A State Space Distribution Policy based on Abstract Interpretation [1]

Simona Orzan Jaco van de Pol Miguel Valero Espada [2]

SEN-2, CWI. Amsterdam, The Netherlands

Abstract

We aim at improving the performance of distributed algorithms for model checking and state space reduction. To this end, we introduce a new distribution policy of states over workers. This policy reduces the number of transitions between states located at different workers. This in turn is expected to reduce the communication costs of the distributed algorithms.

The main idea is to use Abstract Interpretation techniques to compute a small approximation of the state space, starting from some high level description of the system. Based on this approximation, the connectivity of concrete states is predicted. This information is used to distribute states with expected connectivity to the same worker. Experiments show a considerable reduction of cross transitions, at the expense of a modest unbalance of nodes per worker.

1 Introduction

The behaviour of a reactive system can be represented as a *state space* or labelled transition system (LTS). Nodes of the LTS correspond to states, edges to transitions between states, and labels on edges correspond to events. In *enumerative model checking*, an LTS is generated from a system specification, and desired properties are checked by a model checking algorithm.

We here assume a *full generation* approach, where the generation phase and model checking phase are separated. This can be contrasted to *on-the-fly* model checking, where generation and model checking are interleaved. The latter is convenient when a counter example is found before the whole LTS is generated. However, if many properties of the same model must be checked,

[1] Partially supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009.

[2] Corresponding author. Email: miguel@cwi.nl

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and no more counter examples are expected, full generation is a good alternative. In particular, the state space is generated only once, and can be subsequently reduced according to some property preserving equivalence, such as (branching) bisimulation.

Examples of distributed algorithms for enumerative verification are state space generation [7], [5], state space reduction modulo bisimulations [4], detection of strongly connected components [18] and model checking [2], [1]. They operate on very large state spaces that are divided into shares for every node in the network (subsequently called "worker"). Each worker stores the outgoing transitions of all the states (source nodes) that it owns.

The performance of distributed algorithms depends on the *computation* load per worker. But, unlike sequential or parallel shared memory algorithms, distributed algorithms run on a network, so the performance also depends highly on the *communication* costs, that is on the total number of messages passed between the workers.

Most algorithms above are based on reachability procedures, in which information is transfered between states that are connected by transitions. So, in order to reduce the communication costs, we must minimize the number of cross transitions, i.e., edges between states on different workers. At the same time, the number of nodes per worker must be balanced, in order to maximally exploit the parallelism possibilities.

The problem we address in this paper is to find distributions that reduce the number of cross transitions, without compromising the balance of nodes per worker too much. Note that graph partitioning is inherently hard, so even though the state space is completely generated, it is infeasible to directly compute the optimal distribution.

As a solution, we propose to compute a small approximation of the state space by an abstract interpretation [8,15] of the system specification. The connectivity of abstract states is used to predict connectivity between the corresponding concrete states. In particular, the abstraction function is used, in order to assign the states to a worker.

As a feasibility study, we implemented this distribution policy. We measured the number of nodes per worker and the number of cross transitions. We compared the results with the random distribution on a number of large state spaces (millions of states). The improvement on the cross transitions are impressive, while the penalty of unbalanced number of nodes per worker remains modest. So, even though our ideas have not yet been inserted in the actual distributed algorithms, the results are very encouraging.

Related work. Many papers presenting distributed model checking algorithms and other distributed applications on large graphs, acknowledge the importance of a good graph partitioning method. Graph (bi-)partitioning is an NP-hard optimization problem [9], therefore near-optimal solutions are computed using specialized heuristics or general stochastic procedures (like

simulated annealing, genetic algorithms). Our approach is an heuristics especially designed for graphs representing behaviours (LTSs).

The effects of the partitioning scheme on the performance of distributed state space generation for timed Petri nets were documented in [24]. Not surprisingly, it was found that states that are clustered together should be sent to the same processor. But no procedure for finding "good" partitioning functions was suggested.

