Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 128 (2005) 91–105
www.elsevier.com/locate/entcs

UppDMC: A Distributed Model Checker for Fragments of the μ-Calculus

Fredrik Holmén[1] Martin Leucker[2] Marcus Lindström[3]

IT Department
Uppsala University
Uppsala, Sweden

Abstract

We present UppDMC, a distributed model-checking tool. It is tailored for checking finite-state systems and μ-calculus specifications with at most one alternation of minimal and maximal fixed-point operators. This fragment is also known as L^2_μ. Recently, efficient game-based algorithms for this logic have been outlined.
We describe the implementation of these algorithms within UppDMC and study their performance on practical examples. Running UppDMC on a simple workstation cluster, we were able to check liveness properties of the largest examples given in the VLTS Benchmark Suite, for which no answers were previously known.

Keywords: model checking, μ-calculus, game-based model checking

1 Introduction

Model checking [5] is a powerful technique for verifying complex hardware and software systems. However, the so-called *state-space explosion* still limits its application. While *partial-order reduction* or *symbolic model checking* reduce the state space by orders of magnitude, typical verification tasks still take modern sequential computers to their memory limits. One direction to enhance the applicability of today's model checkers is to use the accumulated

[1] Email: frho6915@student.uu.se
[2] Email: Martin.Leucker@it.uu.se This author is supported by the European Research Training Network "Games".
[3] Email: mali1741@student.uu.se

doi:10.1016/j.entcs.2004.10.021

memory (and computation power) of parallel computers. This observation has led to the development of parallel model checking algorithms in recent years.

A well-known logic for expressing specifications is Kozen's μ-calculus [11], a temporal logic offering Boolean combinations of formulae and, especially, labelled *next*-state, minimal and maximal fixed-point quantifiers. The (dependent) nesting of minimal- and maximal fixed-point operators forms the alternation depth hierarchy of the μ-calculus. The complexity of model checking, on the other hand, grows exponentially with the alternation depth for all known algorithms. It is then reasonable to limit the alternation depth of a formula to a practical important level and to develop efficient algorithms for this class of problems. In particular, we only need alternation depth 2 to capture the expressive power of `LTL` and `CTL`* [6].

In [3] and [12], parallel model checking algorithms for μ-calculus formulae of alternation depth 1 and respectively 2 were outlined. These fragments are known as L^1_μ and L^2_μ, respectively. The algorithms are based on a characterization of the model-checking problem in terms of two-player games [8,14]. More specifically, the algorithms describe how to color a game graph in parallel in order to answer the underlying model-checking problem.

For the algorithm for L^2_μ, it was shown that the game graph can be decomposed into components that can (after some simple modifications) easily be colored using a coloring algorithm for games obtained by formulae of L^1_μ. For this, a parallel coloring algorithm was introduced in [3], which is used as a subroutine in coloring graphs for L^2_μ.

In this paper, we describe the actual implementation of the algorithms for workstation clusters and supercomputers, in a system called UppDMC. The system is developed in C++ using the message passing standard MPI [9] for communication among the different computers. The current version is mainly intended to show the effectiveness and benefits of parallel model checking in practice, still leaving opportunities for optimizations. The system is available at `http://www.it.uu.se/research/project/parallelMC`.

Furthermore, we study the performance of the system on several industrial examples, especially on the VLTS Benchmark suite. [4] VLTS stands for "Very Large Transition Systems" and is a collection of (edge-)labelled transition systems. The VLTS benchmarks have been obtained from various case studies about the modelling of communication protocols and concurrent systems. Many of these case studies correspond to real life, industrial systems.

We checked two formulae for all 40 transition systems in VLTS on a varying number of machines in a Linux cluster. The cluster consists of 25 machines,

[4] `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`

each equipped with two 500 MHz Intel Pentium III processors and a main memory of 512 MB. The machines where interconnected with a standard 100 MB switched Ethernet network. The formulae checked are `nodeadlock` and `livelock` since these are the properties accounted for in VLTS and can be checked on all of them, independent of the actions used. We show that almost all properties on all transition systems can be checked within 50 seconds up to 10 minutes. Only checking `livelock` on the largest system requires more than the total available memory of the machines and needs swapping that slows down the running time to about 2.5 hours. Note that many of the systems have not been able to be checked for live-locks previously.

