Towards a Benchmark for Model Checkers of Asynchronous Concurrent Systems*

Diyaa–Addein Atiya, Néstor Cataño and Gerald Lüttgen

Department of Computer Science, The University of York, U.K.
{diyaa,catano,luettgen}@cs.york.ac.uk

Abstract. Benchmarks, such as the established ISCAS benchmarks of digital circuits, have been successfully used to compare the relative merits of many model–checking tools and techniques employed for verifying *synchronous* systems. However, no benchmark for model checkers of *asynchronous* concurrent systems, such as communications protocols and distributed controllers, currently exists. This not only prevents a transparent evaluation of technologies in the field, but also hinders the accumulation and communication of insights into why and where particular technologies work better than others.
This paper takes a first step towards establishing a benchmark for asynchronous concurrent systems. It first discusses the underlying challenges when dealing with model–checking technologies for such systems. A prototype benchmark is then proposed, which is the result of an extensive survey and systematic classification of asynchronous concurrent systems studied in the literature. Finally, the proposed benchmark is evaluated against an established benchmarking theory.

1 Introduction

Temporal–logic model checking [17] is an established field with a wealth of publications and no lack of tool support. Many publications include some sort of evaluation of a proposed technique or tool, often through experimentation with one or more case studies and performance comparisons against alternative techniques or tools. To achieve high academic standards, it is important that this evaluation is as transparent and objective as possible.

Benchmarks are well known for providing a standard context, together with a set of performance criteria, in which the performance of technologies can be rigorously examined. Thus, benchmarks can provide an impartial means for comparing different tools and techniques. Even more, they can also be used for the purpose of validation and thorough examination of research contributions. The importance of benchmarks has already been recognised in the model–checking community. For the class of *synchronous* systems and particularly digital circuits, researchers employ the widely–accepted ISCAS benchmarks [7, 8]. However, an equivalent benchmark does not exist regarding *asynchronous* concurrent

* Research funding was provided by the EPSRC under grant GR/S86211/01.

systems, such as communications protocols and distributed controllers, whose behaviour is governed by the principle of interleaving.

As a consequence, the typical paper proposing a new technology for model–checking asynchronous systems often selects case studies in a way that compares the author's work quite favourably against existing work; otherwise, the paper might not be accepted for publication [27]. However, the successful application of a new technology to a case study only testifies to the viability of the tool in which the technology is implemented, rather than its relative merits compared to other technologies. This stresses the need for benchmarking model checkers for asynchronous concurrent systems, in addition to comparing the features and availability of model checkers as has been done, e.g., by Brim et al. [9]. Also Avrunin [4] highlights the need for benchmarking and provides a set of recommendations in that direction. Leue et al. [33] assembled a database of Promela models, which points to academic and industrial case studies conducted with the SPIN model checker. Corbett [18] compiled various Ada tasking programs to evaluate deadlock detection methods, while the work of Jones et al. [28] focuses on the metrics for comparing parallel model checkers. Pelánek [38] also recognises the need for benchmarking and suggests that properties of state spaces can be used to analyse the performance of model checkers.

This paper. This paper presents a first step towards establishing a benchmark for model checkers of asynchronous concurrent systems. We restrict ourselves to traditional temporal–logic model checking, such as implemented in SPIN [25] and NuSMV [16], and exclude those tools where temporal–logic model checking has been combined with other decision procedures. The envisioned benchmark shall be usable by academics researching model–checking technology and practitioners who want to find out which technology suits best their application. In contrast to the related work mentioned above, we aim for a benchmark that covers the full range of asynchronous concurrent systems, from communications protocols to mutex protocols to controllers. However, our benchmark should not be a simple collection of system models, but should also classify models according to those of their characteristics that effect model–checking technology and performance, e.g., whether models are loosely or tightly coupled, what communication mechanisms they use, and their sizes. The benchmark shall additionally include pointers to the literature where the models have been studied, together with a summary of the reported results. The methodology adopted by us is adopted from the benchmarking theory of Sim, Easterbrook and Holt [42].

We have compiled a *prototype* benchmark by reviewing the model–checking literature, including conference proceedings, journals, and tool distributions. It is currently available as a technical report [3], but will soon be supported and made editable as a collaborative web–database, so that everyone in the model–checking community may contribute. Without the support of the community, no benchmark can be successful. We hope that this paper stimulates discussion on this topic within those in the community who are interested in verifying asynchronous concurrent systems.

2 Requirements and Challenges

We start our quest towards a benchmark for model checkers of asynchronous concurrent systems by compiling its requirements and highlighting its challenges. Given the importance of benchmarking, it is surprising that not much work on its foundations has been published. One notable exception is a paper by Sim, Easterbrook and Holt [42] which presents a theory for benchmarking and illustrates that theory by developing a benchmark for software reverse–engineering tools. In the following, we first briefly re–visit this theory and, on this basis, then discuss the challenges for designing our benchmark.

2.1 Benchmarking Theory

Sim et al. structure benchmarks into three components [42]:

1. **Motivating comparison.** This component shall identify the overall purpose of the benchmark, the technical comparison to be performed, and the sought effect of the results to be obtained.
2. **Task sample.** This is the main component of the benchmark. It comprises a set of examples to be used as representatives of the domain concerned.
3. **Performance measures.** This component states what kind of measurements ought to be taken. The objective is to provide an overall view of the relationship between various tools and technologies.

The theory also identifies seven requirements a benchmark needs to fulfill in order to be successful:

1. **Accessibility.** The benchmark shall be publicly available, which includes the task sample as well as any experimental results conducted with this task sample. The benchmark shall also be easy to understand and easy to use by parties having different levels of expertise in empirical studies.
2. **Affordability.** The cost of using the benchmark, including the cost of human effort and of software and hardware needed to collect the measurements, shall be in proportion to the required benefits.
3. **Clarity.** The specification of the benchmark shall be concise and precise; it shall not leave gaps for misinterpretation or exploitation of ambiguities.
4. **Relevance.** The task sample represented in the benchmark shall have a general scope and be representative of the domain of interest.
5. **Solvability.** It shall be possible to complete the task sample and conduct the required performance measurements.
6. **Portability.** The benchmark shall be impartial in the sense that it does not favour one tool or technique over another. It shall be abstract enough to be portable to different tools and techniques.
7. **Scalability.** The task sample shall vary in such a way that it is applicable to tools and techniques of different levels of maturity.

Our benchmark will follow the above structure and, after presenting it in Sec. 3, we will evaluate in Sec. 4 whether we have met the requirements of Sim et al.

2.2 Challenges

Before prototyping a benchmark for model checkers of asynchronous concurrent systems, we first identify several challenges for its development.

Modelling Language. Benchmarks must be portable between different model–checking tools. However, different tools have typically different languages for representing systems. Regarding model checkers for asynchronous concurrent systems, these languages range from implementation languages such as Java [22], to high–level modelling languages such as Promela [25], to abstract specification languages such as Petri nets [15].

As pointed out by Corbett et al. in [19], a first naive solution to the modelling language problem would consist in the construction of models by hand, but this is time–consuming and error–prone. Second, as proposed by Corbett in [18], a natural alternative would be to express task samples in terms of finite–state automata and then use automatic translations to the input languages of the various model checkers. Similar ideas underly the *Bandera toolkit* [5] for verifying Java programs and the *Symbolic Analysis Laboratory* (SAL) [40]. In Bandera, the Java source code and the temporal specifications are translated into an intermediate representation called *Jimple*, which is a sort of low–level Java code. After being abstracted and sliced, this code is then translated into a common representation called BIR which is used as input for supported model checkers, such as for SPIN [25] and NuSMV [16]. SAL is a framework where different verification techniques and tools for abstraction, theorem–proving and model–checking are integrated. The modelling language of SAL resembles that of SMV and may be used as target for transition–system–based translators. Other intermediate languages employed in automated verification include Verimag's IF [6] and Berkeley's CIL [36].

However, although it might be possible to generate equivalent translations so that validity of involved properties is preserved, the translation process can still blindly favour the performance of one particular tool over another. For example, straightforwardly translating Petri nets to Promela would model places as global variables and thus disable SPIN's partial–order reduction algorithms. Moreover, one of the main reasons for the existence of so diverse modelling languages is that modelling asynchronous concurrent systems allows for many levels of abstraction. Each language suits not only a particular level of abstraction but also a particular application domain. It seems that defining a single language is impossible. However, maybe one can agree on a three or four modelling languages aimed at different levels of abstraction. This would still allow for the benchmark to be portable and affordable.

As our benchmark should be considered as a first prototype, we believe that it would be premature for us to make this important design decision regarding modelling languages at this point of our work. We rather leave this issue open for future discussion within the model–checking community and restrict ourselves here to precisely *describe* rather than *model* systems. Our descriptions will be accompanied with pointers to the literature where concrete models can be found, and illustrated with figures depicting a system's architecture.

Finally, there is of course also the issue of which temporal logic to choose for modelling a system's properties, and one of the great debates has been about *linear–time* vs. *branching–time* temporal logics [17]. As stated in Appendix B of [25], "in practice there is no measure that can reliably tell which method can solve a given problem more efficiently." We feel that it is thus sufficient to classify temporal properties according to whether they describe *safety*, *liveness* or *fairness* properties.

Performance criteria. The ultimate goal for comparing model–checking tools is to establish a relationship between their merits with regard to performance and to gain insights into the different model–checking techniques they implement.

The present literature typically compares tools in terms of CPU time and memory usage which depend on the underlying computing platform. These criteria are not very helpful in establishing a sense of *relativeness* between different tools or techniques. More objective performance criteria can be employed when focusing on a single model–checking technology, such as explicit–state model checking or BDD–based symbolic model checking. In case of the former, one may measure the generated state–space sizes and the number of transitions traversed, and in the latter the peak and final number of BDD–nodes stored.

As for the modelling language challenge, we believe that a discussion within the wider community is necessary to pin down which performance measures ought to be taken. We believe that more objective criteria are needed, with appropriate metrics for measurement. For our prototype benchmark we restrict ourselves to reporting the measurements described in the literature.

Tools vs. techniques. Increasingly, model–checking tools employ not a single but several techniques. For example, the popular NuSMV tool [16] supports both BDD–based and SAT–based model checking, and SPIN implements a wide range of optimisation techniques, from partial–order reduction to bit–state hashing [25]. Thus, our benchmark should not blindly compare tools to each others. Rather, results from benchmarking should specify which aspects of the tools are considered and what options have been used. Many of today's tools have an overwhelming large number of command–line options that have a direct influence on a tool's performance and can be fine–tuned for any given case study.

Documentation. The description of the task sample must be clear and easy to understand. Also, to be representative of the domain of asynchronous systems, the benchmark shall include case studies from industry as well as academia. Unfortunately, apart from widely cited examples such as the *leader election protocol* [39], our experience suggests that the literature lacks proper documentation of reported cases studies. This is often due to space limitations when publishing research papers. Additionally, and particularly for industrial case-studies, the full models cannot be presented due to confidentiality agreements. To achieve a proper documentation of any task sample, we believe that collaboration of both academic researchers and industrial users of model–checking technology is indispensable.

3 The Proposed Prototype Benchmark

This section defines a prototype benchmark for model–checking asynchronous concurrent systems which we hope will trigger a wider discussion within the model–checking community about benchmarking such systems.

3.1 Motivating Comparison

The benchmark shall support researchers in analysing the relative merits of different tools and techniques for model–checking asynchronous concurrent systems. It shall also help practitioners in establishing an objective relationship between the performance of alternative tools, so as to be able to choose the right model checker for their application of interest.

3.2 Task Sample

In order to define an appropriate task sample for our benchmark, we have been surveying the literature for case studies and reported experiments of comparing model–checking tools and techniques for asynchronous systems. We encountered a number of examples multiple times, albeit in different contexts. Our analysis classified the published studies according to the following criteria:

Application domain. Asynchronous concurrent systems typically fall into one of four domains: (*i*) network/communications protocols, (*ii*) computer/-mutex protocols, (*iii*) controllers, and (*iv*) distributed algorithms.

