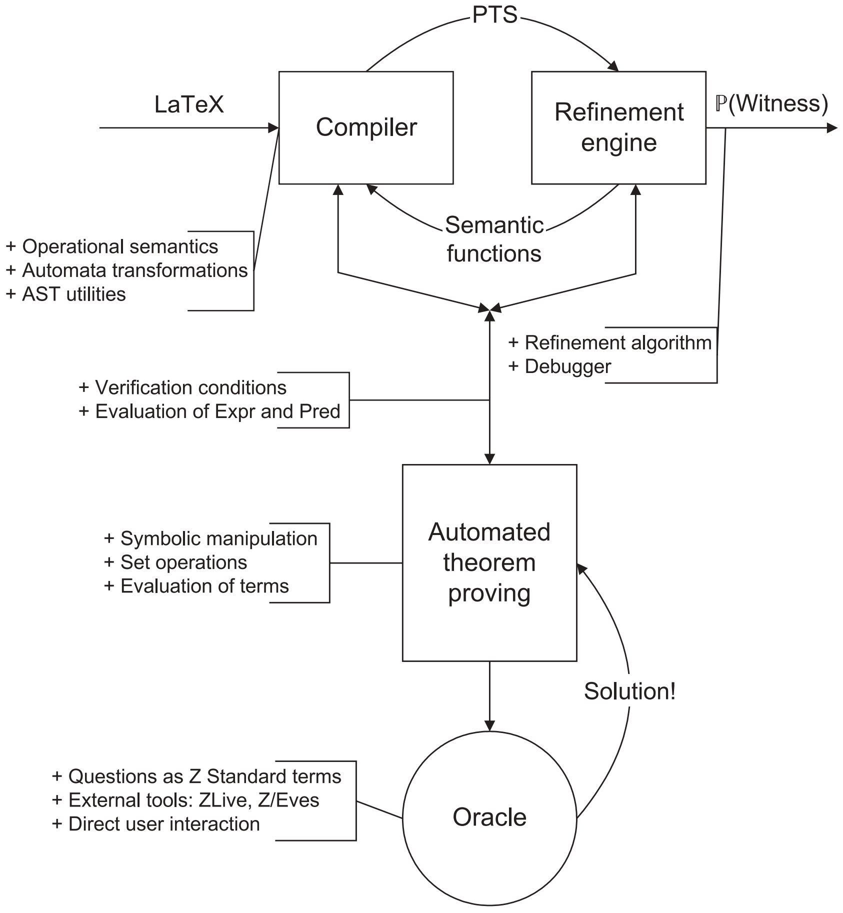

Fig. 4 Using the *Circus* model checker

An interesting aspect of this method still being explored is that the verification conditions can be used to suggest an abstraction to the user. This is a sort of semi-automatic data refinement or abstract interpretation.

Finally, the oracle component is responsible for discharging proofs that the previous stages could not handle automatically. They are passed either to the user, or to general-purpose theorem provers, which can be plugged into the prototype. In the current implementation, a Z animator (ZLive) [59], the *Z-Eves* theorem prover [96], and a simple user interface for direct questioning are integrated. As the prototype is in conformance with the Z standard [111], any theorem prover that supports Z, such as ProofPowerZ, can also be used [52].

This way of integrating iterative analysis, with the use of different tools in a pipeline, handling different levels of complexity, from automatic term rewriting to interactive theorem proving or indeed user intervention, has been used in other tools, such as ESC[Query13]/Java2 [27] for JML, and correctness by construction with the Spark Ada toolset [9].

A subtle point for future research is the analysis of possible dependencies between generated verification conditions and the model-checking search algorithm, in order to enable the use of parallel algorithms. Furthermore, if stand-alone parallelisation of model-checking algorithms is not enough, there is the possibility to try using grid technology in order to enhance the development of parallel algorithms through distribution of search data via a grid network [78]; this is a topic that is yet to be explored.