The area of parallel model checking has gained interest in recent years. In [13], a parallel reachability analysis is carried out. The distribution of the underlying structure is similar to the one presented here but their algorithm is not suitable for model checking temporal-logic formulae. A notable step is done in [10], in which a symbolic parallel algorithm for the full μ-calculus is introduced. [4] presents a model-checking algorithm for `LTL` using a costly parallel cycle detection. Another model-checking algorithm for `LTL` is introduced in [1], based on a nested depth-first search. Our approach, however, explicitly uses the structure of game graphs for L^2_μ.

In Section 2, we recall the main ideas of the implemented model-checking algorithms. We describe the implementation in Section 3. In Section 4, the running time and memory performance of the system is studied.

2 Gist of the parallel model checking algorithms

To make the forthcoming implementation section more understandable, we recall the main ideas of the parallel algorithms that are implemented. However, we restrict our exposition to the algorithm for L^1_μ [3]. Instead of formal definitions, we just give examples. See [3] and [12] for further details.

Let *Var* be a set of fixed-point variables and $\mathcal{A}$ a set of actions. Formulae of the modal μ-calculus over *Var* and $\mathcal{A}$ in positive form as introduced by [11] are defined as follows:

$$\varphi ::= \texttt{false} \mid \texttt{true} \mid X \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [K]\varphi \mid \langle K\rangle\varphi \mid \nu X.\varphi \mid \mu X.\varphi$$

where $X \in \mathit{Var}$ and $K \subseteq \mathcal{A}$.[5] For a formula φ of the μ-calculus, we introduce the notion of *subformulae*, *free* and *bound* variables, and *sentences* as usual. Every formula can be represented by its *graph*. This can be partitioned into *components* based on fixpoint formula. Figure 1(a) shows the graph and its

[5] $\langle -\rangle\varphi$ is an abbreviation for $\langle \mathcal{A}\rangle\varphi$.

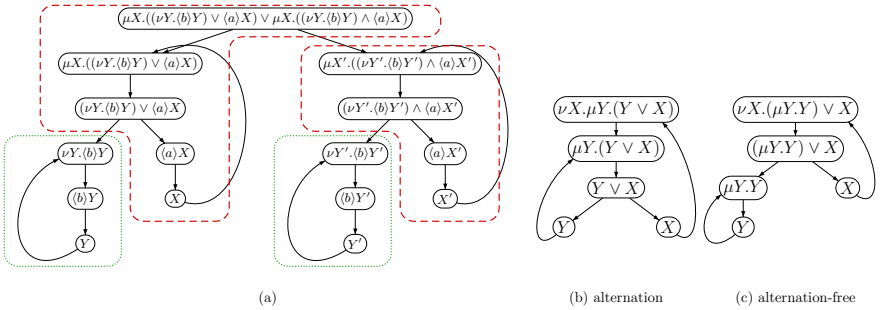

Fig. 1. Graphs of formulae

decomposition for the formula $\Phi = \mu X.((\nu Y.\langle b\rangle Y)\vee\langle a\rangle X)\vee\mu X'.((\nu Y'.\langle b\rangle Y')\wedge\langle a\rangle X')$.

Alternation describes the (dependent) nesting of minimal and maximal fixpoint formulae (see Figures1(b) and 1(c)). To make the procedure produce the right result, we have to limit alternation. However, for the outline of the algorithm this is not important now.

In game-based model checking, a given transition system is combined with the graph of the formula according to *game rules* to a so-called *game graph*. Figure 2 shows a game graph for a transition system that has two states s_1 and s_2, an a-loop from s_1 to itself, and a b-edge from s_1 to s_2, and the formula Φ. The decomposition of the formula's graph induces a partition of the game graph that is shown by dashed and dotted lines. Note that the game graph can be significantly smaller than the product of the sizes of the transition system and the size of the formula.

It can be shown, that a node (s, φ) of the game graph can be labelled green iff s satisfies φ and red otherwise. Thus, the essential things for a parallel model checking algorithm is the construction, distribution, and coloring of

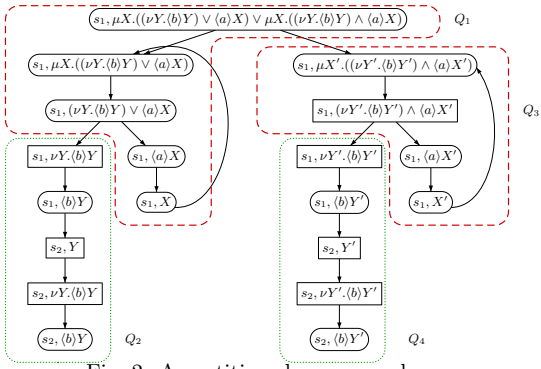

Fig. 2. A partitioned game graph.

this game graph in parallel.