Types of communication and degree of coupling. Types of communication are *shared variables* (SV), *handshake communication* (HC), i.e., rendezvous, and *buffered communication* (BC), i.e., communication through buffered message channels. The degree of coupling of systems can be classified as either *loose*, i.e., communication is mainly between adjacent processes, *medium*, i.e., communication is mainly between one process and the rest of the model, or *strong*, i.e., most processes take part in the communication.

Scalability. Many of the models in our task sample are scalable, e.g., in the number of processes participating in a protocol. This allows the models to *stress test* today's and future model checkers. Moreover, challenges could be set up determining to what degree a model can be scaled so that a given model checker can still manage to verify properties within given time and memory constraints.

Size/growth. Scalable models can be classified in the growth of their state space. The growth can be either *linear* (Lin.), *polynomial* (Pol.), or *exponential* (Exp.). For instance, the state space of the *GNU sliding window protocol* [20] grows exponentially when increasing the window size. Non–scalable models range from small, simple models like the *alternating bit protocol*, to models of moderate complexity like the *production cell* [31, 37], to large models like the *traffic alert and collision avoidance system* (TCAS II) [1]. In the sequel, we write N/A for indicating that no information on a system's state–space size has been made available in the literature.

Properties. Properties being verified of a model may be categorised as *safety*, *liveness* and *fairness* properties. Some verification techniques have been tailored to deal with important specific safety and liveness properties, namely *deadlock* or *livelock*, respectively.

Techniques. Model–checking techniques can be split into two branches: *explicit* model checking (Explicit) such as implemented in SPIN [25], and *implicit* model checking including *decision–diagram–based* (DD) and *SAT–based* model checking (SAT) such as implemented in NuSMV [16]. Each of these may be combined with further techniques, such as *partial–order reduction* [26] and *disjunctive partitioning* [12]. While this classification is satisfactory for our prototype benchmark, it might need to be refined in the future to accommodate new techniques as these emerge. At the moment we have simply added a generic category "Others" to cover somewhat less popular techniques, such as compositional model checking and symmetry reduction.

Significance. As an indication of the significance of a case study we firstly considered the number of times it was cited in the literature. In some cases, whole workshops were dedicated to the verification of a particular case study, such as the *production cell* [31]. Other examples, e.g., the *leader election protocol* [39], are frequently cited and studied by many researchers.
Secondly, we also checked whether systems have industrial significance, as does, e.g., the *space aircraft controller* presented in [21]. Thirdly, we paid attention to examples that showed interesting experimental results with model checking tools. For instance, verifying the *ITU–T multipoint communication service* led to improvements in the SPIN model checker [34].

So as to evenly populate the space defined by the above criteria, we have selected the task sample summarised in Table 1 for our prototype benchmark.[1] We expect the benchmark to evolve as the community gets more engaged in the topic of benchmarking model checkers of asynchronous concurrent systems. Note that some well–known systems have not been included, since these systems are already represented by other systems in the task sample. As examples, consider the *process scheduling problem* [24], the famous *Peterson mutex algorithm* [25], and the *circular queue problem* [10]. The first system has no more than 300 reachable states, whence we do not expect it to exhibit any substantial characteristics in addition to the *alternating bit protocol* or the *telephony model POTS*. The second system is a mutual exclusion protocol, of which many are included in the task sample. Finally, reported experiments of model checking the circular queue show that it exhibits the same properties as the included bounded buffer example of [41].

Each system included in the task sample is documented following a simple structure. The documentation starts with a brief informal introduction to the system concerned. Whenever possible, the informal description is also illustrated with a diagram, showing its abstract structure. Then, a summary of the system's

[1] Note that purists of benchmarking theories might question whether "techniques" should influence the choice of task sample. However, ignoring "techniques" might lead to a biased benchmark, favouring one technology over another.

characteristics and a summary of selected experiments reported in the literature is provided. Finally, there is space for any further remarks. For illustration, the following is the description of the *leader election protocol* [39] in our task sample.

Table 1. The task sample.

System	Comm.	Scalability	Size	Properties	Techniques	Significance
`Network/Communication Protocols`						
Alternating Bit Protocol [18]	BC Loose	–	10^2	Deadlock Liveness	DD, Explic.+POR SAT, Others	Citations
GIOP [29]	HC,BC Medium	#Processes	N/A	Safety Liveness Deadlock Livelock	Explic.+POR Others	Application
GNU I-Protocol [20]	BC Loose	Window Size	Exp.	Deadlock Livelock	DD, Explic.+POR	Tools Application
ITU-T Service [34]	SV, BC Strong	#Nodes	N/A	Safety	Explic.+POR	Application
SIS Protocol [13]	BC Strong	–	N/A	Safety Liveness	Explic.	Application
Telephony Model POTS [30]	HC, BC Medium	–	10^5	Safety Liveness Deadlock	Explic. Others	Tools
`Computer/Mutex Protocols`						
Bakery Algorithm [11]	SV Strong	#Customers	N/A	Safety Liveness	DD, SAT	Citations Tools
Bounded Buffer Prod.–Consumer [41]	SV Loose	Buffer Size #Producers #Consumers	Exp.	Safety Liveness	DD, SAT	Citations Tools Application
Dining Philosophers [2]	SV, HC Loose	#Philosophers	N/A	Deadlock	DD, Explic.+POR SAT, Others	Citations
Queens Problem	SV Loose	#Queens	Exp.	Safety	DD, Others	Citations
Readers-Writers [18]	SV Strong	#Readers #Writers	Pol.	Safety Liveness Deadlock	DD, Explic.+POR SAT	Citations
Sleeping Barber [2]	SV Loose	#Chairs	N/A	Safety Livelock	DD, SAT	Citations
`Controllers`						
Cash Machine [44]	SV, BC Medium	#Tills #Users	N/A	Safety Liveness	Explic.+POR	Citations Application Tools
CDMA Library [14]	SV, HC Strong	#Mobiles	N/A	Safety Liveness Deadlock	Explic.	Application

continued on next page

continued from previous page						
System	**Comm.**	**Scalability**	**Size**	**Properties**	**Techniques**	**Significance**
Cyclic Scheduler [35]	HC Loose	#Cyclers	Exp.	Deadlock	DD, Explic.+POR SAT	Citations
Gas Station [23]	HC Strong	#Pumps #Customers	Exp.	Safety	DD, Explic.+POR	Citations Application
Production Cell [31]	HC Loose	–	10^{12}	Safety Liveness	Explic.+POR	Citations Application
Space Craft Controller [21]	SV Medium	#Properties #Tasks	N/A	Safety	Explic.+POR	Application
TCAS II [1]	N/A	–	10^{65}	Safety	DD	Application
`Distributed Algorithms`						
Cache Coherence for Distributed File System [46]	HC, BC Medium	#Servers #Clients #Files	Exp.	Safety	DD	Citations
Leader Election [39]	HC, BC Medium	#Processes	Pol.	Safety Liveness	DD, Explic.+POR Others	Citations

Documentation example. ***Leader Election Protocol – Introduction:*** Depicted in Fig. 1, the leader election protocol allows a ring of N processes to elect a leader through sending messages around the ring. Each process randomly chooses a number *id*, from a restricted set of values, and sends it around the ring. If there is a unique *id*, then the process with the maximum unique *id* is elected as the leader. Otherwise, the election step is repeated.

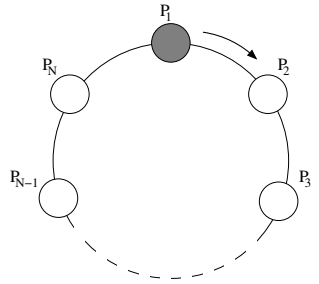

Fig. 1. Top view of the protocol.

System characteristics:

Domain: Distributed Algorithms
Communication: Both handshake and buffered communication
Degree of coupling: Loose
Scalability: Yes, in the number of processes
Size/Growth: Polynomial
Verified properties: Liveness

Selected experiments:

Reference	Technique	Tool	Source code
[24]	Explicit, partial–order reduction	SPIN	N/A
[32]	Directed Model Checking	HSF–SPIN	N/A

Aside. The protocol has also been studied elsewhere, but the couple of references above are sufficient for illustrating our documentation style.

Reported Results:

In [24], two versions of the protocol are specified in Promela and verified using SPIN, with 5 as the total number of processes and with partial–order reduction enabled. The first version requires all processes to participate in the election right from the beginning. In the second version, a process can decide to participate at a later point. Two properties were verified for each version, one safety and one liveness property. The number of reachable states is around 200.

In [32], the emphasis is on studying the results of combining directed model checking with partial–order reduction. The protocol is specified in Promela and verified for safety properties using HSF–SPIN. Results show that combining directed–search strategies with partial–order reduction can lead to significant improvements in terms of state–space size and run time. However, for the leader election protocol, the tool was not able to verify the required safety property.

3.3 Performance Measure

Our prototype benchmark adopts a qualitative approach for comparing tools. As illustrated in the above "Reported Results" section for the documentation of the leader election protocol, we point out experiments from the literature and the performance criteria they used. Then we report which tool or model–checking technique proved higher in merit compared to the other tools or techniques applied to the given example. For this initial stage of the development process of our benchmark, the qualitative result can work as a guide for other researchers who would like to experiment with the benchmark. Say, for example, that we are interested in comparing a new model checker to SPIN. We can then use the "Reported Results" sections in the task sample to identify the set of examples where SPIN is held at high esteem and apply the new model checker to that particular set of examples.

The problem with the proposed qualitative approach is that it allows for picking subsets of the benchmark to experiment with. In turn, this could favour some tools over others, contradicting the objectivity of benchmarking. Also, tracking the cited experiments described in the literature requires some non–negligible effort. A better approach would be to provide a list of measurable criteria and then report the performance of each model–checking tool according to these criteria. However, as discussed in Sec. 2.2, defining these criteria and providing the empirical data of experiments is a difficult task which requires consensus and collaboration from the wider community.

4 Evaluation

We evaluate our work against both the challenges identified in Sec. 2.2 and the requirements of the benchmarking theory of Sim et al. [42].

4.1 Meeting the Identified Challenges

It is important to note that the above benchmark is intended as a prototype only, which should provide a starting point for discussions among the model–checking community. We thus deliberately tried to avoid early design decisions and leave them open for broad discussions and agreements. In particular, the proposed benchmark avoids commitments regarding both concrete languages in which to model asynchronous concurrent systems and specific performance measures which ought to be taken when conducting experiments.

As argued in Sec. 2.2 we believe that identifying a single modelling language will prove difficult, if not impossible, given that model checking is used at various stages of the systems engineering life cycle, namely in the requirements, design and testing stages. Instead, our task sample is described in plain English, with pointers to the literature where formal models of each asynchronous system model can be found. It is also equipped with a summary of each model's characteristics and highlights which model has been studied using which model–checking technologies. This should satisfy both the academic researcher who, e.g., can compare their work on a specific technology using that part of the task sample to which this technology has already been studied by others, as well as the user of model checkers who, e.g., can use the benchmark to identify which technology seems to cope best with the characteristics of their applications at hand.

Our prototype benchmark also leaves open the exact performance measures to be adopted. The performance measures currently used in the literature are either dependent on the deployment platform, such as absolute time and memory measurements, or on the employed technology, such as counting the peak or final number of BDD nodes, as discussed in Sec. 3.3. Instead, for each example in the task list, we identify in the summary of "reported results" section the experiments that have been given in the literature, regardless of the performance measures used in these experiments. Also, the results from these experiments are reported in a qualitative rather than a quantitative fashion. In this format, the prototype benchmark can not only be used as a guide for researchers but also remains generic and easy to update with a formal definition of the performance measures' component.

4.2 Meeting the Benchmarking Requirements

As summarised in Sec. 2.1, the benchmarking theory of Sim et al. [42] states seven essential requirements for the success of a benchmark. In the following we evaluate our prototype benchmark against each of these requirements.

Accessibility. At the moment our prototype benchmark will only be available online as a technical report [3]. As explained in Sec. 3.2, the task sample is illustrated through text and graphical representations, with appropriate references to the literature. Thus, the prototype benchmark is publicly accessible and easy to understand. However, in this format it is not straightforward for other

researchers to contribute to the benchmark in terms of updating experimental results or modifying the task sample. We are currently addressing this issue through the implementation of a collaborative web–database for the benchmark, which will be completed by April 2006. This will allow members of the community to edit the benchmark's content, much as the popular online encyclopedia *Wikipedia* [45] does.