Our approach has similarities to the one presented in [7], where a partitioning function is defined taking into account the structure of the original (Petri net) specification. This gives a small number of cross transitions at the risk of bad balance. However, our experiments show that the abstract interpretation distribution algorithm produces a much lower proportion of cross transitions.

The partitioning problem shows up in symbolic model checking as well. In [13], large BDDs are dynamically (re)sliced while computing the reachable states. The main goal there is to keep the memory requirements balanced throughout the computation.

2 Preliminaries

2.1 Distributed Enumerative Model Checking

All our distributed tools perform on an iterative basis, alternating computation and communication phases. The good performance of computation phases relies on the balanced workload, that is on a balanced assignment of states to workers. The communication phases consist mostly of exchanging information about neighbour states. Therefore the communication performance depends on the amount of transitions that cross worker boundaries (*cross transitions*).

Our algorithm for branching bisimulation reduction [4] walks, in every iteration, through whole subgraphs of silent steps. This operation is especially expensive if these subgraphs are scattered on more workers, as it is usually the case in random state space distributions.

The distributed tool that finds cycles [18] has a phase where all local (i.e., internal to a worker) strongly connected components are collapsed. In a "good" distribution with few cross transitions, it is likely that this reduces the state space considerably.

Definitions. Let Act be a fixed set of labels, representing actions. Act contains a special action τ that stands for an internal (silent) step. A *Labelled Transition System* (LTS) is a triple (S, T, s_0) consisting of a set of states S, a set of transitions $T \subseteq S \times \mathsf{Act} \times S$ and an initial state $s_0 \in S$. LTSs describe in a precise manner the behaviour of protocols or systems under verification and they are the format on which automatic model checking tools perform.

A *distribution* of an LTS to W network nodes (workers) is a partition of its set of states into W pairwise disjoint subsets: $S = S_1 \cup \cdots \cup S_{\mathsf{W}}$. We denote by T_{ij} the set of transitions with the source state assigned to worker i and the

destination state assigned to worker j. Then the elements of the sets T_{ii} are *internal transitions* and those of the sets T_{ij} $(i \neq j)$ are *cross transitions*.
For the goals discussed above, an *efficient* distribution of an LTS should have:

- *balance*, i.e., more-or-less the same number of states on each worker. To measure this, we introduce the notion of *worst case balance* (WCb), as being the difference (in %) between the biggest load (number of states) assigned to a worker and the average load. Since the sets of states assigned to different workers are disjoint, the average load is the total number of states divided by the number of workers.

$$\mathsf{avgload} = \frac{\#S}{\mathsf{W}}, \qquad \mathsf{WCb} = \max_{1 \leq i \leq \mathsf{W}} (100 * \frac{\#S_i - \mathsf{avgload}}{\mathsf{avgload}})$$

- *a minimal number of cross transitions*, or, in other words, as many internal transitions as possible. We express this factor by the ratio (number of internal transitions)/(total number of transitions):

$$\text{internal transitions ratio} \qquad \mathsf{IntT} = \frac{\sum_{1 \leq i \leq \mathsf{W}} \#T_{ii}}{\#T}$$

In Figure 1 a simple state space is shown and two possible distributions of it on a network of two workers [3]. Both are perfectly balanced but the first one is almost a worst-case scenario for distributed algorithms, while the second one is extremely efficient, due to the small number of cross transitions and the localization of cycles.

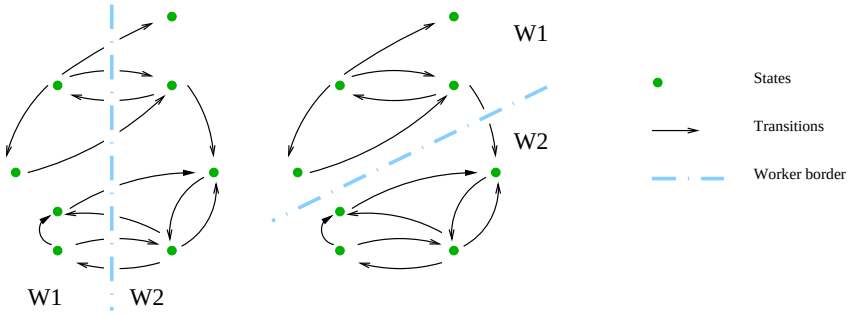

Fig. 1. A balanced but bad state space partition (left) $\mathsf{IntT} = 38.4\%$ and a good partition (right) $\mathsf{IntT} = 92.3\%$