The general ideas for this are as follows: Given a transition system and an L^1_μ-formula, our approach is both to construct the game graph as well as to determine the color of its nodes in parallel. The idea of our parallel algorithm is that all processors are working in parallel on one component, whereas the components are treated one-by-one.

Distributing the game graph

We employ a somehow standard approach distributing and constructing a (component of the) game graph in parallel [13,2]. As a data structure, we employ adjacency lists. We need also links to the predecessor as well as to the successor of a node for the labelling algorithm. A component is constructed in parallel by a typical breadth-first strategy. Given a node q, we determine its successors $q_1, \ldots, q_n$. To obtain a deterministic distribution of the configurations over the workstation cluster, one takes a function in the spirit of a hash function assigning to every configuration an integer and subsequently its value modulo the number of processors. This function f determines the location of every node within the network uniquely and without global knowledge. Thus, we can send each $q \in \{q_1, \ldots, q_n\}$ to its processors $f(q)$. If q is already in the local store of $f(q)$, then q is reached a second time, hence the procedure stops. If predecessors of q were sent together with q, the list of predecessors is augmented accordingly. If q is not in the local memory of $f(q)$, it is stored there together with the given predecessors as well as all its successors. These are sent in the same manner to their (wrt. f) processors, together with the information that q is a predecessor. The corresponding processes update their local memory similarly.

Please consult [2] for a thorough discussion of this and other possible approaches storing distributed transition systems.

Labelling the game graph

As explained in the previous paragraph, it is easy to construct (a component of) the game graph in parallel employing a breadth-first search. When a terminal configuration is reached, a backwards coloring process can be initiated. This can be carried out in parallel in the obvious manner. The main observation established in [3] is that after all color information is propagated all remaining uncolored nodes can be colored on every computer in parallel without any further communication. To check that all color information has been propagated, a distributed termination-check algorithm is employed.

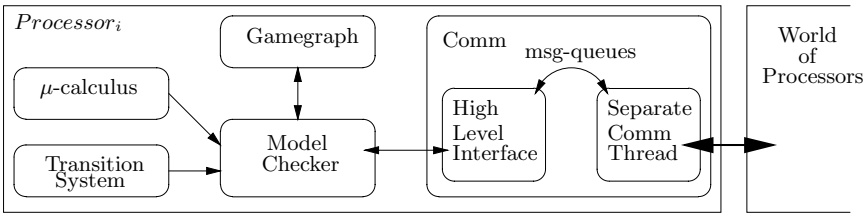

Fig. 3. Simplified structure of UppDMC.

3 Implementation

We have implemented the previously mentioned algorithms in a system called UppDMC . The system is developed in C++ using the message passing standard MPI [9] for communication among the different computers. It does not depend on the previous implementations in Haskell or C++ [2] and is more focused on performance. The current version is mainly intended to show the effectiveness and benefits of parallel model checking in practice.

While some of the algorithms in [3] and [12] can be carried out *on-the-fly*, the current version of UppDMC only makes partial use of it. Especially in the measurements shown in the next section, we work on previously generated transition systems. This only due to practical reasons: To be able to compare our system with existing model checkers, we use the precomputed transition systems made available as the VLTS benchmark suite. [6] It is easy to adapt our system to one behaving in an on the fly manner.

Figure 3 gives an overview of the general structure of UppDMC. The implementation is divided into modules for ease of design and testing, as well as for reuse in larger model-checking platforms. Each of the modules implements either a class or a template as an interface to the rest of the program.

μ-calculus module

The μ-calculus module basically contains functionality for parsing and analyzing the formulae that are read into the model checker. We pre-calculate the graph version of the formula and its division into components. The graph of a formula is simply a parse tree of the formula extended with edges from all fixpoint variables to the node they are defined in. It is used when building the game graph. Here we also figure out which of the fixpoint variables are alternating. Each subformula of the graph is assigned a number used to represent the subformula in the other parts of the model checker.