Affordability. The prototype benchmark identifies experiments from the literature and points out the results and, if available, the source code necessary for repeating these experiments. Unfortunately, chasing the provided citations or web links can incur a significant price in terms of time and effort. However, using the prototype benchmark as a starting point is much easier than starting from scratch, as the literature survey and reported results are already provided. With its implementation as a collaborative web–database, using the benchmark will become more cost–effective.

Clarity. Despite our efforts to be precise, we currently provide no formal definitions of the terms we use but assume mutual understanding. Future work shall support the benchmark with a dictionary of terminology.

Relevance. The proposed task sample includes case studies from both academia and industry. Also, these case studies have been the subject of experiments for various and widely used model checkers, implementing different model–checking techniques. Thus, the prototype benchmark has a global scope and can be considered representative of its domain, i.e., asynchronous concurrent systems.

Solvability. This criteria is trivially satisfied for our prototype benchmark since all problems in the task sample have been studied in the literature before, using one or more model–checking techniques and tools.

Portability. The proposed task sample includes case studies that vary in sizes, types of communication, and application domains. Also, the benchmark abstracts away from specifying the modelling language for the task sample. Together, these give the benchmark an impartial view of the domain and allow it to be used with different model–checking tools and techniques.

Scalability. Again, our task sample includes case studies that vary in sizes. Many of them are parameterised, and their complexities grow when increasing the sizes of their input parameters. Thus, our benchmark can be tuned to suit tools and techniques of different levels of maturity, not only today but also in the future when both model checkers and work stations will have evolved.

5 Conclusions & Future Work

Benchmarks are an effective means for examining and evaluating technologies, as is proved by the ISCAS benchmarks [7, 8] for model checkers of synchronous systems and particularly digital circuits. Our proposed benchmark focuses instead on asynchronous concurrent systems. This subsumes the important class of communications protocols and distributed controllers, for which there are many

reported experiments on model–checking technologies but no obvious ways for comparing and verifying their results.

This paper discussed the challenges for developing a widely–agreeable benchmark for model checkers of asynchronous concurrent systems and introduced and evaluated an according *prototype* benchmark. In contrast to related work [10, 18, 28], our prototype benchmark follows a methodical definition according to a clear benchmarking theory [42]. The proposed task sample ranges across various application domains, satisfies a variety of characteristics, and includes a summary of relevant experiments reported in the literature. Moreover, many of the models included in the task sample are scalable so that the benchmark can challenge both present and future model checkers.

Our prototype benchmark is intended for use by academic researchers and industrial developers. In academia, the benchmark should primarily be employed for validating research results, thus leading to a more transparent approach of conducting experiments and helping to accumulate insight into the relative strengths of the many model–checking technologies available. We believe that this will eventually lead to higher–quality research papers, with robust and verifiable evaluations. Industrial developers are likely to use the benchmark for selecting the right model–checking tool or technique for a given development project, comparing the characteristics of the application at hand to those of the system models in the task sample, and checking which technology is known to be the most promising one for that application.

Future work. We hope that our work will initiate a constructive dialogue which will eventually result in a standardised benchmark with a representative task sample and a well–defined set of performance criteria. To facilitate this process, our benchmark must allow for truly *collaborative* efforts.

We are currently implementing a wiki–based web–accessible database for managing the benchmark [45], which should be available by April 2006. Users can then remotely edit the task sample, update reports on experimental results and add pointers to the literature. Of course, traditional database functionalities would also be supported, such as searching the benchmark or restricting the view to those system models that satisfy a particular characteristic. This will allow our prototype benchmark to go further beyond being a documented collection of case studies. As noted in [42], the benchmark will then be able to act as a tool for achieving a more cohesive understanding of the domain, a more rigorous examination of research results, and faster technical progress.

At the recent "Satisfiability–Modulo–Theories Competition (SMT-COMP)" event affiliated with CAV 2005, a SMT solver competition was carried out. Tools needed to be able to accept given task samples coded in a standard format [43], and performance criteria included giving penalty points to a tool that solved a problem incorrectly. We believe that a similar competition should be carried out for model checking tools, probably split into different categories, depending whether digital circuits, software or timed systems are model checked. This would further encourage faster and more verifiable progress within the community.

References

1. R.J. Anderson et al. Model checking large software specifications. In *SIGSOFT '96*, pp. 156–166. ACM Press, 1996.
2. G.R. Andrews. *Concurrent Programming*. Benjamin Cummings, 1991.
3. D. Atiya, N. Cataño, and G. Lüttgen. A benchmark for model checking asynchronous concurrent systems. Techn. rep. U. York, 2005. Available at `http://www.cs.york.ac.uk/~luettgen/publications/benchmark.pdf`.
4. G.S. Avrunin et al. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE TSE*, 17(11):1204–1222, 1991.
5. The Bandera project. See `http://bandera.projects.cis.ksu.edu/`.
6. M. Bozga et al. IF: An intermediate representation and validation environment for timed asynchronous systems. In *FM '99*, vol. 1708 of *LNCS*, pp. 307–327, 1999.
7. F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *ISCAS '89*, pp. 1924–1934. IEEE Press, 1989.
8. F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits. In *ISCAS '85*, pp. 695–698. IEEE Press, 1985.
9. L. Brim et al. YAHODA: Verification tools database. Faculty of Informatics, Masaryk U. Brno. Available at `http://anna.fi.muni.cz/yahoda/`.
10. T. Bultan. BDD vs. constraint-based model checking: An experimental evaluation for asynchronous concurrent systems. In *TACAS '00*, vol. 1785 of *LNCS*, pp. 441–456, 2000.
11. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM TOPLAS*, 21(4):747–789, 1999.
12. J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *Very Large Scale Integration*, pp. 49–58. North-Holland, 1991.
13. S. Chaki, S.K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *POPL '02*, pp. 45–57. ACM Press, 2002.
14. S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: An industrial case study. In *ICSE '02*, pp. 431–441. ACM Press, 2002.
15. G. Ciardo and A.S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *IPDS '96*, p. 60. IEEE Press, 1996.
16. A. Cimatti et al. Integrating BDD-based and SAT-based symbolic model checking. In *FroCoS '02*, vol. 2309 of *LNCS*, 2002.
17. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
18. J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE TSE*, 22(3):161–180, 1996.
19. J.C. Corbett et al. Bandera: Extracting finite-state models from Java source code. In *ICSE '00*, pp. 439–448. IEEE Press, 2000.
20. Y. Dong et al. Fighting livelock in the i-protocol: A comparative study of verification tools. In *TACAS '99*, vol. 1579 of *LNCS*, pp. 74–88, 1999.
21. K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space craft controller using SPIN. *IEEE TSE*, 27(8), 2001.
22. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000.
23. D. P. Helmbold and D. C. Luckham. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, 1985.
24. G.J. Holzmann. The model checker SPIN. *IEEE TSE*, 23(5), 1997.

25. G.J. Holzmann. *The SPIN Model Checker*. Addison Wesley, 2003.
26. G.J. Holzmann and D. Peled. An improvement in formal verification. In *Formal Description Techniques*, pp. 197–211. Chapman & Hall, 1995.
27. J.N. Hooker. Testing heuristics: We have it all wrong. *J. Heuristics*, 1:33–42, 1996.
28. M. Jones, E.G. Mercer, T. Bao, R. Kumar, and P. Lamborn. Benchmarking explicit state parallel model checkers. *ENTCS*, 89(1), 2003.
29. M. Kamel and S. Leue. Validation of a remote object invocation and object migration in CORBA GIOP using Promela/SPIN. In *SPIN '98*, Paris, France, 1998.
30. M. Kamel and S. Leue. VIP: A visual editor and compiler for v-Promela. In *TACAS '00*, vol. 1785 of *LNCS*, pp. 471–486, 2000.
31. C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell*, vol. 891 of *LNCS*. 1995.
32. A. Lluch-Lafuente, S. Edelkamp, and S. Leue. Partial order reduction in directed model checking. In *SPIN '02*, vol. 2318 of *LNCS*, pp. 112–127, 2002.
33. A. Lluch-Lafuente and S. Leue. Database of Promela models, 2002. Available at `http://www.informatik.uni-freiburg.de/~lafuente/models/models.html`.
34. P. Merino and J.M. Troya. Modeling and verification of the ITU-T multipoint communication service with SPIN. In *SPIN '96*, Rutgers U., New Jersey, 1996.
35. R. Milner. *A Calulus of Communicating Systems*, vol. 92 of *LNCS*. 1980.
36. G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02*, vol. 2304 of *LNCS*, pp. 213–228, 2002.
37. D.O. Paun, M. Chechik, and B. Biechelle. Production cell revisited. In *SPIN '98*, Paris, France, 1998.
38. R. Pelánek. Typical structural properties of state spaces. In *SPIN '04*, vol. 2989 of *LNCS*, pp. 5–22, 2004.
39. M. Raynal. *Distributed Algorithms and Protocols*. Wiley, 1992.
40. Symbolic Analysis Laboratory. See `http://sal.csl.sri.com`.
41. A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM CSUR*, 25(3):225–262, 1993.
42. S.E. Sim, S. Easterbrook, and R.C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *ICSE '03*, pp. 74–83. IEEE Press, 2003.
43. The SMT-LIB Standard. `http://goedel.cs.uiowa.edu/smtlib/papers.html`.
44. J. Tang and W. Zhou. Cash point with Promela/SPIN. Techn. Rep. CSRG-407, U. Toronto, 2000.
45. Wikipedia, 2005. An online encyclopedia available at `http://www.wikipedia.org`.
46. J.M. Wing and M. Vaziri. Model checking software systems: A case study. In *SIGSOFT '95*, pp. 128–139. ACM Press, 1995.

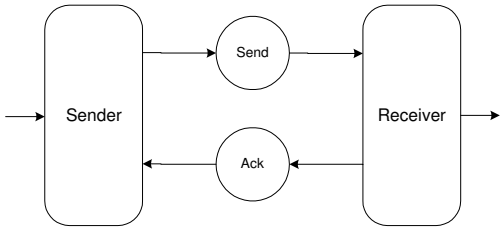

Fig. 1. An overview of the alternating bit protocol.

A Case Studies

We here give the full description, together with selected reported results, for the "task sample" proposed in our prototype benchmark. Although the model checking domain enjoys a vast literature, experiments with cases studies are rarely reported with full evaluation results; experiments that compare several tools are even more scarce. We certainly cannot claim full cover of the literature for the results we cite here. However, our selected experiments do enjoy the application of several tools to one or more examples, or the application of one tool to several examples. Otherwise, i.e. where one tools is applied to one example, the reported experiment is selected because, as in A.4 and A.13 for example, the experiment provides more insights into the model checking technology than just "*... the tools was capable of finding a bug.*".

A.1 Alternating Bit Protocol

Introduction: The alternating bit protocol is simple and frequently cited example for transferring messages in one direction between a two points of a network. As depicted in Figure 1 the protocol consists of a sender, a receiver, and two lossy channels. A message is sent continuously, with the same identification number (0 or 1) until an acknowledgment, containing that number, is received. At this point, the sender flips the identification number and starts sending the next message.

System characteristics:

Domain:	Network/Communication protocols
Communication:	Buffered communication
Degree of coupling:	Loose
Scalability:	–
Size/Growth:	100
Reported verified properties:	Liveness and deadlock-freedom

Selected experiments:

Reference	Technique	Tool	Source code
[16]	Explicit, partial-order reduction	SPIN	N/A
[16]	BDD-based	SMV	N/A
[16]	Inequality Necessary Conditions	INCA	N/A
[23]	Directed Model Checking	HSF-SPIN	N/A
[6]	BDD-based	SMV, ALV	[6]

Reported Results:

– In [16], the problem is modelled in Ada [5] and then automatically translated into the appropriate input language for the three tools SMV [36], SPIN [27] and INCA [3, 17]. The tools are used for detecting deadlock in the model, and are compared according to the consumption of memory and CPU time. The reported results for the alternating bit protocol are:

	SPIN		SPIN+PO		SMV		INCA	
#States	Mem(mb)	Time(s)	Mem(mb)	Time(s)	Mem(mb)	Time(s)	Mem(mb)	Time(s)
113	1.29	0.67	1.71	0.84	1.70	1.03	7.13	7.23

Experiments were conducted on a SPARCstation 10 Model 51 with 96 MB of memory. The memory is measured in MB and CPU time is measured in seconds.