To the extent of our knowledge, there is only one other technique that aims at symbolic refinement model checking and combination with other technologies, such as theorem proving. It is based on a tool called ProB, which combines the different paradigms of behaviour and data from CSP with the B method [2]. Although ProB [15] has already combined CSP and B, support for refinement checking is still limited, as shown in its manual (version 1.1.4) [54].

6 Summary and discussion

Model checking is well accepted in industry, mainly because of its degree of expressiveness with respect to high-value bugs found, and the lower amount of effort required when compared with other techniques. The evaluation of all possible behaviours, even for approximate models, often reveals early flaws; that is, bugs that would have cost more if detected later (or gone otherwise unnoticed) during the development process. This turns out to be a better approach than performing ad hoc testing or simulation on the final product, since these do not guarantee correctness.

Provided the specification of the system and its properties meet the informal requirements to the user's satisfaction, and provided the verification technology used is sound, if a bug is found, it is certain that the system is flawed. Moreover, provided the technology used is also complete, if no bug is found, the system is correct (and hence flawless) with respect to the properties being checked.

Another important factor for the acceptance of model checking is its ability not only to find problems, but also to enable reasoning about them. In model checking, bugs are reported with tractable counterexamples that can be used (often automatically) to draw a test case or simulation scenario. In other words, the returned counterexample can be used by other formal analysis tools for further reasoning.

There are two mainstream approaches to model checking. The first, classical model checking, uses more natural or succinct property specifications, whereas the second refinement-based approach, supports not only property checking, but also correctness by construction via stepwise development. Nevertheless, specification of properties involving predicates in the refinement model-checking approach for CSP is not as straightforward as in classical model checking. For a refinement-based model checker that combines CSP with Z, however, this problem no longer exists, as the user can easily (and elegantly) describe predicates using Z [34].

In classical model checking, the use of OBDDs increases the number of states that can be handled, and it is known in the literature as symbolic (classical) model checking. Nevertheless, OBDDs are not always suitable because classical model checking cannot deal with the huge number of states that normally occur in reactive software systems.

Symbolic (classical) model checking can be considered as a new algorithm for theorem proving—a proof tactic [100]. Properties of OBDDs often determine the kind of system structures that can be efficiently analysed using this approach. For instance, OBDDs are suitable for hardware designs, which are usually less dynamic, rather than complex (state-rich) software systems. The use of symbols has also been explored for finding optimal abstractions of the original concrete domain to cut down the problem to a model-checkable size [101].

In refinement model checking, the use of clever automata manipulation, compression techniques, and modular structure exploration enables an enormous number of states to be checked. Furthermore, whenever state-rich designs are modelled, the compromise is that theorem proving may be needed.

In both approaches, the main properties under consideration are often related to control, instead of data. This is due to the necessity to downscale the number of states to a manageable number. Reduction can be achieved through further formal techniques, such as modular structure exploration, symmetry, partial order reduction, compression, or abstract interpretation. The degree of automation with respect to the state explosion problem and tool support are usually the main issues for an appropriate choice of technique. For state-rich designs, such as those described in *Circus*, it seems to be impossible to use OBDDs or classical (symbolic) model checking, since it is not possible to simplify the complexity of the state representation and operations to simple Boolean formulae as required [21,64].

In [53], it is mentioned that LTL formulae can be translated to FDR, which enables the integrations of the different model-checking approaches. This topic of interchangeability between approaches is also mentioned by Clarke [21].

In the context of combined techniques the direction is clear: to provide theory and tools aimed at shedding light on and providing guidance in bridging this gap between current demands, and actual tool support, with the highest levels of automation possible. There is clear demand not only for integrated methodologies, such as model checking, theorem proving, and abstract interpretation, but also integrated programming paradigms. In this direction, we have discussed the importance of addressing the challenge of integrating refinement model checking with theorem-proving support, as well as support for different language paradigms, such as data and behaviour, through the *Circus* and ProB model checkers.