[6] http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html

Transition system module

The transition system module reads transitions and provides search functions for accessing the set of successors of a given state and action. Currently, the complete transition system is read into memory before the model checking starts. This is because we check previously computed transition systems that are stored on disk. When on-the-fly behavior is demanded, this can easily be changed. To save memory we store only the transitions that might be needed on each processor. A transition might be used on the processor numbered j if the transition looks like $s_1 \rightarrow s_2$ and $f(s_1) = j$, where f is the state distributing hash function. The transitions are stored sorted by predecessor and label so that transitions for a predecessor-label pair are found in $O(\log_2 k_j)$ time where k_j is the number of unique predecessor-label pairs in the set of transitions stored on the processor j. The memory usage for storing the transitions of a predecessor/label pair is 12 bytes if there is only one such transition and $12 + 4 * t$ bytes if there are $t > 1$ such transitions.

Game graph module

The game graph module defines the main data structure of the model checker. Game graph nodes are stored hashed on subformula to avoid storing the subformula in each node. Storing nodes grouped by subformula makes it easy to find the initial nodes of a component since all initial nodes in a component have the same subformula. Recall that the parallel algorithm works on components of the game graph one after the other. Therefore, the initial nodes of components are needed.

The algorithm always propagates colors towards the root of the graph in a backward manner. Thus, storing references to successors is unnecessary, while the predecessors have to be established. The subformula of a predecessor node is known except for the predecessor nodes of variables and μ- or ν-nodes. Here we store the subformula of each predecessor node explicitly. This is because they may have more than one predecessor in the representation of the graph. Hence predecessors are stored only as states when it is possible, otherwise we store both formula and state (see also Figure 1(a)).

The labelling of the nodes in the coloring algorithm is roughly as follows. If it is a kind of or-node, it becomes green when one successor is green, and red, if all successors are red. Checking the latter could be costly when successor nodes and parent nodes are placed on different computers. In general, this suggests to keep information of colors of children also in the parent node to avoid this. However, for our algorithm, we can do better: The algorithm implies that successors only change their color once. Hence it is sufficient to just count the number of red and green successors given that we know the total

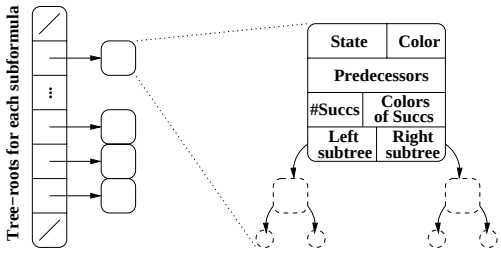

Fig. 4. Game-graph data structure.

number of successors. This greatly improves the handling of edges crossing processors since no copies of nodes have to be stored for color administration.

The typical structure of a game graph is depicted in Figure 4. It consists of the following:

- For each subformula we store a binary search tree of the game graph nodes that contain the subformula.
- For each game graph node we store the following in the search tree:
  - a state,
  - a color,
  - predecessor information(states when we know the preceding subformula, subformula state pairs when we do not),
  - the total number of successors,
  - the number of red and green successors,
  - a reference to the right subtree, and
  - a reference to the left subtree.

One significant modification to the algorithms sketched in the previous sections is the removal of unnecessary nodes. This is done by never storing nodes that have only one successor except for initial nodes and alternating variables that are never removed.

Each game graph node occupies about 36 bytes of memory plus the data needed to store the predecessor of the node.

To get a picture of the memory usage lets assume a total main memory capacity of 1 TByte which is not unrealistic for a fairly modern cluster.[7] Assuming an average branching factor of 10, which is not unreasonable considering the branching factors in VLTS, such a system could handle a transition system of about $2 * 10^9$ states if we assume that we only store the transition system and the game graph in memory. These numbers assume the livelock formula found below.

[7] For example, the Sun Fire SMP-cluster installed at RWTH Aachen has a total of 1 TByte http://www.rz.rwth-aachen.de/computing/hpc/sun/

The transition system and game graph data structures are designed to give a reasonable time as well as memory efficiency. They are not optimized from a memory usage perspective. However, our focus has been to check the practical applicability of the algorithm and not to integrate it with more efficient data structures (like BDDs). This could be a direction for future work.