– In [23], the emphasis is on studying the results of incorporating heuristics search algorithms into the explicit model checker SPIN. The protocol is specified in Promela and verified for LTL-specified liveness properties using HSF–SPIN. Results showed that HSF–SPIN was more efficient than SPIN in verifying liveness properties for the alternating bit protocol. However, this result is not conclusive as experiments with other examples showed that heuristics can improve the search process only if they have very specific knowledge of the system considered.

– In [6], the main objective is the efficient construction of BDD representations for integer arithmetic constraints. An algorithm for generating BDDs for *linear* arithmetic constraints is proposed and shown to produce BDDs with sizes that are linear in the number of variables involved and the number of bits used to encode each variable. However, that result was only valid when all the variables had the same bounds, or different bounds but with all are powers of two. When the variables have different bounds, which are not necessarily power of two, the size of the generated BDDs grows exponentially as the number of variables is increased. Consequently, as noted in [6], this suggests that if the choice of bounds does not compromise the specification then it may be a good idea to choose these bounds as powers of two.

The proposed algorithm is incorporated into the Action Language Verifier (ALV) tool [10]. Then, using the alternating bit protocol and other examples as a testbed, the performance of ALV is compared to three different implementations of SMV (CMU SMV version 2.5.4.3, Cadence SMV version

08-20-01p2, and NuSMV version 2). The experiments verified both safety and liveness properties and were conducted on SUN ULTRA 10 workstation with 678 MB of memory, running SunOs 5.7. The results have shown that, the recorded execution times for the three implementations of SMV grow exponentially in terms of the size the involved variables. ALV, on the other hand, has shown linear growth in execution time. Nevertheless, other experiments in [6] have shown that the size of BDDs generated by ALV grows exponentially if the involved variables have different bounds and these bounds are not all power of two.

A.2 GIOP

Introduction: The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) for communication between distributed applications over heterogeneous networks. Central to CORBA is the implementation of Object Request Brokers (ORB) which enables client/server object interaction between applications. Communications between ORBs are independent of the implementation language and the platform of the communicating applications. The standard protocol for achieving this, as defined by CORBA, is the General Inter-ORB Protocol (GIOP).

System characteristics:

Domain:	Network/Communication protocols
Communication:	Handshake and buffered communication
Degree of coupling:	Medium
Scalability:	Yes, in the number of processes
Size/Growth:	N/A
Reported verified properties:	Safety, liveness, deadlock- and livelock-freedom

Selected experiments:

Reference	Technique	Tool	Source code
[30, 31]	Explicit, partial-order reduction	SPIN	Online
[23, 34]	Directed Model Checking	HSF-SPIN	N/A

Reported Results:

– In [30, 31], the GIOP protocol is verified, using SPIN, for LTL-specified safety and liveness properties, including absence of deadlock and livelock. The difference between [30] and [31] is in the modelling process. In [30] a hand-built model of GIOP was developed and validated in Promela, and then verified using SPIN; the model assumes two servers processes, two agents, two users, and one client. In [31] an equivalent model is presented in the v-Promela language, which is a graphical extension of Promela. Then the *Visual Interface to Promela* (VIP) is used to translate the graphical model into Promela code. Experiments for verifying the model properties have shown that the VIP generated code requires larger state vectors, but results in much smaller state

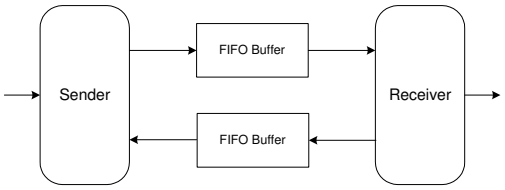

Fig. 2. An overview of the GNU i-protocol.

spaces. The improved performance of the VIP is attributed by [31] to two man reasons. First, the fact that VIP generated code uses goto statements, rather than an event loop construct (in the hand-built model) in which control states are represented through variables. Second, the fact that channels are all represented as global variables in the hand-built model, whereas they are represented as local variables (of the `Env` process) in the VIP generated code.

– In [23, 34], the model of GIOP (presented in [30]) is used and verified for LTL-specified properties using HSF–SPIN. In [34], the emphasis is on studying the results of incorporating heuristics search algorithms into SPIN. In that work the GIOP protocol is verified for safety properties. In [23] the emphasis is on studying the effect of combining directed model checking with partial-order reduction; the GIOP protocol is verified for absence of deadlock and for other liveness properties. Results showed that HSF–SPIN was more efficient than SPIN in verifying safety and liveness properties for the GIOP protocol. However, this result is not conclusive as experiments with other examples showed that heuristics can improve the search process only if they have very specific knowledge of the system considered.

A.3 GNU I-Protocol

Introduction: The GNU i-protocol is a sliding window protocol which allow for remote execution of commands and transfer of data between Unix computers. The protocol is optimised for reducing the acknowledgment and retransmission traffic. As depicted in Figure 2, the protocol can be modelled as an asynchronous system comprising a sender process, a receiver process, and two (lossy) FIFO buffer processes. One buffer is used by the sender process to send the data packets to, and receives acknowledgments from, the receiver process. The other buffer is used by the receiver process to receive data packets from, and send acknowledgments to, the sender process. If the window size for the i-protocol is W, then the sender process can send up to W contiguous packets without waiting for acknowledgments from the receiver process.

System characteristics:

Domain:	Network/Communication protocols
Communication:	Buffered communication
Degree of coupling:	Loose
Scalability:	Yes, in the window-size
Size/Growth:	Exponential
Reported verified properties:	Deadlock- and livelock-freedom

Selected experiments:

Reference	Technique	Tool	Source code
[22]	Explicit	Concurrency Factory, Murφ, XMC	Online
[22]	BDD-based	COSPAN, SMV	Online
[22]	Explicit, partial-order reduction	SPIN	Online
[28]	Explicit, partial-order reduction	SPIN	N/A

Reported Results:

- In [22], the i-protocol is used as a testbed for comparing six model checkers: The Concurrency Factory [15] COSPAN [25] (version 8.15), Murφ [20] (version 3), SMV (version 2.4), SPIN (version 2.9.7), and XMC [42]. Three parameters are toyed with in order to obtain eight variations of the protocol; the existence/absence of a livelock error, the window size (1 or 2), and the existence/absence of message corruption during communication. The protocol model is written for each of the tools in its own input language separately. The tools are then used to detect a livelock error in an erroneous version of the protocol. Afterwards, with the livelock error fixed, experiments with tools are repeated to verify that the new version of the protocol is both livelock- and deadlock-free. Experiments are then repeated with $W = 1$ and $W = 2$ and with buffers that can only drop messages versus ones that can also corrupt messages. The performance of tools is measured in terms of the number of states explored, the number of transitions traversed, CPU time usage and memory usage. Experimental results have shown that the explicit model checkers Murφ and XMC have performed better than all the other tools, in terms of completing all experiments successfully, and efficient use of memory and CPU time. Next performances are the SPIN and COSPAN tools. The poorest performance was the SMV tool – though it showed efficient memory-usage on all experiment with $W = 1$, the tool failed to complete in a reasonable amount of time for $W = 2$.
- In [28] the experiments in [22] are repeated using the same version of SPIN (2.9.7) and the same Promela model of the i-protocol, but this time with two corrections of the way SPIN is used. Namely, the "stacksize" parameter reduced, and the loss-less COLLAPSE compression option is enabled. The result is that SPIN outperformed XMC in six of the eight variations of the i-protocol. Editing the Promela model and adopting some of the compression

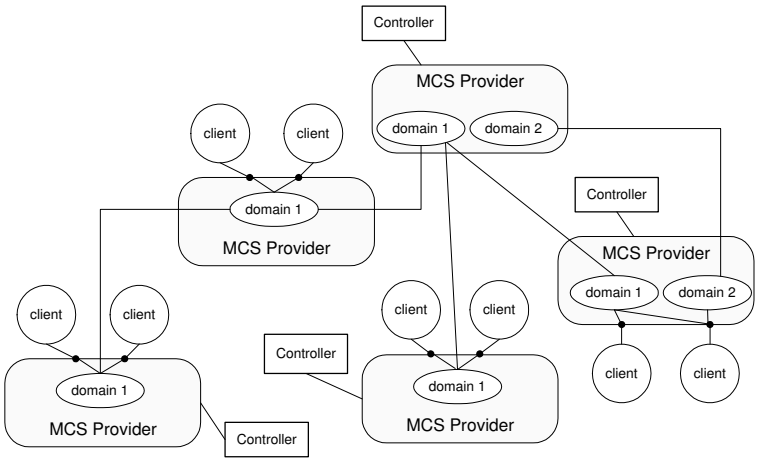

Fig. 3. Two MCS domains.

techniques used in XMC into a new version of SPIN (3.3.0) has then resulted in the latter outperforming XMC in all the eight variations of the i-protocol.

Further Comments: So far it looks like explicit model checking outperforms symbolic model checking (more precisely, SPIN outperforms SMV) in verifying (asynchronous) network/communication protocols

A.4 ITU-T Service

Introduction:

Overview The ITUT Multipoint Communication Service (MCS) is a multiparty communication protocol which provides support for interactive multimedia conferencing applications. The protocol allows applications to send data, using a single primitive, to one or more destinations. Depicted in Figure 3 The MCS is organised around the concepts of *domains*. Each domain groups a set of *client* applications that exchange data within that domain. The dynamic construction and deconstruction of a domain is the responsibility of the domain *controller*. For example, a domain is created when the controllers of two MCS providers agree on a new connection between two clients – a client can be part of different domains. A client can exchange data with other clients in the domains it is attached to. This multipoint communication is supported with multipoint channels, and messages are propagated through the whole tree structure of the domain to which the client is attached.

System characteristics:

Domain:	Network/Communication protocols
Communication:	Shared variables and buffered communication
Degree of coupling:	Strong
Scalability:	Yes, in the number of nodes
Size/Growth:	N/A
Reported verified properties:	Safety and liveness

Selected experiments:

Reference	Technique	Tool	Source code
[37]	Explicit, partial-order reduction	SPIN	Online

Reported Results:

– In [37], a simplified version of the MCS protocol is specified in Promela, and verified for LTL-specified safety properties using SPIN (version 2.8.4). However, more than just verifying the protocol, the work in [37] aimed at evaluating the capabilities of Promela and SPIN for specifying and verifying multi-point communication protocols. Experiments showed that SPIN is suitable for detecting errors in such protocols. However the dynamic nature of the MCS protocol has lead to unnecessarily large code in Promela. As a result, some extensions to Promela have been proposed in order to facilitate direct representation of dynamically created processes.

A.5 SIS Protocol

Introduction: The Service Incident-exchange Standard (SIS) [19] is communication protocol that allows incident tracking systems to share data and facilitate resolutions. The basic idea is to represent the various status of a service request as a finite state machine (FSM). Copies of this FSM can then be shared between service requesters and providers so that these parties can keep track of the request at various points of time. Of course, the requesters and providers have to maintain a consistent view of the FSM. Figure 4 provides an overview of the SIS protocol. Each of the Requester and Provider consists of two components:

– The Client: represented by RClient and PClient. The client is responsible for transmitting transaction requests.
– The Virtual Machine (VM): represented by RVM and PVM. The VM represents the Requester's(/Provider's) copy of the request FSM. Each VM is responsible for processing transaction requests from the corresponding client. To maintain consistency between the two VM, when one of them successfully processes a transaction request from the client, it then issues a request for an appropriate state change in the other VM.