The main idea is to take advantage of the various (formal) verification techniques available, by extracting the best out of the most successful ones, that is, model checking, theorem proving, and abstract interpretation: high automation levels with as few compromises on expressiveness and soundness as possible.

References

1. Abdallah AE, Jones CB, Sanders JW (eds) (2004) Communicating sequential process: the first 25 years, no. 3525, in Lecture Notes in Computer Science, symposion on the occasion of 25 years of CSP, Springer, London UK, July 2004
2. Abrial J-R (1996) The B book—assigning programs to meanings. Cambridge University Press, Cambridge
3. Alur R (2002) Mocha: Modularity in model checking [Query14]
4. Amey P (2005) Correctness by construction: better can also be cheaper. Crosstalk J Def Softw Eng Dec: 5–8
5. Atiya D, King S, Woodcock JCP (2003) Ravenscar protected objects: a Circus semantics. Technical Report 356, Department of Computer Science, University of York, York
6. Austin PD, Welch PH (2000) Java communicating sequential process—JCSP. PowerPoint slides, August 2000 [Query15]
7. Back R-J, von Wright J (1998) Refinement calculus: A systematic introduction. Graduate text in computer science. Springer, Berlin Heidelberg New York
8. Ball T, Cook B, Das S, Rajamani SK (2004) Refining approximations in software predicate abstraction. In: Proceedings of 10th international conference on tools and algorithms for the construction and analysis of systems – TACAS'04, pp. 388–403
9. Barnes J (2003) High integrity software: the spark approach to safety and security, 2nd edn. Addison–Wesley, Reading
10. Bensalem S, Ganesh V, Lakhnech Y, Munoz C, Owre S, Rueß H, Rushby J, Rusu V, Saïdi H, Shankar N, Singerman E, Tiwari A (2000) An overview of SAL. In: Holloway CM (ed.) LFM 2000: 5th NASA Langley formal methods workshop. NASA Langley Research Center, Hampton, VA, pp. 187–196
11. Biere A, Cimatti A, Clarke EM, Fujita M, Zhu Y (1999) Symbolic model checking using SAT procedures instead of BDDs. In: DAC '99: Proceedings of the 36th ACM/IEEE conference on design automation, ACM, New York, pp. 317–320

12. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput 35(8):677–691
13. Burch I (1994) Symbolic Model Checking for Sequential Circuit Verification. IEEE Trans Comput Aided Des Integr Circ Syst 13:401–424
14. Burdy L, Cheon Y, Cok DR, Ernst MD, Kiniry JR, Leavens GT, Rustan K, Leino M, Poll5 E (2003) 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. University of Nijmegen, Elsevier, pp. 73–89
15. Butler M, Leuschel M (2005) Combining CSP and B for specification and property verification. In: Fitzgerald J, Hayes IJ, Tarlecki A (eds.) FM 2005: Formal methods, no. 3582, Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York, pp. 221–236
16. Cavada R, Cimatti A, Olivetti E, Pistore M, Roveri M (2005) NuSMV 2.2 user's manual. Carneige Mellon University, Trento, Italy, nusmv.irst.itc.it
17. Chaki S, Clarke EM, Ouaknine J, Sharygina N, Sinha N (2004) State/event-based software model checking. In: Boiten EA, Derrick J, Smith G (eds.) In: Proceedings of the 4th international conference in integrated formal methods, no. 2999, Lecture Notes in Computer Science, pp. 128–147
18. Clarke EM, Kurskan RP (eds.) (1990) In: Proceedings of 2nd international conference in computer-aided verification. No. 531, Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York
19. Clarke EM, Jha S (1993) Symmetry and induction in model checking. Technical report, Carnegie Mellon University, Pittsburgh
20. Clarke EM, Wing JM (1996) Formal methods—state of the art and future directions. ACM Comput Surv 28(4):626–643
21. Clarke EM, Grumberg O, Peled D (2000) Model checking. MIT Press, Cambridge
22. Cleaveland R, Hennessy M (1993) Testing equivalence as a bisimulation equivalence. Formal Aspects Comput J 5(1):1–20
23. Cleaveland R, Iyer P, Yankelevich D (1993) Optimality in abstractions of model checking. Technical report, North Carolina State University, US and University of Buenos Aires, Argentina
24. Cook B, Podelski A, Rybalchenko A (2005) Abstraction refinement for termination. In: Proceedings of international static analysis symposium – SAS'05, London
25. Cousot P, Cousot R (1992) Abstract interpretation framworks. J Logic Comput 2(4):511–547
26. Deharbe D, Shankar S, Clarke EM Jr (1998) Model checking VHDL with CV. In: Formal methods in circuit automation design (FMCAD'98), Lecture Notes in Computer Science, vol. 1522. Springer, Berlin Heidelberg New York, pp. 508–513
27. Detlefs D, Rustan K, Leino M, Nelson G, Saxe JB (1998) Extended static checking. Technical Report 159, COMPAQ Systems Research Center (SRC), www.research.digital.com/SRC/
28. Dovier A, Piazza C, Policriti A (2000) A fast bisimulation algorithm. Technical report, University di Verona and University Udine, November 2000, UDM/14/00/RR
29. Elseaidy W (ed.) (1994) Modeling and verifying active structural control systems. Sci Comput Program 29(1–2):99–122
30. Farias AC (2003) Efficient and mechanised analysis of infinite CSP-Z processes. Master's thesis, Universidade Federal de Pernambuco, Pernambuco
31. Fischer C (2000) Combination and implementation of process and data: From CSP-OZ to Java. PhD thesis, University of Oldenburg, Oldenburg
32. Formal Systems (Europe) Ltd. (2000) ProBE user's manual version 1.28
33. Freitas L (2004) Predicate transition system—automata theory. Appendix A.3 in [34] (CD-ROM)
34. Freitas L (2005) Model checking Circus. PhD thesis, Univeristy of York, York
35. Freitas L, Cavalcanti A, Sampaio A (2002) JACK—a framework for process algebra implementation in Java. In: Proceedings of XVIII Simposio Brasileiro de Engenharia de Software in Gramado, October 2002, pp. 98–113
36. Goldsmith M (2000) FDR2 user's manual version 2.67. Formal Systems (Europe) Ltd, Oxford
37. Goldsmith M (2001) Overview of FDR in [95], chap. 4. Addison–Wesley, Reading, pp. 125–140
38. Hall A, Chapman R (2002) Correctness by construction: developing a commercial secure system. IEEE Softw J 19(1):18–25
39. Har'el Z, Kurshan RP (1990) Software for analytical development of communications protocols. AT&T Tech J 69(1):45–59
40. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–583
41. Hoare CAR, Jifeng H (1998) Unifying Theories of Programming. International series in computer science. Prentice-Hall, Englewood Cliffs
42. Holzmann GJ (1997) The Model-Checker SPIN. IEEE Trans Softw Eng 23(5):1–17
43. Hopcroft J, Motwani R, Ullman JD (2001) Introduction to automata theory, languages, and computation, 2nd edn. Addison–Wesley, Reading
44. The ICS Group (2005) ICS Manual (Version 2.0). SRI International, Computer Science Laboratory, SRI International 333 Ravenswood Avenue, Menlo Park, CA 94025, USA
45. Jackson D, Schechter I, Shlyakhter I (2000) Alcoa: the alloy constraint analyzer. In: Proceedings of the 22nd international conference on software engineering, June 2000, pp. 730–733
46. Jones G, Goldsmith M (1998) Programming in occam 2. International series in computer science, 2nd edn. Prentice-Hall, Englewood Cliffs
47. Kang H-J, Park I-C (2003) SAT-based unbounded symbolic model checking. In: Proceedings of the 40th design automation conference (DAC'03), IEEE, pp. 840–843
48. Kokkarinen I (1998) A veridication-oriented theory of data in labelled transition systems. PhD thesis, Tampere University, Finland
49. Kozen D (1998) Results on the propositional μ-calculus. Theor Comput Sci 27:333–354
50. Lahriri SK, Ball T, Cook B (2005) Predicate abstraction via symbolic decision procedures. Technical Report MSR-TR-2005-53, Microsoft Research
51. Lazić RS (1999) A semantic study of data independence with applications to model checking. PhD thesis, Programming Research Group, Oxford University, Oxford
52. Lemma-One (2003) ProofPower Tutorial
53. Leuschel MA, Massart T, Currie A (2001) How to make FDR spin: LTL model checking of CSP by refinement. In: Oliveira JN, Zave P (eds.) Formal methods Europe 2001, vol. 2021. Springer, Berlin Heidelberg New York, pp. 99–118
54. Leuschel LA, Butler M, Lo Presti S (2005) ProB User Manual version 1.1.4. Declarative systems and software engineering, University of Southampton, and Softwaretechnik und Programmiersprachen, University of Düusseldorf, Germany
55. Lowe G (1996) A hierarchy of authentication specifications. Technical report, University of Leicester, Leicester
56. Lowe G (1997) CASPER user manual. Oxford University, Oxford
57. Lowe G (2002) Simplifying transformations—the CyberCash security protocol in [95], chap. 8. Addison Wesley, Reading, pp. 201–220
58. Lowe G, Roscoe B (1997) Using CSP to detect errors in the TMN protocol. Technical report, Oxford University, Oxford
59. Malik P, Utting M (2005) CZT: A framework for Z tools. In: Treharne H, King S, Henson M, Schneider S (eds.) ZB 2005: Formal specification and development in Z and B: 4th international conference of B and Z users, Guildford, UK, Springer, Berlin Heidelberg New York, pp. 13–15
60. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems—specification, vol. 1. Springer, Berlin Heidelberg New York
61. Manna Z, Pnueli A (1995) The temporal logic of reactive and concurrent systems—safety, vol. 2. Springer, Berlin Heidelberg New York