Communication

The communication module provides one big class. It provides a high level interface for sending and receiving different kinds of messages used in both the algorithm and the termination detection. When an object is created of this class a separate thread is created for network communication. The class contains a protected data structure that stores all the inbound and outbound messages. The data structure is accessed by the network thread as well as the model checker module. This makes the implementation of the algorithm independent of the method of communication. We have implemented communication classes that uses both MPI and TCP/IP. The MPI version buffers messages to better utilize the network bandwidth. Several messages are then sent in one packet if the buffer is full or a time out flushes the buffer. Since the communication to a large part is the bottleneck the size of the buffers and the time out interval have great influence on overall performance. If packets are sent too often with too few messages the network will become congested. If on the other hand packets are sent with too large intervals the receiver might be waiting with nothing to do, wasting valuable processor time.

Termination detection module

The termination detection module is used to figure out when one step of the algorithm, for example expansion of a component, ends and the next, for example recoloring, can start. We use a termination detection algorithm by Dijkstra [7]. The algorithm is based on a virtual token ring formed by all the processors. One of the processors is a termination detection server. This processor initiates the termination detection and is responsible for telling the other processors when termination has occurred. For short the algorithm lets the token circle the token ring changing color depending on the state of the computation on each processor. The termination detection server can then tell if the computation has terminated or not when the token returns.

Implementation of the algorithm

The model checker module implements a template with a formula object, a transition system object and communication object as parameters. The implementation closely resembles the algorithm sketched in the previous section

Property	Formula
No Deadlock	$\nu X.([-]X \wedge \langle - \rangle \texttt{true})$
Livelock	$\mu X.(\langle - \rangle X \vee \nu Y.(\langle \tau \rangle Y))$

Fig. 5. The two formulae used during testing.

Name	# of states	# of transitions	nodeadlock (s)	livelock (s)
vasy_2581_11442	2,581,374	11,442,382	44 s	47 s
vasy_4220_13944	4,220,790	13,944,372	56 s	67 s
vasy_4338_15666	4,338,672	15,666,588	64 s	64 s
vasy_6020_19353	6,020,550	19,353,474	59 s	125 s
vasy_6120_11031	6,120,718	11,031,292	95 s	108 s
cwi_7838_59101	7,838,608	59,101,007	149 s	314 s
vasy_8082_42933	8,082,905	42,933,110	162 s	134 s
vasy_11026_24660	11,026,932	24,660,513	150 s	160 s
vasy_12323_27667	12,323,703	27,667,803	160 s	177 s
cwi_33949_165318	33,949,609	165,318,222	560 s	8715 s

Fig. 6. The running time (s) for ten of the largest transition systems in VLTS when running on 25 machines.

and described in detail in [3] and [12]. The main differences were mentioned in relation to the game graph above.

4 Practical Experiences

We checked two formulae for all 40 transition systems in VLTS on 3,6,12,18 and 25 machines in a Linux cluster. Each machine had two 500 MHz Intel Pentium III processors and a main memory of 512 MB. The machines where interconnected with a standard 100 MB switched Ethernet network. The formulae, seen in Figure 5, were nodeadlock and livelock since these are the properties accounted for in VLTS. We run two threads on each machine, one is the communication thread and the other runs the algorithm. Since our machines have two processors and the threads may run in parallel we might get a slight performance improvement compared to running on single processor machines.

Our tests showed that the algorithm presented in [12] can be used in practice for verification of large models. In Figure 6 we see the running times, in seconds, for some of the largest transition systems in VLTS on 25 machines. The times are the running time of the algorithm only, reading of formulae and transition systems are not included. In almost all examples, the answer was given within seconds or minutes. The long running time for livelock on cwi_33949_165318 is due to swapping when memory is full.

The memory usage of the game graph is the real bottleneck. A theoretical

Name	$k * n$	# game graph nodes	predecessors (bytes)	estimated (bytes)	real (bytes)
vasy_2581_11442	15,488,244	10,113,184	97,007,420	923,733,008	461,082,044
vasy_4220_13944	25,324,740	16,259,519	141,316,760	1,357,910,544	726,659,444
vasy_4338_15666	26,032,032	16,427,458	153,659,948	1,438,483,968	745,048,436
vasy_6020_19353	36,123,300	24,082,199	218,734,872	1,919,749,968	1,085,694,036
vasy_6120_11031	36,724,308	22,087,877	152,118,976	1,675,076,432	947,282,548
cwi_7838_59101	47,031,648	31,354,431	402,914,512	3,584,371,552	1,531,674,028
vasy_8082_42933	48,497,430	27,081,699	399,108,828	3,119,767,000	1,374,049,992
vasy_11026_24660	66,161,592	43,652,440	327,786,140	3,170,953,728	1,899,273,980
vasy_12323_27667	73,942,218	48,788,082	367,199,944	3,547,289,544	2,123,570,896
cwi_33949_165318	203,697,654	134,404,263	1,724,364,396	12,623,298,648	6,562,917,864

