# McScM: A General Framework for the Verification of Communicating Machines

Alexander Heussner, Tristan Le Gall, Grégoire Sutre

# McScM: A General Framework for the Verification of Communicating Machines*

Alexander Heußner[1], Tristan Le Gall[2], and Grégoire Sutre[3]

[1] Université Libre de Bruxelles, Brussels, Belgium
[2] CEA, LIST, DILS/LMeASI, Gif-sur-Yvette, France
[3] Univ. Bordeaux & CNRS, LaBRI, UMR 5800, Talence, France

**Abstract.** We present McScM, a platform for implementing and comparing verification algorithms for the class of finite-state processes exchanging messages over reliable, unbounded FIFO channels. McScM provides tools for the safety verification and controller synthesis of these infinite-state models. Our verification tool implements several model-checking techniques: CEGAR with different abstraction-refinement methods, abstract interpretation, abstract regular model checking, and lazy abstraction. Seen as a general framework for the class of transition systems with finite control/infinite data, McScM delivers the basic infrastructure for implementing verification algorithms, and privileges to conveniently implement new ideas on a high level of abstraction. It also allows us to compare and benchmark different algorithmic approaches with the same backend.

## 1 Introduction

The automatic verification of distributed algorithms and communication protocols is one of the most crucial tasks in software/hardware development and maintenance. It is also one of the hardest, e.g., as one cannot directly infer the global behaviour of a distributed system from its local components due to asynchronous communication. This renders already simple analysis, verification, and synthesis questions hard problems in theory. However, in practice, this leads to a growing demand for versatile tools that also apply semi-algorithmic solutions, approximations, abstractions, and heuristics.

We focus on the safety verification of *communicating finite-state machines* (CM), an infinite-state formalism that consists of a set of local, finite state machines that communicate via global, asynchronous, reliable and unbounded FIFO channels. The latter are demanded in practice by, e.g., distributed applications based on TCP, the Sockets API, or MPI. Note that CM do not demand the channels to be a priori point-to-point. The safety verification question demands, given a CM and a set of "bad" states, whether no execution of this CM reaches the bad states. This is known to be undecidable [3].

* This work was partially supported by the ANR project Vacsim (ANR-11-INSE-004).

Cormac Flanagan and Barbara König (Ed.): Proc. of 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), March 2012. LNCS 7214, pp. 478–484, 2012. 

As for other classes of Turing-complete infinite-state models, there are two ways to tackle this problem. The first one is to restrict CM to a decidable fragment, e.g., by imposing a bound on the size of the FIFO channels, or assuming the channels to lose messages [1]. The second one is to provide only "partial" results, either by semi-algorithmic methods that may not terminate, or over-approximative approaches that may be inconclusive. Despite the rich theoretical work concerning CM, there is currently no versatile tool that can be directly applied to CM's safety question , and that gives the user a choice among different algorithmic approaches to solve a concrete verification question.

We aim at filling this gap by presenting a *Model Checker for Systems of Communicating Machines* (McScM) that combines different algorithms for the safety verification problem of CM under the same roof and provides a ready-to-use front-end with the tool `verify`. McScM is available via our project's web page [14] either as a precompiled binary distribution (including man pages, a suite of examples, and some benchmarking scripts), or as source code release. McScM is programmed in OCaml and available under a BSD license. The development of McScM takes place in a software forge [14] that provides a wiki for documentation (including man pages and API), a bug and issue tracker, as well as our theoretical work [6, 7]. The following discussion refers to McScM's release 1.2.

In the following, we present our implemented algorithms (Section 2) and `verify`'s modular architecture (Section 3) before applying the tool in a small comparative benchmark in Section 4 that shows its capabilities. Finally, we change the focus to McScM as generic API/framework (Section 5) to implement novel algorithms and ideas and compare to existing tools and frameworks.

## 2 Safety Verification of Communicating Machines

In our setting, an instance of the safety verification problem is given by a textual representation of a CM (in a simple and intuitive automata-based language), and a set of bad states, i.e., a set of global control states together with a representation of the channel's contents by regular expressions (for details, see `scm(5)` man page). The tool `verify` allows the user to input this instance and to choose among a variety of verification techniques. After completion of the analysis, `verify` outputs either "model safe" if it finds an inductive invariant that proves the system safe, or returns a counterexample, i.e., a proof that the system is not safe. The tool aborts if it runs out of a resource that was a priori limited by the user, e.g., the number of analysis steps or the maximum precision allowed for abstraction. A closer look on the modular architecture of `verify` and the generic aspects underlying McScM is postponed to the next section.