2.2 Modal Transition Systems

To model abstractions we use a structure that allows to represent approximations of the concrete system in a suitable way. A *Modal Labelled Transition*

[3] We do not include action labels in the figures, because they are not necessary for the understanding of the paper.

System (*Modal*-LTS) is a graph in which edges have two modalities *may* and *must* which denote the possible and necessary steps in the refinements. If in a *Modal*-LTS, there is a *must* transition between two states then the transition is required in all the refinements. If there is a *may* transition then the transition is allowed in the refinements. This concept was introduced by Larsen and Thomsen [16]. The formal definition trivially extends the definition of LTSs (given in the previous section) by having the two modalities and by requiring that every *must* transition is also *may* one.

From a concrete system described by an LTS we can generate an abstraction of it by relating concrete states with abstract ones, using a mapping function. The *may*-transitions correspond to an over-approximation of the original and the *must* ones to an under-approximation. Therefore, the abstract approximations capture in some sense the shape of the original system and the relations among the states. The next figure presents an example of abstraction of the system presented in Figure 1.

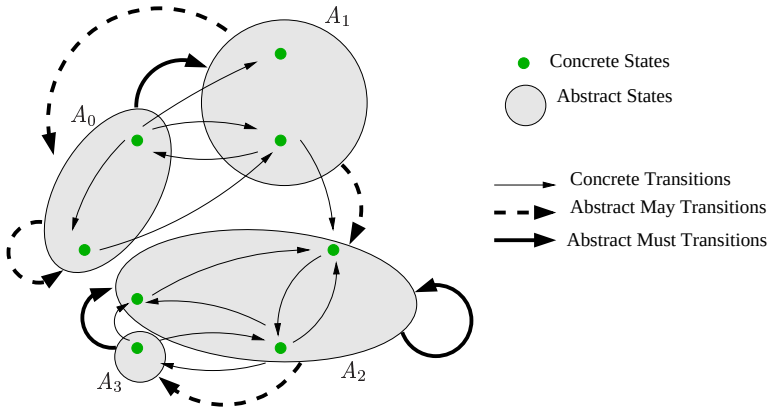

Fig. 2. Example of Abstraction

All concrete states mapped to the abstract state A_0 have a transition to a state related to the abstract state A_1, therefore there is a *must* transition between A_0 and A_1. Furthermore, we see that one concrete state (but not all) related to A_1 has a transition to a concrete state related to A_2, so there is a *may* transition between them but not a *must* one. In the figure, *may* abstract transitions are marked by the dashed arrows. Whenever there is a *must* transition, there is also a *may* one, but we do not draw these arrows.

2.3 Abstract Interpretation

The theory of abstract interpretation [8,15] denotes a classical framework for program analysis. The idea is to extract, from high level descriptions of systems, approximations by eliminating uninteresting information.

A formal specification is interpreted over an abstract domain which is typically smaller than the concrete domain. A simple example of the technique is the so-called "rule of signs" used to determine the sign of arithmetic expres-

sions by performing the computation only over the signs of the operators, i.e., the expression $-5 * 10$ is abstracted to $neg * pos$ which preserves the sign of the result. We are going to use this method to generate predictions about the concrete state spaces and distribute efficiently the states among the workers.

To obtain an abstract approximation from a given specification, we interpret the concrete specification over an abstract domain. This results in an abstract transition system. In order to apply the method, it is needed to provide the abstract domain, its relation with the concrete domain and a safe definition of the functions, that appear in the concrete specification, over the abstract domain.