Fig. 7. The memory usage compared to the worst-case estimated upper limit for no deadlock runs.

upper limit for the number of nodes in the game-graph is $k * n$, where k is the number of subformulae in the formula and n is the number of states in the transition system. This corresponds to a scenario where all possible pairs of subformulae and states are reachable. Since each game-graph node occupies 36 bytes they occupy a total of $36 * k * n$ bytes. Since the memory needed for the predecessors depends on the formula we must count how many of the subformulae that results in the predecessor begin stored in 4 bytes and 8 bytes respectively. We call the number of subformulae resulting in 4 byte storage r. We then end up with the worst-case formula $36*k*n+4*r*m+8*(k-r)*m$ for approximating the size of the total game graph, where m is the number of transitions. In this formula we assume that all subformulae have the same probability of ending up in the game graph.

In Figure 7 we compare this worst-case estimation with measurements of the memory usage on ten transition systems from VLTS.

How good this simple approximation estimates the real memory usage is highly dependant on which formula we are checking. Since the nodeadlock formula only has $-$ as label in the modal operators the size of the game graph will be closer to $k * n$ than in a case where we use specific labels. In Figure 7 we can see that the approximation is about two times bigger than the real value. This difference is probably even larger for other formulae. In other words, the more specific the formula is, the more likely it is that parts of the game graph are not constructed.

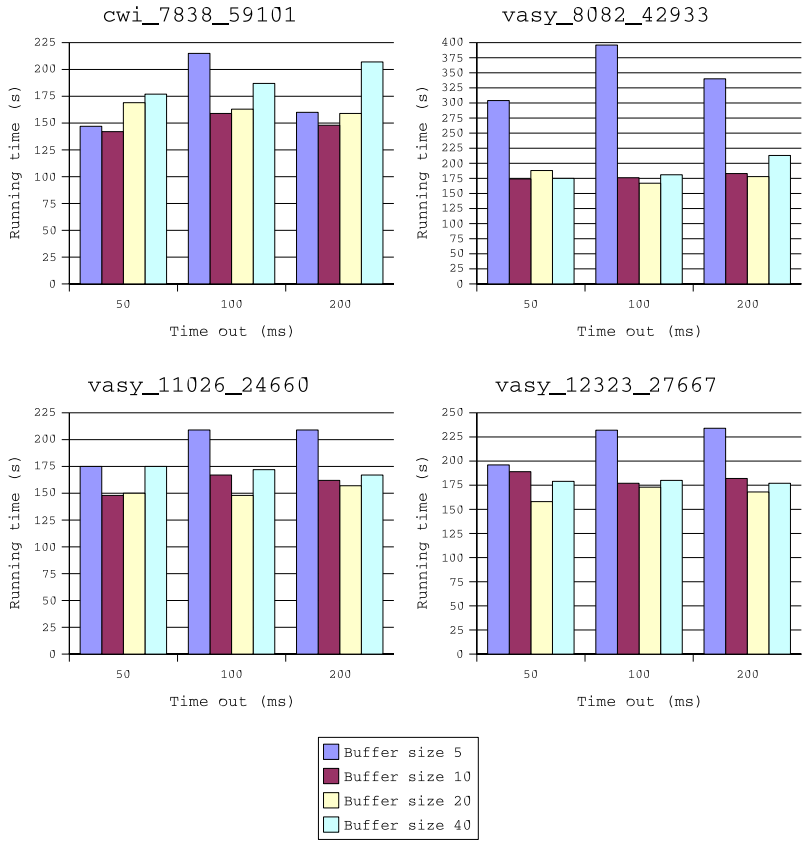

Fig. 8. The impact of changing the communication parameters buffer size and time out, on the running time.

In Figure 8 we see the impact of different buffer sizes and time outs for the communication. We see some variation in the running time with changes in buffer size. A buffer size of 10 to 20 messages seems to be the most efficient for these particular systems. The variation in time out does not seem to be as important for the running time as the buffer size. All these variations are small when compared to running times without buffering. The running times with no buffering for these systems are, 21769 s for cwi-7838-59101, 83008 s for vasy-8082-42933, 26174 s for vasy-11026-24660 and 29917 s for vasy-12323-27667. The key point is that buffering lowers the running time dramatically, while the exact values for buffer size and time out appear to be not that significant.