McScM currently implements the following four different verification techniques:

`absint`: this abstract interpretation based approach [8] reduces verification to the calculation of a fixpoint in an abstract lattice, and terminates in a finite number of steps with either a positive answer (model safe), or aborts.

armc: the *Abstract Regular Model Checking* semi-algorithm [2] refines a global regular abstraction of the system by symbolic successor (or predecessor) calculation; we reimplemented the basic idea in our setting;

cegar: *Counterexample Guided Abstraction Refinement* is a semi-algorithmic approach that allows to start with a rough, safety-conservative abstraction that is refined along spurious counterexamples [4]; McScM started originally by porting this approach to CM relying on a novel notion of path invariant based refinement [6]; the implemented generic algorithm allows for a variety of parameterization (in particular, path invariant generation methods);

lart: we implemented the lazy abstraction approach [9] based on the construction of an abstract reachability tree; each vertex of the tree contains an abstract region, which may be refined with the help of path invariants when needed;

We can compare the algorithms on the same background as they share an underlying infrastructure implementing abstraction/extrapolating for CM, as well as a library of graph algorithms and (path) invariant generators. The first three techniques are semi-algorithms based on the *abstract-check-refine* paradigm. When the CM is not safe, they provide *counterexamples* that are an important feedback when using safety verification in practical (engineering) situations. Contrariwise, absint always terminates without guaranteeing a conclusive answer. A comparison of the four approaches with respect to a suite of example protocols derived from practice follows in Section 4.

Already revealing the benchmark's outcome, there is no silver bullet among the four techniques. Hence, to tackle a given instance of a CM safety verification problem, one has to choose among approaches and need fine-grained influence by additional parameterizations to the underlying algorithms (e.g., depth-first versus breadth-first exploration of the CM). This is exactly what `verify` offers: a "swiss army knife" for model checking systems of communicating machines.

In addition, McScM includes a supervisory control tool: `control`. If a CM system does not satisfy a safety property, `control` automatically computes a restriction of this system that assures safety by implementing the distributed control algorithm presented in [7] (see `control(1)` man page for details).

## 3 A Closer Look on `verify`'s Modular Architecture

Figure 1 shows the modular architecture of McScM's `verify` tool. The latter provides a common (command line based) interface and infrastructure for the implemented verification algorithms (absint, armc, cegar, lart), as well as allows to plug in a symbolic representation of the infinite data part of the CM, i.e., the queue contents. Currently, we only provide a wrapper for a library based on queue-content decision diagrams (QDD) [15]. The tool's input is an instance of the safety decision problem. Each algorithm accepts additional adaptations via command line parameters. The tool outputs either a counterexample, a positive result, or an abort message. McScM provides additional logging and profiling information that can be output by the tool, and helps to benchmark and compare algorithms, as seen in the next section.

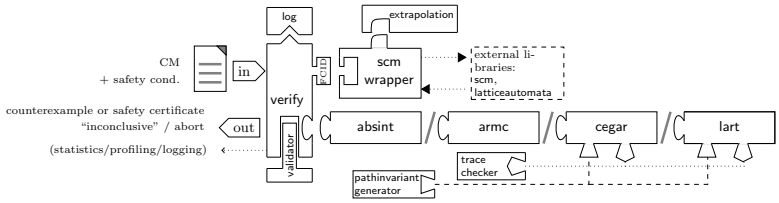


**Fig. 1.** Modular Architecture of McScM's Verification Tools

The cornerstone of McScM's development is *generic programming* which is supported by OCaml's modules and functors. This functional programming language based on type inference encloses the proof of behavioural guarantees at compile time with respect to our implemented algorithm's interfaces. In addition, we also provide means for the on-the-fly validation of both intermediary results (i.e., checking the result of a path invariant generation in cegar) and the inductive safety invariant. Both add an additional layer of *reliability*, especially when implementing new algorithms in McScM.

As our API is well specified and reasonably documented, it is relatively easy to implement also other algorithms for CMs. For example, the previously mentioned `control` tool is build of top of several OCaml modules of `verify`, and uses the same fixpoint computation as absint.