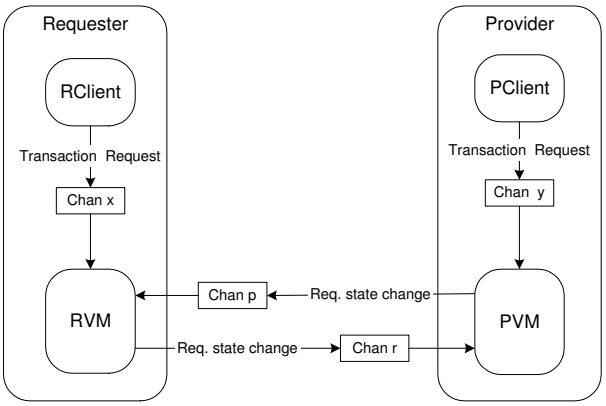

Fig. 4. An overview of the SIS protocol.

System characteristics:

Domain:	Network/Communication protocols
Communication:	Buffered communication
Degree of coupling:	Strong
Scalability:	–
Size/Growth:	N/A
Reported verified properties:	Safety

Selected experiments:

Reference	Technique	Tool	Source code
[11]	Explicit Model Checking	PIPER, SPIN	N/A

Reported Results:

– The work in [11] focuses on automatic extraction of abstract CCS models from π-calculus specifications, using annotations in the form of type signatures, and the verification of these CCS models using model checking techniques. The SIS is specified as a set of communicating π processes, with corresponding type signatures. The properties of the system are states as LTL formulae. The PIPER tool (a back end to the model checker SPIN) is then used to automatically extract the abstract CCS processes, as well as necessary subtyping proof obligations. PIPER then automatically translate the CCS models into Promela specifications. Finally, the SPIN tool is used to verify the temporal properties and to discharge the subtyping proof obligations – the proof obligations are discharged through simulations between the CCS processes. No specific verification results, e.g. memory consumption or size of generates state space, are given in [11].

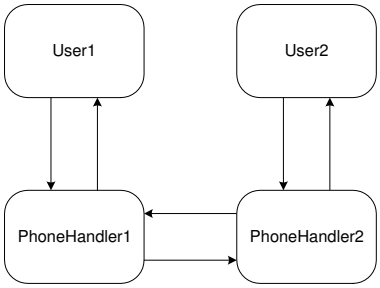

Fig. 5. A simple POTS system

A.6 Telephony Model POTS

Introduction: Plain Old Telephone Service (POTS) is the analogue telephone service which was available prior to the introduction of electronic telephone exchanges. In the POTS model, users are connected through wires to a nearby central office. The central office acts as a switch point, and is connected to other central offices and long-distance facilities. A simple POTS, consisting of two telephones and two switch points, is illustrated in Figure 5. Respectively, the User and PhoneHandler processes represent the telephones and the call processing software at the switch points. Each User process communicates with its designated PhoneHandler. A PhoneHandler process responds to events from its User and communicates with the other PhoneHandler to establish connections.

System characteristics:

Domain:	Network/Communication protocols
Communication:	Handshake and buffered communication
Degree of coupling:	Medium
Scalability:	–
Size/Growth:	10^5
Reported verified properties:	Safety, liveness, and deadlock-freedom

Selected experiments:

Reference	Technique	Tool	Source code
[31]	Explicit, partial-order reduction	SPIN	N/A
[23, 34]	Directed Model Checking	HSF-SPIN	N/A

Reported Results:

– In [31], the POTS system is verified, using SPIN, for LTL-specified safety and liveness properties, including absence of deadlock and livelock. The system model is represented in the v-Promela language, a graphical extension of Promela. Then the *Visual Interface to Promela* (VIP) is used to translate the graphical model into Promela code. Experiments have revealed deadlock

error, and that the model can only satisfies trivial liveness properties. However, it was possible to verify that the model can indeed establish a phone call; that is, "there exists a scenario in which both PhoneHandler processes are in the respective conversation states".

– In [23, 34], the model of POTS presented in [31] is used and verified for LTL-specified safety properties using HSF–SPIN. In [34], the emphasis is on studying the results of incorporating heuristics search algorithms into SPIN. In [23] the emphasis is on studying the effect of combining directed model checking with partial-order reduction. Results showed that HSF–SPIN was more efficient than SPIN in verifying the safety properties for the POTS model. However, this result is not conclusive as experiments with other examples showed that heuristics can improve the search process only if they have very specific knowledge of the system considered.

A.7 Bakery Algorithm

Introduction: The algorithm solves the mutual exclusion problem for $P_1, .., P_N$ processes accessing a shared critical section of code. Each process P_i has a shared number n_i that is readable by all other processes but writeable only by P_i. Initially all the numbers $n_1, .., n_N$ are equal to zero. When a process P_i wants to access the critical section it sets n_i to be greater than all other n_j $(j \neq i)$. P_i then waits for all other processes that has a lower number $n_j (\neq 0)$ – a tiebreaker is resolved for the favour of the process with the lower id. Finally, P_i accesses the critical section; upon leaving the process sets n_i back to zero.

System characteristics:

Domain:	Computer/Mutex Protocols
Communication:	Shared variable
Degree of coupling:	Strong
Scalability:	Yes, in the number of processes
Size/Growth:	N/A
Reported verified properties:	Safety and liveness

Selected experiments:

Reference	Technique	Tool	Source code
[7]	BDD-based	SMV	N/A
[7, 9]	Constraint-based	OMC	N/A
[4]	Implicit	SAL	Online
[4]	BDD-based	SAL	Online
[6]	BDD-based	SMV, ALV	[6]

Reported Results:

– In [7, 9], the main objective is the application of constraint-based model checking (which uses arithmetic constraints as a symbolic representations)

for verifying concurrent systems with unbounded integer variables. In [9] the bakery algorithm is verified for safety and liveness properties using the Omega library model checker (OMC) [32, 41]. The OMC tool uses Presburger arithmetic (integer arithmetic without multiplication) [8, 9] constraints as its underlying state representation and transition relations. In [7], with memory usage and execution time as the criteria, the performance of OMC is compared to the BDD-based model checker SMV for verifying safety properties of the bakery algorithm. Experiments were conducted on two stages.

In the first stage, with only two concurrent processes, the performance of the tools was studied against the increase in the domain range of the state variables. Results have shown that the performance of the OMC tool remains constant with respect to increasing variable domains. The execution time and the memory usage of SMV, however, increased exponentially with the number of boolean variables required for the binary encoding of each input variable. The experiments were repeated again but with more appropriate encoding of variable ordering in SMV. As a result, the execution time and the memory usage of SMV were reduced to a linear increase with respect to the number of boolean variables required for the binary encoding of input variables.

In the second stage the performance of SMV and OMC was studied with respect to increasing the number of concurrent processes in the algorithm, from 2 to 4. This time the performance of both SMV and OMC deteriorated significantly, with SMV having more graceful degradation than OMC. For 4 processes, SMV has finished in less than 20 seconds, while one hour was not enough for the OMC tool.

According to [7], the overall experimental results with SMV and OMC suggest that constraint-based model checking can outperform BDD-based model checking for verifying asynchronous concurrent systems with finite but large integer domains.

– In [4], John Rushby at SRI has prepared a tutorial on different methods of formal analysis, illustrating their strengths and limitations. The tutorial is mainly conducted around the PVS modelling and verification system, but as stated by the author of the tutorial, it is still valid in the framework of similar verification tools. In the tutorial, the Bakery algorithm problem is modelled in the SAL intermediate language and different formal analysis techniques such as finite-state model checking by explicit state enumeration and symbolic BDDs-based methods are illustrated.

– In [6], the main objective is the efficient construction of BDD representations for integer arithmetic constraints. An algorithm for generating BDDs for *linear* arithmetic constraints is proposed and shown to produce BDDs with sizes that are linear in the number of variables involved and the number of bits used to encode each variable. However, that result was only valid when all the variables had the same bounds, or different bounds but with all are powers of two. When the variables have different bounds, which are not

necessarily power of two, the size of the generated BDDs grows exponentially as the number of variables is increased. Consequently, as noted in [6], this suggests that if the choice of bounds does not compromise the specification then it may be a good idea to choose these bounds as powers of two.

The proposed algorithm is incorporated into the Action Language Verifier (ALV) tool [10]. Then, using the bakery algorithm and other examples as a testbed, the performance of ALV is compared to three different implementations of SMV (CMU SMV version 2.5.4.3, Cadence SMV version 08-20-01p2, and NuSMV version 2). The experiments verified both safety and liveness properties and were conducted on SUN ULTRA 10 workstation with 678 MB of memory, running SunOs 5.7. The results have shown that, the recorded execution times for the three implementations of SMV grow exponentially in terms of the size the involved variables. ALV, on the other hand, has shown linear growth in execution time. Nevertheless, other experiments in [6] have shown that the size of BDDs generated by ALV grows exponentially if the involved variables have different bounds and these bounds are not all power of two.

A.8 Bounded-Buffer Producer-Consumer Problem

Introduction: A bounded buffer is a queue with limited size. The buffer supports a *write* operation, in which a producer can insert a data item into the queue. The buffer also supports a *read* operation, in which a consumer can remove a data item from the queue. The buffer must not overflow, thus a producer cannot add items if the buffer is full. Also, a consumer should not be allowed to read from the queue if the buffer is empty. To achieve these functionalities, proper synchronisation should be provided between the producer and consumer for accessing the buffer. The problem is even more interesting when there are more than one producer and/or consumer.

System characteristics:

Domain:	Computer/Mutex Protocols
Communication:	Shared variable
Degree of coupling:	Loose
Scalability:	Yes, in the buffer size and the number of producers/consumers
Size/Growth:	Exponential
Reported verified properties:	Safety and liveness

Selected experiments:

Reference	Technique	Tool	Source code
[7]	BDD-based	SMV	N/A
[7, 9]	Constraint-based	OMC	N/A
[6]	BDD-based	SMV, ALV	[6]

Reported Results:

- In [7, 9], the main objective is the application of constraint-based model checking (which uses arithmetic constraints as a symbolic representations) for verifying concurrent systems with unbounded integer variables. In [9] the bakery algorithm is verified for safety and liveness properties using the Omega library model checker (OMC) [32, 41]. The OMC tool uses Presburger arithmetic (integer arithmetic without multiplication) [8, 9] constraints as its underlying state representation and transition relations. In [7], with memory usage and execution time as the criteria, the performance of OMC is compared to the BDD-based model checker SMV for verifying safety properties of the bakery algorithm.

  The performance of the tools was studied against the increase in the domain range of the state variables. Results have shown that the performance of the OMC tool did not remain constant, but was not greatly affected by the increasing the domain ranges. The execution time and the memory usage of SMV, however, increased exponentially with the number of boolean variables required for the binary encoding of each input variable. The experiments were repeated again but with more appropriate encoding of variable ordering in SMV. As a result, the execution time and the memory usage of SMV were reduced to a linear increase, with respect to the number of boolean variables required for the binary encoding of input variables. Though it generally outperformed SMV for the bounded buffer, the OMC tool did not converge (for a whole hour) when attempting to verify certain safety properties. According to [7], the overall experimental results with SMV and OMC suggest that constraint-based model checking can outperform BDD-based model checking for verifying asynchronous concurrent systems with finite but large integer domains.

- In [6], the main objective is the efficient construction of BDD representations for integer arithmetic constraints. An algorithm for generating BDDs for *linear* arithmetic constraints is proposed and shown to produce BDDs with sizes that are linear in the number of variables involved and the number of bits used to encode each variable. However, that result was only valid when all the variables had the same bounds, or different bounds but with all are powers of two. When the variables have different bounds, which are not necessarily power of two, the size of the generated BDDs grows exponentially as the number of variables is increased. Consequently, as noted in [6], this suggests that if the choice of bounds does not compromise the specification then it may be a good idea to choose these bounds as powers of two.

  The proposed algorithm is incorporated into the Action Language Verifier (ALV) tool [10]. Then, using the producer-consumer problem and other examples as a testbed, the performance of ALV is compared to three different implementations of SMV (CMU SMV version 2.5.4.3, Cadence SMV version 08-20-01p2, and NuSMV version 2). The experiments verified both safety and liveness properties and were conducted on SUN ULTRA 10 workstation with

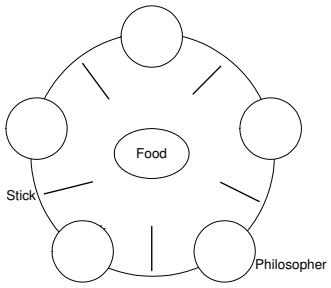

Fig. 6. A view of the dining philosophers problem, N = 5

678 MB of memory, running SunOs 5.7. The results have shown that, the recorded execution times for the three implementations of SMV grow exponentially in terms of the size the involved variables. ALV, on the other hand, has shown linear growth in execution time. Nevertheless, other experiments in [6] have shown that the size of BDDs generated by ALV grows exponentially if the involved variables have different bounds and these bounds are not all power of two.