Finding good abstractions, i.e. abstractions that preserve essential behavioural properties of the concrete specification, requires much effort and a very good knowledge of the system analyzed. However, for our goal of using abstractions as an indication of the state space structure, any abstraction is useful! In particular, we will use a few that are automatically generated.

There are several techniques to automatically abstract systems preserving interesting information. The simplest one is called *hiding*: first, the value of some variables of the system is considered as unknown, subsequently, extra non-determinism is added to the system when there are predicates over the abstracted variables. We consider a simple example of a recursive process P with two variables x and y of type natural:

$$\begin{aligned} P(x,y) &: (x > 3 \wedge y < 5) \rightarrow a.P(x+1, y-1) \\ &: (x < 5 \vee y > 3) \rightarrow b.P(x-1, y+1) \end{aligned}$$

The pseudo-code expresses that if one condition is satisfied for a given value of the variables then an action (a or b) is executed and the values are updated. Let us consider that the variable y is *hidden*, then its value is replaced by ?. Moreover, the functions that operate over y return *unknown* (?):

$$\begin{aligned} P(x) &: (x > 3 \wedge ?) \rightarrow a.P(x+1) \\ &: (x < 5 \vee ?) \rightarrow b.P(x-1) \end{aligned}$$

If at some point of the execution x has the value 4, then, the first condition will be $(true \wedge ?)$ whose evaluation returns ?, therefore, the abstract system will have a transition $P(4) \xrightarrow{a}_{may} P(5)$. The second condition will be $(true \vee ?)$ which always evaluates to $true$, therefore, there will be a transition $P(4) \xrightarrow{b}_{must} P(3)$.

Obviously, variable *hiding* is not the unique abstraction technique that may be applied. Our method, explained in detail in [23], is more general allowing the use of any abstraction approach.

3 Abstraction Guided Distribution

As we have presented above, computing an abstract approximation of a system gives a prediction about the shape of the state graph and the connections among the states. We use the abstract transition system to optimize the number of *cross transitions*. However, the abstract graph does not contain information about the number of concrete states that are related to every abstract state, therefore the minimization of the *cross transitions* may have the drawback of the loss of balance of the system. To attack this problem, we first distribute the abstract states over C classes. The partitioning algorithm is:

> **1** - Generate an abstract approximation from the original specification.
> **2** - Split the abstract transition system in C classes, optimizing the number of *abstract cross transitions* between classes.
> **3** - Generate the concrete transition system.
> **4** - Redistribute the classes among the workers balancing the load of the system.

After the algorithm, one can apply the desired model checking tool, such as bisimulation reduction, cycle elimination, etc. In the next section, we will explain a basic implementation of the crucial steps of the algorithm, now we discuss some general considerations.

Step **1** requires to select the suitable parts of the concrete specification to be abstracted and to provide the abstraction function H to map concrete states to abstract states. Even the simplest technique of abstract interpretation *variable hiding* requires some "intelligence" in order to select the variables to hide. It is, therefore, important to develop an algorithm to automatically select interesting abstractions. The generation of the abstract transition system is done using existing tools [3]. The output of this operation is a *Modal*-LTS.

To split the abstract graph (step **2**), optimizing the number of *cross transitions* we use the following principles:

- If there is a *must* transition between two abstract states, $A \rightarrow_{must} B$, then there are transitions from all concrete states related with A to some state related to B.
- If there is not any *may* transition between two abstract states, $A \not\rightarrow_{may} B$, then there are no transitions from any of the concrete states related with A to the ones related to B.

The idea is to assign to the same class the concrete states (as many of them as possible) that correspond to abstract states that are connected by *must* transitions. Moreover, to assign to different classes the concrete states that correspond to abstract states that are not connected by *may* transitions. Furthermore, abstract loops possibly correspond to concrete loops hence it may be convenient to assign them to a single class. Considering these ideas, the example on Figure 2 will be split in two classes as follows: $\{A_0, A_1\}$ and $\{A_2, A_3\}$.