In Figure 9 we can see the effect of increasing the number of machines on the running time. The structure of the transition system can limit the

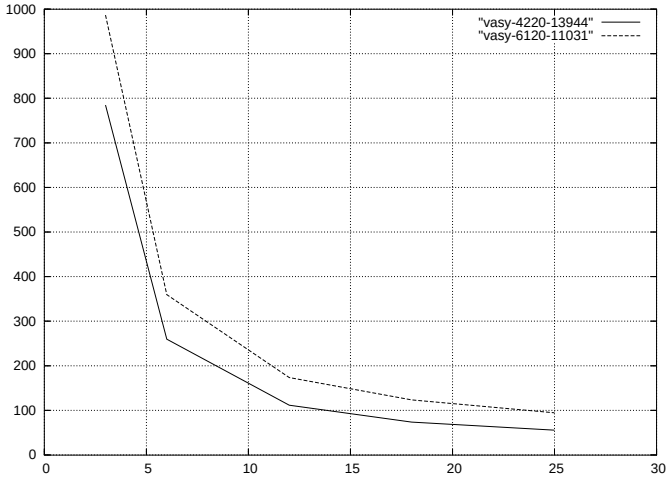

Fig. 9. The running times of the two transition systems vasy-4220-13944 and vasy-6120-11031 on 3,6,12,18 and 25 processors. The horizontal scale is number of processors and the vertical scale is time in seconds.

Transition system	Livelock
vasy_8082_42933	Not satisfied
vasy_11026_24660	Not satisfied
vasy_12323_27667	Not satisfied
cwi_33949_165318	Satisfied

Fig. 10. The livelock results from VLTS that where previously n/a.

effect of increasing the number of machines. If the system has a very low branching factor the computation is doomed to be largely sequential since it is the branching of the game graph that makes parallelization possible. Furthermore, we see that parallel power mainly benefits when it makes use of the accumulated memory. The examples were too small to profit a lot from more than 12 computers.

While our measurements show that the algorithm behaves and scales well in practice, we want to mention a more important point. Most of the larger examples in the VLTS benchmark suite have not been previously checked for live-locks. This is due to the fact that current model checker were not able to do so. UppDMC, however, managed to answer all question within a short time. For practical applicability of model checking tools, the most important feature is to get an answer. Figure 10 shows the results that we have obtained.

5 Conclusion

In this paper, we have described the implementation of a *parallel* game-based model-checking tool for an important fragment of the μ-calculus. It can be used to check LTL and CTL* as well. Furthermore, we have examined the performance of the model checker on real-life examples. We were able to get answers for systems that could not be handled before.

As future work, it would be interesting to combine our model checking tool with a state-space generator so that a modelling and model-checking workbench could be provided.

References

[1] J. Barnat, L. Brim, and I. Černá. Property driven distribution of nested DFS. In *VCL 2002: The Third International Workshop on Verification and Computational Logic*, Pittsburgh PA, 2002.

[2] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation free μ–calculus. Technical Report AIB-04-2001, RWTH Aachen.

[3] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation-free mu-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*, volume 2318 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., 2002.

[4] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model-checking based on negative cycle detection. In *Proceedings of 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'01)*, Lecture Notes in Computer Science. Springer, Dec. 2001.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[6] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, Apr. 1994.

[7] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, Aug. 1980.

[8] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In *Proc. 5th International Computer-Aided Verification Conference*, volume 697 of *Lecture Notes in Computer Science*, Springer, 1993.

[9] The MPI Forum. Document for a Standard Message-Passing Interface. CS-93-214, University of Tennessee, 11 1993.

[10] O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for μ-calculus. In *Proceedings of the 13th Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 350–362. Springer, July 2001.

[11] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, Dec. 1983.

[12] M. Leucker, R. Somla, and M. Weber. Parallel model checking for LTL, CTL* and L^2_μ. In L. Brim and O. Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier Science Publishers, 2003.

[13] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In O. Grumberg, editor, *Computer-Aided Verification, 9th International Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer, June 1997.

[14] C. Stirling. Games for bisimulation and model checking, July 1996. Notes for Mathfit Workshop on finite model theory, University of Wales, Swansea.