A.9 Dining Philosophers

Introduction: This is probably the most reported case-study in the literature of *Concurrency Theory*. The problem consists of N philosophers sitting at a table. Between each two philosophers, there is a single stick. In order to eat, a philosopher must pick the sticks on both sides. A problem can arise if each philosopher picks the stick on the right, then waits for the stick on the left. In such case a deadlock occurs and all philosophers will starve.

System characteristics:

Domain:	Computer/Mutex Protocols
Communication:	Shared variable and handshake communication
Degree of coupling:	Loose
Scalability:	Yes, in the number of philosophers
Size/Growth:	N/A
Reported verified properties:	Deadlock-freedom

Selected experiments:

Reference	Technique	Tool	Source code
[16]	Explicit, partial-order reduction	SPIN	N/A
[16]	BDD-based	SMV	N/A
[16]	Inequality Necessary Conditions	INCA	N/A
[14]	MDD-based	[13]	N/A

Reported Results:

- In [16], the problem is modelled in Ada [5] and then automatically translated into the appropriate input language for the three tools SMV [36], SPIN [27] and INCA [3, 17]. The tools are used for detecting deadlock in the model, and are compared according to the growth of resources consumed as the example is scaled up. The reported results can be summarised as follows:

Criteria	Performance, in decreasing order
Growth of consumed resources	INCA, SMV, SPIN+PO, SPIN

- The work reported in [14] presents an MDD-based algorithm, called *saturation*, for generating state spaces. Implemented in SMART [13], the saturation algorithm was applied to the dining philosopher problem, as well as a suite of other examples. Reported results [14] have shown that the saturation algorithm outperforms traditional BDD-based techniques [35] for generating state spaces, in terms of memory and execution time efficiency.

A.10 Queens Problem

Introduction: The N queens problem requires to find a way to position N queens on a $N \times N$ chess board such that they do not attack each other.

System characteristics:

Domain:	Computer/Mutex Protocols
Communication:	Shared variable
Degree of coupling:	Loose
Scalability:	Yes, in the number of queens
Size/Growth:	Exponential
Reported verified properties:	Safety

Selected experiments:

Reference	Technique	Tool	Source code
[14]	MDD-based	SMART	N/A
[21]	MDD-based, Symmetry	SPIN	Online
[21]	Constraint-based	SPIN	Online

Reported Results:

- The work reported in [14] presents an MDD-based algorithm, called *saturation*, for generating state spaces. Implemented in SMART [13], the saturation algorithm was applied to the dining philosopher problem, as well as a suite of other examples. The criteria considered for the experiments were the peak and final number of MDD nodes, as well as the CPU time required for the state-space generation. Compared to other examples in [14], the queens' problem was the most challenging and had the poorest performance of the saturation algorithm. Nonetheless, results showed that even in that case the

saturation algorithm outperformed traditional BDD-based techniques [35], as the algorithm was substantially faster and required less nodes.

– In [21] the n–queens problem is modelled in Promela, and then the SPIN model checker is used to find all the possible solutions. The paper not only compares the use of symmetry reduction in constraint processing and model checking, but shows how model checking with symmetry reduction outperforms model checking for the particular example of the n–queens. Some results on the number of queens, number of solutions found, number of states searched and time and memory used are presented for the symmetric case.

A.11 Readers-Writers Problem

Introduction: The readers-writers problem often rises in data bases systems. There are a number of reader and writers processes that can access the database. Several readers can be actively reading at the same time. However, if one process is writing then no other readers or writers can share access to the database.

System characteristics:

Domain:	Computer/Mutex Protocols
Communication:	Shared variable
Degree of coupling:	Strong
Scalability:	Yes, in the number of readers/writers
Size/Growth:	Polynomial
Reported verified properties:	Safety

Selected experiments:

Reference	Technique	Tool	Source code
[16]	Explicit, partial-order reduction	SPIN	N/A
[16]	BDD-based	SMV	N/A
[16]	Inequality Necessary Conditions	INCA	N/A
[7]	BDD-based	SMV	N/A
[7, 9]	Constraint-based	OMC	N/A

Reported Results:

– In [16], the problem is modelled in Ada [5] and then automatically translated into the appropriate input language for the three tools SMV [36], SPIN [27] and INCA [3, 17]. The tools are used for detecting deadlock in the model, and are compared according to the growth of resources consumed as the example is scaled up. The following table shows the time and memory growth rate for each of the tools as the number of readers (= number of writers) is increased.

SPIN		SPIN+PO		SMV		INCA	
Mem	Time	Mem	Time	Mem	Time	Mem	Time
9.4	10.2	10.9	14.0	0.4	1.5	1.4	1.4

Surprisingly perhaps, SPIN+PO exhibited higher growth rates than SPIN. This, as noted in [16], is mainly due to the *strong* coupling of communications in the readers-writers problem; all processes have to synchronise with a controller task before (and when finished) accessing the database.

– In [7], the main objective is the comparison between constraint-based model checking (which uses arithmetic constraints as a symbolic representations) and traditional BDD-based model checking. The readers-writers problem, among other examples, is modelled and verified for safety-properties using the Omega library model checker (OMC) [32, 41] and the SMV tool. OMC uses Presburger arithmetic (integer arithmetic without multiplication) [8, 9] constraints as its underlying state representation and transition relations, whereas SMV uses BDD for binary encoding of state representation and transition relations. To compare OMC and SMV, 16 different instances of the readers-writers model were generated. Then, the performance of the two tools was measured in terms of the execution time and memory usage required to verify the safety properties considered. Initial results showed that the execution time and the memory usage of SMV increased exponentially as the model is scaled up. The experiments were repeated again but with more appropriate encoding of variable ordering in SMV. As a result, the execution time and the memory usage of SMV were reduced to a linear increase. In both cases, however, the performance of OMC was always better than the SMV tool.

A.12 Sleeping Barber

Introduction: The sleeping barber problem simulates a finite queue of processes waiting to access a single shared resource. The barber shop has one barber, a barber chair, and an n-chair waiting room. When not busy, the barber sleeps in his chair. An arriving customer finding the barber asleep, wakes him up, sets in the barber's chair, and gets a haircut. An arriving customer finding the barber busy takes a seat in the waiting room – if there are no chairs available, then the customer leaves the shop. After finishing a hair cut, the barber serves the next waiting customer – if the waiting room is empty, then the barber goes back to sleep.

System characteristics:

Domain:	Computer/Mutex Protocols
Communication:	Shared variable
Degree of coupling:	Loose
Scalability:	Yes, in the number of chairs
Size/Growth:	N/A
Reported verified properties:	Safety

Selected experiments:

Reference	Technique	Tool	Source code
[7]	BDD-based	SMV	N/A
[7]	Constraint-based	OMC	N/A
[6]	BDD-based	SMV, ALV	[6]

Reported Results:

- In [7], the main objective is the comparison between constraint-based model checking (which uses arithmetic constraints as a symbolic representations) and traditional BDD-based model checking. The sleeping barber problem, among other examples, is modelled and verified for safety-properties using the Omega library model checker (OMC) [32, 41] and the SMV tool. OMC uses Presburger arithmetic (integer arithmetic without multiplication) [8, 9] constraints as its underlying state representation and transition relations, whereas SMV uses BDD for binary encoding of state representation and transition relations. To compare OMC and SMV, 16 different instances of the sleeping barber problem were generated, by restricting the ranges of variables to different values. Then, the performance of the two tools was measured in terms of the execution time and memory usage required to verify the safety properties considered. Initial results showed that the execution time and the memory usage of SMV increased exponentially as the model is scaled up. In this case, the execution time of OMC was better than SMV when the number of chairs is more than 6. The experiments were repeated again but with more appropriate encoding of variable ordering in SMV. As a result, the execution time and the memory usage of SMV were reduced to a linear increase. In this case the execution time of OMC was better than SMV when the number of chairs is more than 2^{14}. In all experiments, however, the memory usage of OMC was always better than SMV.
- In [6], the main objective is the efficient construction of BDD representations for integer arithmetic constraints. An algorithm for generating BDDs for *linear* arithmetic constraints is proposed and shown to produce BDDs with sizes that are linear in the number of variables involved and the number of bits used to encode each variable. However, that result was only valid when all the variables had the same bounds, or different bounds but with all are powers of two. When the variables have different bounds, which are not necessarily power of two, the size of the generated BDDs grows exponentially as the number of variables is increased. Consequently, as noted in [6], this suggests that if the choice of bounds does not compromise the specification then it may be a good idea to choose these bounds as powers of two.

  The proposed algorithm is incorporated into the Action Language Verifier (ALV) tool [10]. Then, using the sleeping barber and other examples as a testbed, the performance of ALV is compared to three different implementations of SMV (CMU SMV version 2.5.4.3, Cadence SMV version 08-20-01p2, and NuSMV version 2). The experiments verified both safety and liveness

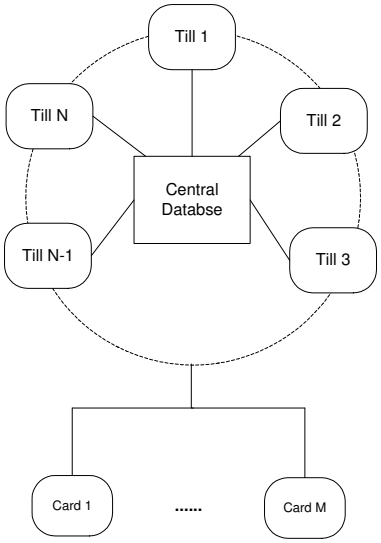

Fig. 7. An overview of the cash machine system

properties and were conducted on SUN ULTRA 10 workstation with 678 MB of memory, running SunOs 5.7. The results have shown that, the recorded execution times for the three implementations of SMV grow exponentially in terms of the size the involved variables. ALV, on the other hand, has shown linear growth in execution time. Nevertheless, other experiments in [6] have shown that the size of BDDs generated by ALV grows exponentially if the involved variables have different bounds and these bounds are not all power of two.

A.13 Cash Machine

Introduction: Illustrated in Figure 7, the cash machine system involves a central database connected to N number of tills, which each can service requests from M number of users. While a till can only service one customer at a time (by taking in a bank card), the cash machine system should provide concurrent access to the central database from two or more tills – the database may not be available all the time. Also, the system should ensure that an initiated transaction eventually runs to a completion, preferably within realistic time constraint. The service from a cash point is provided only when a card is identified as *legal* through the use of a PIN number; *illegal* cards should be kept by the till. Two customers with a shared account should be able to concurrently access that account from two different tills. Finally, the system should be secure and reliable. Thus, it is important to minimize the possibility of the use of stolen cards, and to be robust against hazards such as poor transmission between a till and the central database.

System characteristics:

Domain:	Controllers
Communication:	Shared variable and buffered communication
Degree of coupling:	Medium
Scalability:	Yes, in the number of tills and user cards
Size/Growth:	N/A
Reported verified properties:	Safety and liveness

Selected experiments:

Reference	Technique	Tool	Source code
[43]	Explicit, partial-order reduction	SPIN	N/A

Reported Results:

– In [43], the proposed solution for the problem takes a distributed approach, where the system consists of multiple controllers, one for till. The cash machine system is specified in Promela and verified for LTL-specified liveness and safety properties using SPIN. The verification process is limited to a cash machine system that has only two users and two tills. Also, the withdraw transaction is restricted to a fixed amount. The tool has successfully verified the safety and liveness properties. The table below gives the maximum value recorded over all experiments for the execution time, memory usage, number of transactions, the search depth, and the number of generated states.

Time (min)	Memory (mb)	#Transitions	Search Depth	#States
29.03	491.074	5.79×10^{07}	1974536	3.31×10^{07}

It was not possible, however, to verify time related properties such as "all initiated transactions should be completed within a given time". This is mainly due to the limitation of LTL, which does not provide a mechanism for defining time spans.