## 4 A Comparative Benchmark of Verification Algorithms

Figure 2 was generated by using `verify` to benchmark the included verification algorithms (on default parameters) on a suite of examples derived from practice. The latter includes the alternating bit protocol (ABP), a simplified version of TCP, and a distributed leader election algorithm ("Peterson"); the examples range from simple protocols with 5 global control states ("c/d") to around $10^4$ for Peterson. The benchmark was run on an off-the-shelf computer (3.2GHz Intel i7-965, 64-bit Linux) and is contained as shell script in the latest McScM release.

In general, `verify` is able to give a solution for each example in a reasonable amount of time and memory. However, there is no algorithm that proves to be superior. absint provides a fast way of determining if a protocol is safe, however, it is not able to cope with unsafe examples. Due to its termination guarantee, it proves to be ideal as first line of attack when trying to verify an unknown protocol. The main difference between cegar and armc is their way of refining the abstraction, either locally and adapted, or globally for the whole model. This gives an advantage for cegar in the examples that require a "precise" abstraction only for a few control loops (like the Peterson algorithm, the erroneous load balancer, or the token ring example), and for armc in most other cases. However, our armc implementation is not able to cope with a simple non-regular protocol. As cegar allows a variety of additional parameters to the algorithm, we can fill the two gaps in the table by changing the underlying path invariant generation (e.g., `-tc-engine apinv-fwd -k-min -1` leads to $13.48s/15.56MiB$ (BRP) and

| | ABP | BRP | c/d | load-balancer | load-balancer (err) | nested c/d | non-regular | Peterson | POP3 | server/2 clients | sliding windows | (simplified) TCP | TCP (error) | token ring | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ω | ω | ω | ω | ω | ω | b | b | b | ω | b | ω | ω | b | bounded |
| | ✓ | ✓ | ↯ | ✓ | ↯ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ↯ | ✓ | safe |
| absint | 0.05 | 1.70 | 0.02 | 0.00 | 1.72 | 0.00 | 0.07 | 85.00 | 1.18 | 298.6 | 6.53 | 0.10 | 0.16 | 27.43 | time ($s$) |
| | 2.97 | 5.88 | 2.97 | 2.75 | 4.91 | 2.97 | 2.97 | 54.31 | 6.84 | 5.58 | 7.81 | 3.94 | 3.94 | 14.44 | mem (MiB) |
| armc | 0.11 | 331.92 | 0.01 | 0.00 | — | 0.02 | — | 4.79 | 3.14 | 195.88 | 0.21 | 0.12 | 0.03 | 328.18 | |
| | 4.95 | 773.8 | 2.97 | 2.97 | | 2.97 | | 106.62 | 31.06 | 14.59 | 6.84 | 5.88 | 3.94 | 3143.52 | |
| cegar | 0.23 | — | 0.02 | 0.06 | 2.66 | 0.40 | 0.02 | 1.41 | 8.06 | — | 7.92 | 0.89 | 0.12 | 14.9 | |
| | 3.94 | | 2.97 | 2.97 | 7.91 | 3.94 | 2.97 | 46.56 | 14.59 | | 18.47 | 6.84 | 3.94 | 35.91 | |
| lart | — | — | 0.01 | 0.02 | 56.21 | — | 0.02 | 1.62 | 1184.4 | — | 437.81 | — | 0.01 | — | |
| | | | 2.97 | 2.97 | 16.53 | | 2.97 | 41.56 | 18.47 | | 73.69 | | 2.97 | | |

**Fig. 2.** Benchmarking `verify`'s Different Algorithms on a Suite of Examples (we denote an abort due to an $1h$ time limit by "—", and note for each example if it is safe (✓) or not safe (↯), as well as if queues are used in a bounded way ($b$) or without restriction ($\omega$); inconclusive results of absint are marked gray)

$5.67s/10.72MiB$ (server)); however, there is also no default parameterization for cegar that can be shown to be superior (see [6] for details).

To conclude, there is no silver bullet for the safety verification of CM among our algorithms; however, their common front-end via `verify` proves to be a flexible tool that can cope with all our examples.