62. Martin JMR (1996) The design and construction of deadlock-free concurrent systems. PhD thesis, University of Buckingham, Buckingham
63. Martin JMR, Huddart Y (2000) Parallel algorithms for deadlock and livelock analysis of concurrent systems. Communicating Process Architectures [Query16]
64. McMillan KL (1993) Symbolic model checking. Kluwer, Dordrecht
65. Microsoft Research (2004) SLAM: A static driver verifier. research.microsoft.com/slam/
66. Milner R (1990) Communication and concurrency. International series in Computer lence. Prentice-Hall, Englewood Cliffs
67. Misra J, Chandy KM (1990) Proofs of networks of processes. IEEE Trans Softw Eng SE 7(4):417–426
68. Morgan C (1994) Programming from specifications. Prentice-Hall, Englewood Cliffs
69. Mota A (1997) Formalization and analysis of the SACI-1 micro satellite in CSP-Z. Master's thesis, Universidade Federal de Pernambuco, Pernambuco (in Portuguese)
70. Mota A (2001) Model cecking CSP-Z: Techniques to overcome state explosion. PhD thesis, Universidade Federal de Pernambuco, Pernambuco
71. Mota A, Sampaio A (2001) Model checking CSP-Z. Science of computer programming, vol. 4. Elsevier, Amsterdam
72. de Moura L, Rueß H, Sorea M (2002) Lazy theorem proving for bounded model checking over infinite domains. In: Proceedings of the 18th conference on automated deduction (CADE), Lecture Notes in Computer Science, Copenhagen, Denmark, 27–30 July, Springer, Berlin Heidelberg New York
73. de Moura L, Rueß H, Sorea M (2003) Bounded model checking and induction: From refutation to verification. In: Voronkov A (ed.) Computer-aided verification, CAV 2003, Lecture Notes in Computer Science, vol. 2725. Springer, Berlin Heidelberg New York pp. 14–26
74. de Moura L, Owre S, Rueß H, Rushby J, Shankar N, Sorea M, Tiwari A (2004) SAL 2. In: Proceedings of the 16th international conference on computer aided verification (CAV), Lecture Notes in Computer Science, Boston, July 2004, Springer, Berlin Heidelberg New York
75. Oliveira M (2006) Formal derivation of state-rich reactive programs using Circus. PhD thesis, University of York, York
76. Oliveira M, Cavalcanti A, Woodcock J (2005) Unifying theories in ProofPowerZ Draft, Univeristy of York, York
77. Paige R, Tarjan R (1987) Three partition refinement algorithms. SIAM J Comput 16(6):973–989
78. Paranhos D, Cirne W, Brasileiro F (2003) Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In: Proceedings of the Euro-Par 2003: International conference on parallel and distributed computing, August 2003, pp. 169–180
79. Parashkevov AN, Yantchev J (1996) ARC—A tool for efficient refinement and equivalence checking for CSP. In: IEEE 2nd international conference on algorithms and architectures for parallel processing ICA3PP, pp. 68–75
80. Parashkevov AN, Yantchev J (1996) ARC—A verification tool for concurrent systems. In: Proceedings of the 3rd Australasian parallel and real-time conference. Brisbane, Australia
81. Peled D (1994) Combining partial order reductions with on-the-fly model checking. In: CAV '94: Proceedings of the 6th international conference on computer aided verification. London, UK, Springer, Berlin Heidelberg New York
82. Pnueli A (1984) In transition for global to modular temporal reasoning about programs. In: Apt KR (ed.) Logics and models of concurrent systems, NATO ASI. Springer, Berlin Heidelberg New York
83. Poll E, van den Berg J, Jacobs B (2000) Specification of the JavaCard API in JML, chap. 3. pp 135–154. Kluwer, Dordrecht. Also Department of Computer Science, University of Nijmegen. CSI report CSI-R0005
84. Pong F, Dubois M (1997) Verification techniques for cache coherence protocols. ACM Comput Surv 29(1) 82–126