A.14 CDMA Library

Introduction: Typically, a wireless communication network comprises three components: the mobile devices (e.g. mobile phones), base stations (which contain the control hardware and software for managing communications), and switching centres (which handle specific features such as location management). The CDMA (code-division multiple access) library is that part of the software at the base stations which sets up and manages calls to and from the mobile devices. The CDMA software must also support continuity of connection as a mobile device changes its location. The CDMA system is important from an industrial point of view as it is widely deployed in wireless networks across the globe. Also, from a verification point of view, the system is challenging due to its large size (typically hundreds of KLOC) and complexity of architecture; the CDMA is embedded in a highly networked environment and can be invoked through multiple interfaces.

System characteristics:

Domain:	Controllers
Communication:	Shared variable and handshake communication
Degree of coupling:	Strong
Scalability:	Yes, in the number of mobiles
Size/Growth:	N/A
Reported verified properties:	Safety, liveness, and deadlock-freedom

Selected experiments:

Reference	Technique	Tool	Source code
[12]	Explicit	VeriSoft	N/A

Reported Results:

– In [12], the emphasis is on model checking the *source code* of concurrent software systems, rather than abstract models specified in some finite-state modeling language. The idea is to perform automatic and systematic testing that derives a program through all possible (concurrent) executions. As an illustration of that approach, testing on Lucent Technologies' CDMA call-processing library was performed using the VeriSoft [24] tool. First, several nondeterministic programs were created to simulate the system's environment; that is, mobile phones and their activities. Then, VeriSoft was used to systematically drive the execution of these programs through all possible behaviour. Consequently, the tool was able to analyse the behaviour of the CDMA library with respect to millions of scenarios. The result was the exposition of several implementation errors that were previously unknown. In general, the VeriSoft tool is capable of deadlocks, livelocks, divergences and assertion violations. However, like all other model checkers, the tool suffers from the state explosion problem. This drawback can be rectified in VeriSoft by limiting the amount of nondeterminism visible to tool. Unfortunately, this comes with a price as hiding nondeterminism can result in errors being missed.

A.15 Cyclic Scheduler

Introduction: The system describe a scheduler for N concurrent processes. The processes are scheduled in a cyclic fashion so that the first process is reactivated after the Nth process has been activated. Also, each process must signal termination before it can be reactivated. The scheduler consists of N *cyclers*. Each cycler, say C_i, maintains communication with one process, say P_i. When C_i starts, it activates P_i. Then, C_i waits for P_i to terminate. When P_i terminates, C_i activates the next cycler and waits to be activated again.

System characteristics:

Domain:	Controllers
Communication:	Handshake communication
Degree of coupling:	Loose
Scalability:	Yes, in the number of cyclers
Size/Growth:	Exponential
Reported verified properties:	Deadlock-freedom

Selected experiments:

Reference	Technique	Tool	Source code
[16]	Explicit, partial-order reduction	SPIN	N/A
[16]	BDD-based	SMV	N/A
[16]	Inequality Necessary Conditions	INCA	N/A

Reported Results:

– In [16], the problem is modelled in Ada [5] and then automatically translated into the appropriate input language for the three tools SMV [36], SPIN [27] and INCA [3, 17]. The tools are used for detecting deadlock in the model, and are compared according to the growth of resources consumed as the example is scaled up. The following table shows the time and memory growth rate for each of the tools as the number of cyclers is increased.

SPIN		SPIN+PO		SMV		INCA	
Mem	Time	Mem	Time	Mem	Time	Mem	Time
12.8	15.7	1.1	1.6	0.3	1.5	1.0	2.4

As indicated by the numbers, SMV showed the slowest growth rate for memory usage, followed by INCA, SPIN+PO, and then SPIN. The story is almost the same for growth rate of execution time, except that SPIN+PO came before INCA this time.

A.16 Gas Station

Introduction: Illustrated in Figure 8 the system simulates a self-service gas station with an operator and a number of pumps and customers. A customer first prepays the operator to get a gas from a specific pump. Once prepaid, the operator activates the pump; this service is done in a FIFO order by passing the customer id to a queue associated with the pump. In turn, a customer can start filling gas from a pump, provided successful verification of id. When the customer is finished, the pump passes the charge back to the operator, who then charges the customer and returns the change, if any.

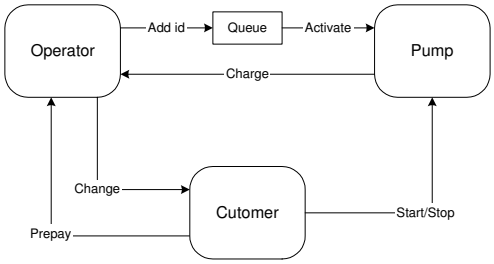

Fig. 8. A layout of the gas station system

System characteristics:

Domain:	Controllers
Communication:	Handshake communication
Degree of coupling:	Strong
Scalability:	Yes, in the number of customers/pumps
Size/Growth:	Exponential
Reported verified properties:	Safety

Selected experiments:

Reference	Technique	Tool	Source code
[38]	Explicit, partial-order reduction	SPIN	N/A
[38]	BDD-based	SMV	N/A

Reported Results:

– In [38], the emphasis is on model checking the *source code* of concurrent software systems, rather than abstract models specified in some finite-state modeling language. The idea is to statically analyse the code, abstract away details, and then automatically generate a model in the input language of a model checker[1]. The generated model can then be checked for temporal properties. The work reported in [38] focused on unit testing Ada programs for safety properties. A unit, e.g. a procedure or a package, is annotated with assume-guarantee assertions (in LTL) that define the behaviour of the environment. These assertions are then used to synthesis Ada implementations of testing stubs and drivers, thus resulting in a complete program. Using abstract interpretation [18] and partial evaluation [29] techniques, the complete program is then abstracted and simplified before it is finally submitted to the INCA tool to generate corresponding models in the input language of SMV and SPIN. The tools are then used for verifying safety properties of the considered Ada unit. This process has been applied to three versions

[1] This is the alternative approach to the technique used in [12], Section A.14, where a property is verified by automatic testing that derives the program through all possible executions

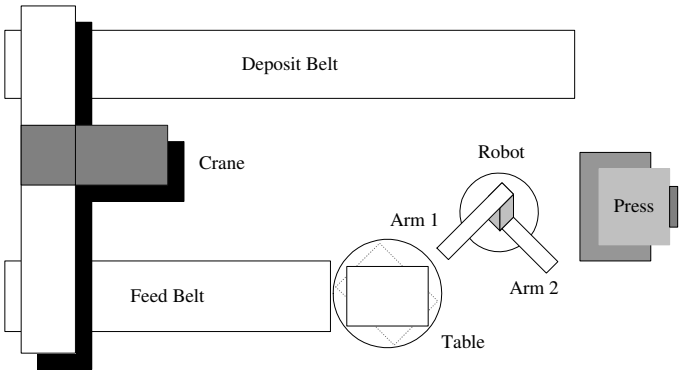

Fig. 9. A top view of the production cell

of the gas station system: with two, three, and four customers but only one pump. For the case of two customers, SPIN showed better performance in terms of execution time and memory usage. For the cases with three and four customers, however, SMV outperformed SPIN in both preceding criteria.[2]

A.17 Production Cell

Introduction: Depicted in Figure 9, the Production Cell system comprises six machines, working concurrently: *Feed Belt*, *Elevating Table*, *Robot*, *Press*, *Deposit Belt*, and *Crane*. The sequence of production starts by the feed belt transporting a metal plate to the table. The table then elevates and rotates so that the robot can pick up the plate. The robot picks up the plate with its first arm, then turns anticlockwise and feeds the metal plate into the press. The press forges the plate and returns to bottom position in order to unload. The robot picks up the plate from the press with its second arm, then rotates further to unload the plate on the deposit belt. The deposit belt transports the plate to the travelling crane. The crane picks up the metal plate from the deposit belt, moves to the feed belt and then unloads the metal plate; a new cycle begins. This sequence is further complicated by the fact that the robot can go back to the table and pick up a metal plate while the press is forging another one.

[2] Another result in [38], which is not relative to the benchmark is that the use of synthesized environments (as opposed to universal environments which are capable of invoking any sequence of operations in a unit's interface) enable faster model-checking.

System characteristics:

Domain:	Controllers
Communication:	Handshake communication
Degree of coupling:	Loose
Scalability:	–
Size/Growth:	10^{12}
Reported verified properties:	Safety and liveness

Selected experiments:

Reference	Technique	Tool	Source code
[39]	Explicit, partial-order reduction	SPIN	N/A

Reported Results:

– In [39], the Production Cell is specified in Promela and verified using SPIN version 2.9.5; the partial-ordering reduction algorithm was enabled, but no memory compression options were used. No timing properties were specified or verified. The liveness and safety properties were formalised in linear-time logic (LTL) and verified using full state-space search. The model was designed so that up to 8 blanks can stay in the Production Cell concurrently. However, due to memory constraints, the liveness and safety properties could be verified for only up to 3 blanks present in the system; tests were performed on a Sun Ultra Enterprise 3000, with 4 250MHz, 4MB cache UltraSPARC CPUs, and 1 GB RAM. Based on the results reported in [39], the following table lists the *verification time* (in seconds), *memory* used (in mega bytes), and *number of computed transitions* (in millions) for the liveness properties. For the safety properties, the table lists the average, the minimum, and the maximum reported value of the preceding three parameters.

	Liveness	Safety Properties		
	Properties	Average	Minimum	Maximum
Time(s)	56	22.5	13	54
Mem(mb)	80	62.5	54	93
Transition(m)	1.4	0.515	0.3	1.3

A.18 Spacecraft Controller

Introduction: NASA's Remote Agent (RA) [40] is an "artificial intelligence"-based space-craft control system architecture which comprises several software modules. The RA executive module, illustrated in Figure 10, is the one designed to support safe execution of software controlled tasks on board the spacecraft. Typically, in order to execute correctly, a task would require number of properties to hold; otherwise, the task execution must be interrupted. For example, a task responsible on running and surveying a camera would require that camera to be turned on throughout the execution. Thus, the RA executive system maintains a *database* of all the properties describing the current

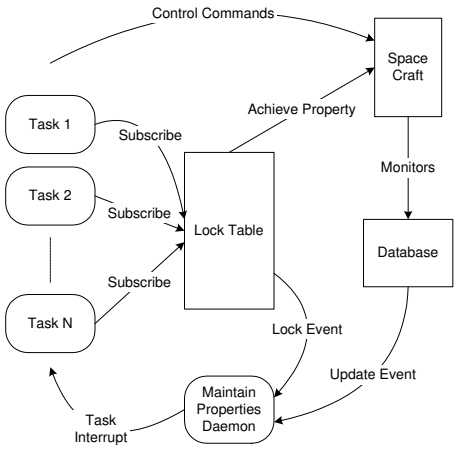

Fig. 10. An overview of the remote agent executive

state of the spacecraft; the properties are saved as pairs of variable names and their corresponding values. To ensure coherence, tasks use *locks* to prevent other tasks with incompatible properties from executing concurrently. For example, the camera-surveying task would lock the property $(camera, ON)$ so that no other tasks, with $camera = OFF$ request, can execute concurrently. The state of the database and the lock table is monitored by a *maintain properties daemon*. Whenever the database of the lock table is changed, say as a result of an update operation or requiring/releasing a lock, the daemon is signaled so that it can examine the renewed system state. If there is any inconsistency between the database and the lock table, the daemon suspends the involved tasks and takes appropriate actions. The RA executive system is implemented in LISP, hence communication between processes is via shared variables.