## 5 The McScM Framework

McScM's generic approach is based on symbolic finite control infinite data transition systems (FCID) (a notion inspired by [5]); the latter are given by a finite transition system enhanced by an appropriate region algebra as symbolic representation of the infinite data. For CM, the region algebra is given by QDD, and we define a regular abstraction for our systems thereupon.

Thus, McScM provides a generic API for, on the one hand, implementing new algorithms on a high level of abstraction; and, on the other hand, allows to apply the implemented algorithms to other members of the FCID family, by supplying a fitting region algebra and a suited notion of abstraction.

*Related Tools:* McScM relates to other symbolic model checking tools that can verify CM or subclasses thereof. SPIN [16], for example, allows to verify only CM with a priori bounded channels (e.g., those marked $b$ in Figure 2), but allows for deciding properties specified in linear temporal logic. The CADP [10] toolbox includes a $\mu$-calculus model checking tool limited to finite labelled transition systems, i.e., CM with bounded channels only. The same restriction to finite transition systems holds for other tools, like LTSA [13]. TReX [18] analyzes infinite state systems: *lossy* channel systems with local timed/counter automata. LEVER [12] is a learning-based model checker that supported CM with regular channel languages in a previous, not further available version. So, McScM offers—to our knowledge—the only currently freely available tool that can directly verify CM with reliable, unbounded FIFO channels.

The LASH library [11] offers only symbolic data structures channels, but does not provide any model checking algorithm for CM. The LASH API permits to symbolically present several classes of FCID (e.g., by QDD, number decision

diagrams (NDD), real vector automata (RVA)), and to implement algorithms for each. Modular front-ends like TaPAS [17] (for FCIDs based on Presburger arithmetic) even allow to implement for multiple FCID libraries at once. Even tough McScM can be used in the same spirit to implement and compare model checking algorithms for a given class of FCID, we are able to provide *generic algorithms* that can be parameterized by any FCID for which we can provide a symbolic representation. The latter must only be conform to the above mentioned region algebra and supply an appropriate notion of abstraction, e.g., a suitable wrapper for LASH's RVA would directly port `cegar` to FCID representable by real vector automata, e.g. timed or hybrid systems.

*Future Work:* McScM is a work in progress, hence, we are always optimizing internals and provide extensions that prove handy for practical verification tasks, like our planned direct support for PROMELA as input language. Our next big step will lead beyond CM by allowing the local machines to have infinite data (like counters or timers), which demands new insights and notions for abstractions and invariants for these systems, as well as practicable algorithmic data-structures for implementing a region algebra.

## References

[1] P. Aziz Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1):71–90, 1996.
[2] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. of CAV'04*, *LNCS* 3114, 372–386, 2004.
[3] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.
[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50(5):752–794, 2003.
[5] T. Henzinger, R. Majumdar, and J.F. Raskin. A classification of symbolic transition systems. *ACM Transactions on Computational Logic*, 6:1–32, 2005.
[6] A. Heußner, T. Le Gall, and G. Sutre. Extrapolation-based path invariants for abstraction refinement of fifo systems. In *Proc. of SPIN'09*, *LNCS* 5578, 107–124. Springer, 2009.
[7] G. Kalyon, T. Le Gall, H. Marchand, and T. Massart. Global state estimates for distributed systems. In *Proc. of FMOODS/FORTE'11*, *LNCS* 6722, 198–212, 2011.
[8] T. Le Gall, B. Jeannet, and T. Jéron. Verification of Communication Protocols using Abstract Interpretation of FIFO queues. In *Proc. of AMAST'06*, *LNCS* 4019, 204–219. Springer, 2006.
[9] K. McMillan. Lazy abstraction with interpolants. In *Proc. of CAV'06*, *LNCS* 4144, 123–136. Springer, 2006.
[10] CADP. `http://www.inrialpes.fr/vasy/cadp/`.
[11] LASH. `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.
[12] LEVER. `http://abhayspace.com/static/lever.html`.
[13] LTSA. `http://http://www.doc.ic.ac.uk/ltsa/`.
[14] McScM. `http://altarica.labri.fr/forge/projects/mcscm`.

[15] Scm, Lattice Automata. http://gforge.inria.fr/projects/bjeannet/.
[16] SPIN. http://spinroot.com.
[17] TaPAS. http://altarica.labri.fr/forge/projects/3/wiki/TaPAS/.
[18] TReX. http://www.liafa.jussieu.fr/~sighirea/trex/.