85. Rajasekaran S, Lee I (1998) Parallel algorithms for relational coarsest partition problems. In: Proceedings of the IEEE transactions on parallel and distributed systems, vol 9(7). IEEE CS, pp. 687–699[Query17]
86. Roscoe AW (ed.) (1994) A classical mind: Essays in honour of C. A. R. Hoare. International series in computer science. Prentice-Hall, Englewood Cliffs
87. Roscoe AW (1994) Model checking CSP in [86], chap. 21. Prentice-Hall, Englewood Cliffs, pp. 353–378
88. Roscoe AW (1997) The theory and practice of concurrency. International series in computer science. Prentice-Hall, Englewood Cliffs
89. Roscoe AW, MacCarthy H (1994) Verifying a replicated database: A case study in model checking CSP. Technical report, Oxford University, Oxford
90. Roscoe AW, Gardiner PHB, Goldsmith MH, Hulance JR, Jackson DM, Scattergood JB (1995) Hierarchical compression for model checking CSP or how to check 10^{20} dining philosophers for deadlock. First TACAS in Lecture Notes in Computer Science, vol. 1019(1)
91. Rushby J (1995) Model checking and other ways of automating formal methods. Model checking for concurrent programs software, quality week—San Francisco, Position Paper—-SRI International
92. Rushby J (1997) Specification, proof checking, and model checking for protocols and distributed systems with PVS. Formal description techniques and protocol specification, testing and verification (FORTE/PSTV)—Osaka, Japan; SRI international—paper and tutorial slides, pp. 9–12
93. Rushby J (1999) Mechanised formal methods: Where next? In: The World congress on formal methods—Toulouse France, no. 1708, Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York. pp. 48–51, invited paper; SRI international—paper and tutorial slides
94. Rushby J (2000) From refutation to verification. Formal description techniques and protocol specification, testing and verification (FORTE XIII/PSTV XX)—Pisa, Italy, pp. 369–374
95. Ryan P, Schneider S, Roscoe B, Goldsmith M, Lowe G (2001) Modelling and analysis of security protocols. Addison-Wesley, Reading
96. Saaltink M (1992) Z/Eves 2.0 user's guide. ORA Canada TR-99-5493-06a
97. Scattergood JB (1992) A parser for CSP. Technical report, Oxford University, Oxford
98. Schneider S (1997) Verifying authentication protocols with CSP. Technical report, Royal Holloway, University of London, London
99. Schneider S (1998) Security properties and CSP. Technical report, Royal Holloway, University of London, London
100. Shankar N (2002) Mechanised verification methodologies. In: Summer school in specification, verification, and refinement, Turku, Finland
101. Shankar N, Sorea M (2004) Counterexample-driven model checking. CSL technical report SRI-CSL-03-04, SRI International
102. Spivey JM (1998) The Z notation: a reference manual. Prentice-Hall, Englewood Cliffs
103. Valmari A (1990) A stubborn attack on state explosion in [18], chap. 2. No. 531, Lecture Notes in Computer Science. Springer, Berlin Heidelberg New York, pp. 156–165
104. Valmari A (2005) What does theory say about the possibilities of improving efficiency. UK Model Checking Days, University of York, York, www.cs.york.ac.uk/~luettgen/ukmcdays
105. Wehrheim H (2000) Data abstraction techniques in the validation of csp-oz specifications. Formal Aspects Comput J 12(3):147–164
106. Williams PF, Biere A, Clarke EM, Gupta A (2000) Combining decision diagrams and SAT procedures for efficient symbolic model checking. In: CAV '00: Proceedings of the 12th international conference on computer aided verification, London, UK, Springer, Berlin Heidelberg New York, pp. 124–138
107. Woodcock J (2003) UK grand challenge in computer science: dependable systems evolution. www.nesc.ac.uk