System characteristics:

Domain:	Controllers
Communication:	Shared variable
Degree of coupling:	Medium
Scalability:	Yes, in the number of tasks and properties
Size/Growth:	N/A
Reported verified properties:	Safety

Selected experiments:

Reference	Technique	Tool	Source code
[26]	Explicit, partial-order reduction	SPIN	N/A

Reported Results:

– In [26], the RA executive program is translated from Lisp into Promela, and then verified using SPIN. The Promela model is restricted to three tasks, including the daemon. Also, the model is restricted to two property names; hence, the size of the database (and the lock table) is also equal to two. The *Environment* process, which runs in parallel with the tasks and the daemon, is then used to introduce violations into the Promela model. Basically the environment always assigns the value 0 to one of the two properties. This may result in an inconsistency between the database and the lock table if a lock has already been created for that property with a value different from 0. The model is then verified for two safety properties, specified as assertions and LTL formulae. Both of the safety properties turned out to be not satisfied by the model. In total, five errors were discovered. One of these five errors was just a minor efficiency problem, some code executed twice instead of once. The remaining four errors, however, were due to processes interleaving in ways not foreseen by the RA programmer. The RA code, and Promela model, and verified again; successfully this time. The following table shows the maximum values reported for the number of states explored, the memory consumption, and execution time, for both the erroneous and the corrected model.

	#States	Memory (mb)	Time (s)
Erroneous Model	49038	3.708	5.4
Corrected Model	222840	7.088	21.2

A.19 TCAS II

Introduction: TCAS is short for Traffic Alert and Collision Avoidance System, an airborne system required on most commercial aircraft. A TCAS-equipped aircraft is surrounded by a protected volume of airspace and is capable of identifying the location and tracking the progress of other aircraft equipped with beacon transponder. When an aircraft breaks in the protected airspace of another flight, the TCAS system generates warnings to the pilot, may be even suggest escape manoeuvres to avoid collision. Currently, there are three versions of the TCAS system: I, II, and III. The specification of such systems is rather large and complex. For example, the system requirements specification [1] of TCAS II is a 400 page document, written in Requirements State Machine Language (RSML) [33].

System characteristics:

Domain:	Controllers
Communication:	N/A
Degree of coupling:	N/A
Scalability:	–
Size/Growth:	10^{65}
Reported verified properties:	Safety

Selected experiments:

Reference	Technique	Tool	Source code
[2]	BDD-based	SMV	N/A

Reported Results:

– In [2], the emphasis is on evaluating symbolic model checking of state-machine based specifications. The TCAS II is used as a test bed, and verified using SMV. As the TCAS II system is quite large, only a portion of the system is used. That is the "Own-Aircraft", which occupies 30% of the overall specification and is responsible on generating the escape manoeuvres to the pilot. The RSML specification of Own-Aircraft is (manually) translated into the input language of SMV, and then verified for a number of safety properties. The size of the state space reached (approximately) 1.4×10^{65}, with at least 9.6×10^{56} reachable states. The following table shows the maximum values reported, over all experiments, for the number of states explored, the memory consumption, and execution time

# BBD nodes	Memory (mb)	Time (s)
717275	16.4	387

A.20 Cache Coherence for Distributed File System

Introduction: To increases performance and availability in a distributed file system, clients are allowed to store copies of the server-files locally in their caches. As these different copies of the files could be updated by the clients, a problem of inconsistency arises. In that context, a file is called *valid* if it is the most recent copy in the system. The goal of a cache coherence protocol, therefore, is to ensure that a client performs work only on valid files. Typically, there are two techniques to achieve this, validation-based and invalidation-based. In validation-based protocols, the clients ask the server whether their local copies are valid. In invalidation-based protocols, it is the responsibility of the server to tell the clients once their local copies are no longer valid. In all cases, a cache coherence protocol must ensure that the *invariance* "If a client believes that a cached file is valid then the authorized server believes that the client's copy is valid." hold. For a simple distributed file system consisting of one server and several clients, Figure 11 illustrates the typical communicated messages in a cache coherence protocol.

System characteristics:

Domain:	Distributed Algorithms
Communication:	Handshake and buffered communication
Degree of coupling:	Medium
Scalability:	Yes, in the number of servers, clients, and files
Size/Growth:	Exponential
Reported verified properties:	Safety

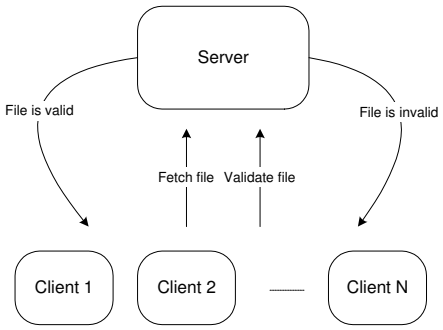

Fig. 11. Overview of communications in cache coherence protocol of a distributed file system

Selected experiments:

Reference	Technique	Tool	Source code
[44]	BDD-based	SMV	[44]

Reported Results:

– In [44], the emphasis is on model checking finite state machine abstractions of software systems. Three cache coherence protocols for distributed file systems are considered and verified using SMV. First, the C code of the protocols is (manually) translated into abstract models in the language of SMV; the abstract models are restricted to two clients, one server, and one file. The protocols are then verified for a CTL-specified correctness property. The maximum reported execution time in all experiments is lees than a second. Also, the maximum reported number of reachable states is 43684.

A.21 Leader Election Protocol

Introduction: Depicted in Figure 12, the leader election protocol allows a ring of N processes to elect a leader through sending messages around the ring. Each process randomly chooses a number *id*, from a restricted set of values and sends it around the ring. If there is a unique *id*, then the process with the maximum unique *id* is elected as the leader. Otherwise, the election step is repeated.

System characteristics:

Domain:	Distributed Algorithms
Communication:	Handshake and buffered communication
Degree of coupling:	Loose
Scalability:	Yes, in the number of processes
Size/Growth:	Polynomial
Reported verified properties:	Liveness

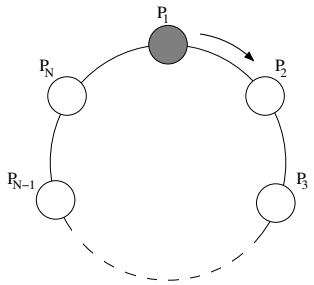

Fig. 12. A top view of the leader election protocol.

Selected experiments:

Reference	Technique	Tool	Source code
[27]	Explicit, partial-order reduction	SPIN	N/A
[34]	Directed Model Checking	HSF–SPIN	N/A

Reported Results:

– In [27], two versions of protocol are specified in Promela and verified using SPIN, with 5 as the total number of processes and partial-order reduction enabled. The first version of the protocol requires that all processes participate in the election right from the beginning. In the second version, a process can decide to participate in the election at a later point of the execution. Two properties were verified for each version, a safety property and a liveness property. In all experiments the number of reachable states was around 200.

– In [34], the emphasis is on studying the results of combining directed model checking with partial-order reduction. The protocol is specified in Promela, and verified for safety properties using HSF–SPIN. Results show that combining directed–search strategies with partial-order reduction can lead to significant improvements in terms of state–space size and the run time. However, for the leader election protocol, the solution quality was also lost and the tool was not able to verify the required safety property.

References

1. Federal Aviation Administration. TCAS II collision avoidance system (CAS) system requirements specification, change 6.00. Techn. rep. U.S. Dept. Transportation, 1993.
2. R.J. Anderson et al. Model checking large software specifications. In *SIGSOFT '96*, pp. 156–166. ACM Press, 1996.
3. G.S. Avrunin et al. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE TSE*, 17(11):1204–1222, 1991.

4. Tutorial On Mechanized Formal Methods. Available at `http://www.csl.sri.-com/users/rushby/abstracts/fm-tut`.
5. J. Barnes. *Programming in Ada 95*. Addison-Wesley, 1998.
6. C. Bartzis and T. Bultan. Construction of efficient BDDs for bounded arithmetic constraints. In *TACAS '03*, vol. 2619 of *LNCS*, pp. 394–408, 2003.
7. T. Bultan. BDD vs. constraint-based model checking: An experimental evaluation for asynchronous concurrent systems. In *TACAS '00*, vol. 1785 of *LNCS*, pp. 441–456, 2000.
8. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *CAV '97*, vol. 1254 of *LNCS*, pp. 400–411, 1997.
9. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM TOPLAS*, 21(4):747–789, 1999.
10. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '01)*, p. 382, 2001.
11. S. Chaki, S.K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *POPL '02*, pp. 45–57. ACM Press, 2002.
12. S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: An industrial case study. In *ICSE '02*, pp. 431–441. ACM Press, 2002.
13. G. Ciardo, R.L. Jones III, R.M. Marmorstein, A.S. Miner, and R. Siminiceanu. Logical and stochastic modeling with smart. In *TOOLS '03*, vol. 2794 of *LNCS*, pp. 78–97, 2003.
14. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *TACAS '01*, vol. 2031 of *LNCS*, pp. 328–342, 2001.
15. R. Cleaveland, P.M. Lewis, S.A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In *CAV '96*, vol. 1102 of *LNCS*, pp. 398–401, 1996.
16. J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE TSE*, 22(3):161–180, 1996.
17. J.C. Corbett and G.S. Avrunin. Using integer programming to verify general safety and liveness properties. *FMSD*, 6:97–123, 1995.
18. P. Cousot. Program analysis: the abstract interpretation perspective. *ACM CSUR*, 28(4es):165, 1996.
19. Customer Support Consortium (CSC) and Desktop Management Task Force (DMTF). Service incident standard (sis) specification, version 1.1. Techn. rep. Available at `http://www.dmtf.org`.
20. D.L. Dill. The Murφ verification system. In *CAV '96*, vol. 1102 of *LNCS*, pp. 390–393, 1996.
21. A. Donaldson, A. Miller, and M. Calder. Comparing the use of symmetry in constraint processing and model checking. In *Symmetry and Constraint Satisfaction Problems*, pp. 18–25, 2004.
22. Y. Dong et al. Fighting livelock in the i-protocol: A comparative study of verification tools. In *TACAS '99*, vol. 1579 of *LNCS*, pp. 74–88, 1999.
23. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with hsf-spin. In *SPIN '01*, vol. 2057 of *LNCS*, pp. 57–79, 2001.
24. P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97*, pp. 174–186, 1997.

25. R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In *CAV '96*, vol. 1102 of *LNCS*, pp. 423–427, 1996.
26. K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space craft controller using SPIN. *IEEE TSE*, 27(8), 2001.
27. G.J. Holzmann. The model checker SPIN. *IEEE TSE*, 23(5), 1997.
28. G.J. Holzmann. The engineering of a model checker: The Gnu i-protocol case study revisited. In *SPIN '99*, vol. 1680 of *LNCS*, pp. 232–244, 1999.
29. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall International, 1993.
30. M. Kamel and S. Leue. Validation of a remote object invocation and object migration in CORBA GIOP using Promela/SPIN. In *SPIN '98*, Paris, France, 1998.
31. M. Kamel and S. Leue. VIP: A visual editor and compiler for v-Promela. In *TACAS '00*, vol. 1785 of *LNCS*, pp. 471–486, 2000.
32. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide. Techn. Rep. CS-TR-3445, U. Maryland, 1995.
33. N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE TSE*, 20(9):684–707, 1994.
34. A. Lluch-Lafuente, S. Edelkamp, and S. Leue. Partial order reduction in directed model checking. In *SPIN '02*, vol. 2318 of *LNCS*, pp. 112–127, 2002.
35. K.L. McMillan. *Symbolic Model Checking: An Approach to the State-explosion Problem.* PhD thesis, Carnegie-Mellon U., 1992.
36. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
37. P. Merino and J.M. Troya. Modeling and verification of the ITU-T multipoint communication service with SPIN. In *SPIN '96*, Rutgers U., New Jersey, 1996.
38. C.S. Pasareanu, M.B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *SPIN '99*, vol. 1680 of *LNCS*, pp. 168–183, 1999.
39. D.O. Paun, M. Chechik, and B. Biechelle. Production cell revisited. In *SPIN '98*, Paris, France, 1998.
40. B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan execution for autonomous spacecrafts. In *International Joint Conference on Artificial Intelligence*, 1997.
41. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pp. 4–13. ACM Press, 1991.
42. Y.S. Ramakrishna and S.A. Smolka. Partial-order reduction in the weak modal mucalculus. In *CONCUR '97*, vol. 1243 of *LNCS*, pp. 5–24, 1997.
43. J. Tang and W. Zhou. Cash point with Promela/SPIN. Techn. Rep. CSRG-407, U. Toronto, 2000.
44. J.M. Wing and M. Vaziri. Model checking software systems: A case study. In *SIGSOFT '95*, pp. 128–139. ACM Press, 1995.