Typically, the abstract graphs are small. Therefore algorithms with high complexity can be safely used. In this step, we generate a mapping G between abstract

states and classes. The number of classes C is important because it is what makes the method flexible to equilibrate the balance of the system. On the one hand, if we select a lot of classes then there will be few abstract states associated to each of them, therefore the final result will be well balanced but the number of *cross transitions* will not be minimized. On the other hand, if we select few classes the number of *cross transitions* will be low but the system may be unbalanced. In any case, C should be greater than the number of workers and smaller than the number of abstract states.

The generation of the concrete transition system (step **3**) is done using the existing distributed state space generator tool. For every explored concrete state s, we compute the class to which it belongs by applying $G \circ H$ to s. The transitions whose source state belongs to a determined class c are stored together in the same file. Therefore the output of generation will be a set of C files, one for every class.

While generating the state space we store the number of states that belong to each class. This number is first computed locally, i.e., every worker stores the number of states explored by itself. At the end of the computation the results are gathered. Subsequently (step **4**), we can redistribute the classes over the workers, in order to equilibrate the balance during the application of the different distributed algorithms.

If we apply the method to on-the-fly model checking, we cannot redistribute in the balance of the system in the step 4, because we do not have the full state space. The balance of the system can be guaranteed using other approaches, for instance, dynamic redistribution of the load.

4 Basic Implementation

For a feasibility study of our distribution policy, we implemented the steps above with minimal effort. We added code to our existing tools, to measure the effect on balance and cross transitions, but the implementation leaves still space for improvements. The results are compared with an existing and widely used implementation.

Abstraction (step **1**): We start the process with a specification written in μCRL, which is a language that combines process algebra and abstract data types (see [11]). In [23], it is discussed how to extract modal abstractions from μCRL specifications and in [22] the tool kit that implements the abstraction techniques is described.

The selection of the initial abstraction is a crucial point in order to obtain satisfactory results. In our preliminary experiments, we hide the main part of the systems. We let unhidden some variables that control the flow of the system. This technique has the advantage that it can be completely automatized since the control flow variables can be easily detected. Typically, the control flow represents accurately the shape of the final graph. This technique generates rather small abstractions which mainly contain *may* transitions. We believe that by increasing the size of the abstractions the results can be better. Moreover, we think that the abstractions have to contain not only information about the control flow but also about data.

Splitting (step **2**): To assign abstract states to the classes, we traverse the abstract graph using a breadth first search algorithm giving priority to *must* transitions. Then, the first A/C abstract states are assigned to the first class, the next A/C

states to the second class and so on... If $A = k * C$ then all classes will receive k abstract states, otherwise some ones will receive one extra abstract state. The algorithm splits the abstract state space in *layers* keeping together abstract states that are connected without performing any kind of calculation.
Redistribution (step **4**): After gathering the results of the concrete computation, the redistribution of the C classes among W workers is done using a "greedy heuristics". First, we order the classes by number of concrete states, then we assign one class to each worker. Subsequently, for each worker, while the number of states assigned to the worker is smaller than the average, we assign new classes to it.

The choice of the parameter W depends on the availability of resources of the system, and C on the size of the abstract graph A, as we have pointed in the previous section.

The three algorithms are very simple and the implementation is straightforward. In the next section we present the results of this basic implementation on different examples.

5 Experimental Results

We have applied our method to a set of different applications composed by distributed algorithms, communication protocols and industrial case studies. The experiment consists of the distributed generation of the full state space of the systems and the computation of some measurements that will determine the performance of the rest of distributed model checking tools. The next table presents the result of the experiments, the explanations come below:

	Size		Parameters			Random dist.		Abstract dist.	
system	states	transitions	W	A	C	WCb	IntT	WCb	IntT
JavaSpaces	1,464,665	5,660,242	8	352	32	22.66%	**12.55%**	21.52%	**49.71%**
Lift	2,165,446	6,289,045	4	960	16	1.12%	**27.91%**	21.54%	**78.50%**
IEEE-1394	371,804	641,565	8	742	32	0.42%	**12.54%**	43.18%	**69.68%**
Leader	2,416,632	16,605,592	8	2916	64	1.92%	**12.51%**	23.12%	**81.09%**
Splice	18,140,058	186,085,954	4	40	16	0.93%	**25.00%**	123.26%	**91.89%**
CCP	8,079,312	60,887,345	8	3740	64	1.54%	**54.44%**	8.92%	**84.13%**
Bad Client	13,859,510	71,817,848	8	540	32	1.16%	**28.63%**	23.48%	**26.87%**

Fig. 3. Benchmark for abstraction guided distribution

In every state space generation, we, basically, compute for the random distribution and for the abstraction guided distribution, the worst case balance (WCb) and the ratio of internal transitions (IntT), as defined in Section 2.1. Remark that we want to maximize the first figure and minimize the second one. The table shows also: the number of workers of the cluster (W), the number of abstract states (A) and the number of classes (C).

We briefly describe the systems analyzed in order to give an impression of the heterogeneity of the experiments: *JavaSpaces* is a shared data space architecture used to facilitate the coordination and communication of distributed applications. The specification was described in [21], it describes a fault-tolerant algorithm that

performs the parallel summation of a set of numbers. *Lift* models an industrial system to control a distributed system with multiple lift legs (see [10]). *IEEE-1394* [17] is a standard that models a high performance serial multimedia bus for connecting together a collection of systems and devices in order to carry all forms of digitized information. The specification describes the link layer of the protocol. *Leader* models a protocol for leader election that is implemented in the IEEE-1394. Information about the two latter specifications can be found at [20]. *Splice* is a high performance publish-subscribe architecture. The specification analyzed presents a simple producer-transformers-consumer (see [14]). *CCP* models a cache coherence protocol for distributed multi-threaded Java programs (see [19]). *Bad Client* [6] is a security protocol for fair exchange of items using a smart card in mobile environments.

In all the cases but one the percentage of internal transitions is much higher using the abstract distribution than using the random one. In the case of the *Leader* specification it is 6.5 times better. The only non-positive result is given by the *Bad Client* protocol. In this case the number of internal transitions is slightly worse than in the random case. This bad result may come from the fact that the number of abstract states of the initial abstraction is too small in relation with the number of concrete states or from a wrong selection of the initial abstraction. Even though the last non-positive result, in the rest of the cases the gain is considerable.

The random distribution maximizes the balance of the system, the worst case balance is typically very close to the average. For the abstraction guided distribution the balance is not that perfect. The only case that the loss of balance is important is for the *Splice* system in which one worker receives more than half of the states. This is due to the very small number of abstract states of the selected abstraction, which, however, leads to a very good transitions ratio.

The abstractions are computed in few seconds and the complexity of the state space generation is not incremented by performing the abstraction guided distribution. It is only needed to compute the mapping function from concrete states to abstract classes for every new state. The final redistribution is quickly computed due to the very small number of classes.

6 Conclusions and Future Work

We presented an efficient method of partitioning very large state spaces, based on Abstract Interpretation techniques. The information given by the abstract graph is used to distribute the concrete state space in such a way that the number of cross transitions becomes small, while the block sizes are kept balanced.

Although good abstractions are hard to define because they require a good insight into the analyzed system, *any* abstraction serves our goal of predicting the connectivity of the concrete states. Therefore, standard automatically generated abstractions can also be used. Moreover, this method is very well scalable: the abstract graph is always small (a few hundred transitions at most), regardless the size of the concrete state space.

The next phase of the experiments will consist in doing some tests about the performance speed-up of the distributed model checking tools. For this purpose, we have to integrate the methodology in the existing distributed tools. We consider

that the measurements presented above are a good indication about the final results, therefore we expect to increase the performance, specially for the cycle reduction algorithm in which the distribution of transitions plays a very important role. Our method is completely scalable and has no extra cost for bigger instances of systems.