108. Woodcock J, Davies J (1996) Using Z: Specification, refinement, and proof. International series in computer science. Prentice-Hall, Englewood Cliffs
109. Woodcock JCP, Cavalcanti ALC (2001) The steam boiler in a unified theory of Z and CSP. In: Proceedings of 8th Asia–Pacific software engineering conference (APSEC01), IEEE Computer Society, pp. 291–298
110. Woodcock J, Cavalcanti A (2002) *Circus* —a concurrent language for refinement. Technical report, University of Kent, Canterbury
111. Z Standard (2000) Formal specification, Z notation, syntax, type and semantics—consensus working draft 2.6. Technical Report JTC1.22.45, BSI panel IST/5/-/19/2 (Z notation) and ISO panel JTC1/SC22/WG19 (Rapporteur Group for Z), www.cs.york.ac.uk/˜ian/zstan/

Queries

1. [Query1]– Please define abbreviation JML on first use
2. [Query2]– Please define abbreviation SLAM on first use
3. [Query3]– Please define abbreviation PVS on first use
4. [Query4]– Please check: should this read OBDDs? Otherwise please define new abbreviation used here
5. [Query5]– Please define abbreviation VHDL on first use
6. [Query6]– Please define abbreviation RISC on first use
7. [Query7]– Please confirm abbreviation definition added for FDR on first use
8. [Query8]– Please define abbreviation CCS on first use
9. [Query9]– Please define abbreviation ARC on first use
10. [Query10]– Please define abbreviation TMN on first use
11. [Query11]– Please define abbreviation API on first use
12. [Query12]– Please define abbreviation SAL on first use
13. [Query13]– Please define abbreviation ESC on first use
14. [Query14]– Please complete reference
15. [Query15]– Please provide further information if possible
16. [Query16]– Please confirm publisher and location
17. [Query17]– Please confirm publisher location
18. The references Clarke and Kurskan (1990) and Roscoe (1994) are not cited in the text. So please provide citations for those references or remove the references from the reference list.
19. Please update the following references:
ú Alur R (2002) Mocha: modularity in model checking (Reference [3]),
ú Martin JMR, Huddart Y (2000) Parallel algorithms for deadlock and
livelock analysis of concurrent systems (Reference [63]).