As we have seen, the algorithms used on the experiments are very simple. We believe that using more sophisticated algorithms may produce better distributions of states and transitions. We are currently working on an interface of our implementation with the software package Chaco [12], which provides a collection of efficient graph partitioning algorithms.

The abstraction guided approach is "model independent", it will likely work for other state space representations, like Kripke structures, Markov chains and Petri nets. It would be also interesting to see if it can be applied to symbolic (BDD based) model checking.

The fact that the abstraction is computed directly from the system specification allows the use of this method for on-the-fly model checking as well. In this case, because the number of concrete states per abstract state is not known, the balance must be guaranteed in a different way, for example by implementing a dynamic redistribution of abstract states.

References

[1] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL model-checking in SPIN. In *Proceedings SPIN'01*, *LNCS* 2057, pages 200–216, 2001.

[2] G. Behrmann, T. Hune, and F.W. Vaandrager. Distributed timed model checking - How the search order matters. In *Proceedings CAV'00*, *LNCS* 1855, pages 216–231, 2000.

[3] S.C.C. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proceedings CAV'01*, *LNCS* 2102, pages 250–254, 2001.

[4] S.C.C. Blom and S.M. Orzan. Distributed branching bisimulation reduction of state spaces. In *Proceedings PDMC'03*, *ENTCS* 89.1, 2003.

[5] S.C.C. Blom, I. van Langevelde, and B. Lisser. Compressed and distributed file formats for labeled transition systems. In *Proceedings PDMC'03*, *ENTCS* 89.1, 2003.

[6] J.G. Cederquist and M.T. Dashti. Formal analysis of a fair payment protocol. In *Proceedings FAST'04*, 2004. To Appear.

[7] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. *INFORMS Journal of Computing*, 10(1), pages 82–93, 1998.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings POPL'77*, pages 238–252, 1977.

[9] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *TCS*, 1, pages 237–267, 1976.

[10] J.F. Groote, J. Pang, and A.G. Wouters. A balancing act: analyzing a distributed lift system. In *Proceedings FMICS'01*, pages 1–12, 2001.

[11] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62, 1995.

[12] B. Hendrickson and R. Leland. The chaco user's guide: Version 2.0. Technical Report SAND94–2692, Sandia, 1994.

[13] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proceedings CAV'00*, *LNCS* 1855, pages 20–35, 2000.

[14] J. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings SAC'02*, pages 351–358, 2002.

[15] N.D. Jones and F. Nielson. Abstract Interpretation: A Semantics-Based Tool for Program Analysi. In *Handbook of Logic in Computer Science*, pages 527–636, Oxford University Press, 1995.

[16] K.G. Larsen and B. Thomsen. A modal process logic. In *Proceedings LICS'88*, pages 203–210, 1988.

[17] S.P. Luttik. Description and formal specification of the Link Layer of P1394. In *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design*, 1997.

[18] S.M. Orzan and J.C. van de Pol. Detecting strongly connected components in large distributed statespaces. Technical report, CWI, 2004. To Appear.

[19] J. Pang, W. Fokkink, R. Hofman, and R. Veldema. Model checking a cache coherence protocol for a Java DSM implementation. In *Proceedings IPDPS'03*, pages 238–253, 2003.

[20] C. Shankland and M. van der Zwaag. The tree identify protocol of IEEE 1394 in μCRL. In *Formal Aspects of Computing*, pages 509–531, 1988.

[21] J.C. van de Pol and M. Valero Espada. Verification of JavaSpaces parallel programs. In *Proceedings ACSD'03*, pages 196–205, 2003.

[22] J.C. van de Pol and M. Valero Espada. An abstract interpretation toolkit for μCRL specifications. In *Proceedings FMICS'04*, 2004. To Appear.

[23] J.C. van de Pol and M. Valero Espada. Modal abstraction in μCRL. In *Proceedings AMAST'04*, *LNCS* 3116, pages 409–423, 2004.

[24] W.M. Zuberek. Performance study of distributed generation of state spaces using coloured Petri nets. In *Proceedings CPN'02*, 2002.