The mCRL2 toolset

J.F. Groote, J.J.A. Keiren, A.H.J. Mathijssen, B. Ploeger,
F.P.M. Stappers, C. Tankink, Y.S. Usenko, M.J. van Weerdenburg,
J.W. Wesselink, T.A.C. Willemse, J. van der Wulp

Eindhoven University of Technology, Dept. of Mathematics and Computer Science

Abstract

We describe the toolset for the behavioural specification language mCRL2. The purpose of the toolset is to analyse abstract models that describe the communication behaviour of software based systems. With the help of the toolset we want to efficiently detect and prevent problems in software, preferably before it is built. The tools allow to transform specifications, generate and visualise state spaces, verify modal properties, and much more. In order to facilitate reuse of the code most of the functionality is included in libraries. This makes the toolset suitable as a platform for third party tool development and for other specification languages as well. The toolset is distributed under the Boost license, which permits such use.

1. Introduction

The mCRL2 language is a specification language for describing communication behaviour among systems. In general we consider communicating computers and computer programs, but the language is also suitable for other systems that exchange messages (for instance business processes or social networks). The language is supported by a toolset enabling simulation, visualisation, behavioural reduction and verification of software requirements. The purpose is to efficiently study and understand the communication patterns in software, which can be extremely complex, especially in parallel and distributed systems. As such the toolset is versatile, employing the power of automated tools when necessary, and human intuition and ingenuity when needed.

The behavioural part of the language is based on process algebra and by which the toolset is rooted in the same methodology as CADP [1], μCRL [2] (which is the direct predecessor of mCRL2) and FDR2 (based on CSP [3]). Like UPPAAL [4], mCRL2 allows to specify real-time behaviour. Other process specification formalisms are directly based on automata or a mixture of automata and programming languages (such as Promela [5]).

The data part of the language is based on higher-order abstract equational data types. It contains quantifiers, (unbounded) integers, (infinite) sets and bags, structured types, lists and real numbers. All these concepts are set up to be as close to their mathematical counterparts as possible. This means that the language is very expressive and it is easy to write down undecidable properties. For the decidable part advanced algorithms have been devised (such as just-in-time compiling rewriting [6]) giving the tools high performance despite the generality of the data types.

There is also a property specification language based on the modal μ-calculus [7] extended with data and time. We do not know of other property specification formalisms with the same expressivity, especially regarding its data part.

The mCRL2 toolset is based on human-guided transformation of specifications. Figure 1 illustrates the basic concepts of the mCRL2 toolset. Generally, an mCRL2 specification is first translated into a linear process where all parallel operators have been removed. There are several tools to optimise linear processes. Subsequently, they can be transformed into transition systems or, in combination with modal formulas, to parameterised boolean equation systems. For transition systems and parameterised boolean equation systems also a family of transformation tools exist. We experienced that for non-trivial verification tasks, human-guided application of the tools together with a good understanding of the specification, is the only route to success. To further aid the understanding, several visualisation tools are available.

We believe that it is important that not only the tools themselves but also their design and implementation are shared among the scientific community, much in the same way as scientific ideas are communicated and elaborated among groups of people. Therefore, we provide the toolset as open source under the Boost Software License [8], which allows free use of the software and its source, as long as the authors of the tools are acknowledged. Furthermore, we invested heavily in documenting the software to make the toolset a practical starting point for other behavioural analysis environments.

In this article we provide a short overview of the structure of the toolset. The toolset itself is available from the mCRL2 website [9].

2. Overview of the toolset

The mCRL2 toolset is comprised of several tools that allow the user to conduct verification and validation of systems specified in the mCRL2 language. An abstract view on the mCRL2 toolset is given in Figure 1. For instance, there are converters that allow importing other formalisms such as coloured Petri nets [10], χ [11] and μCRL [2]. The actual list of tools per category can be found on the mCRL2 website [9].

Every analysis of an mCRL2 specification (see Section 4.1 for examples) starts by a transformation of the specification into linear form. This is achieved by the *lineariser*, which transforms a restricted yet practical subset of mCRL2 specifications to *linear process specifications* (LPSs). These LPSs are a compact symbolic representation of the labelled transition system of the specification. Due to its restricted form, an LPS is especially suited as input for tools; there is no need for such tools to take into account all the different operators of the complete language. (See [12] for the details of the linearisation process.)

Once an LPS has been generated from an mCRL2 specification, the user has several

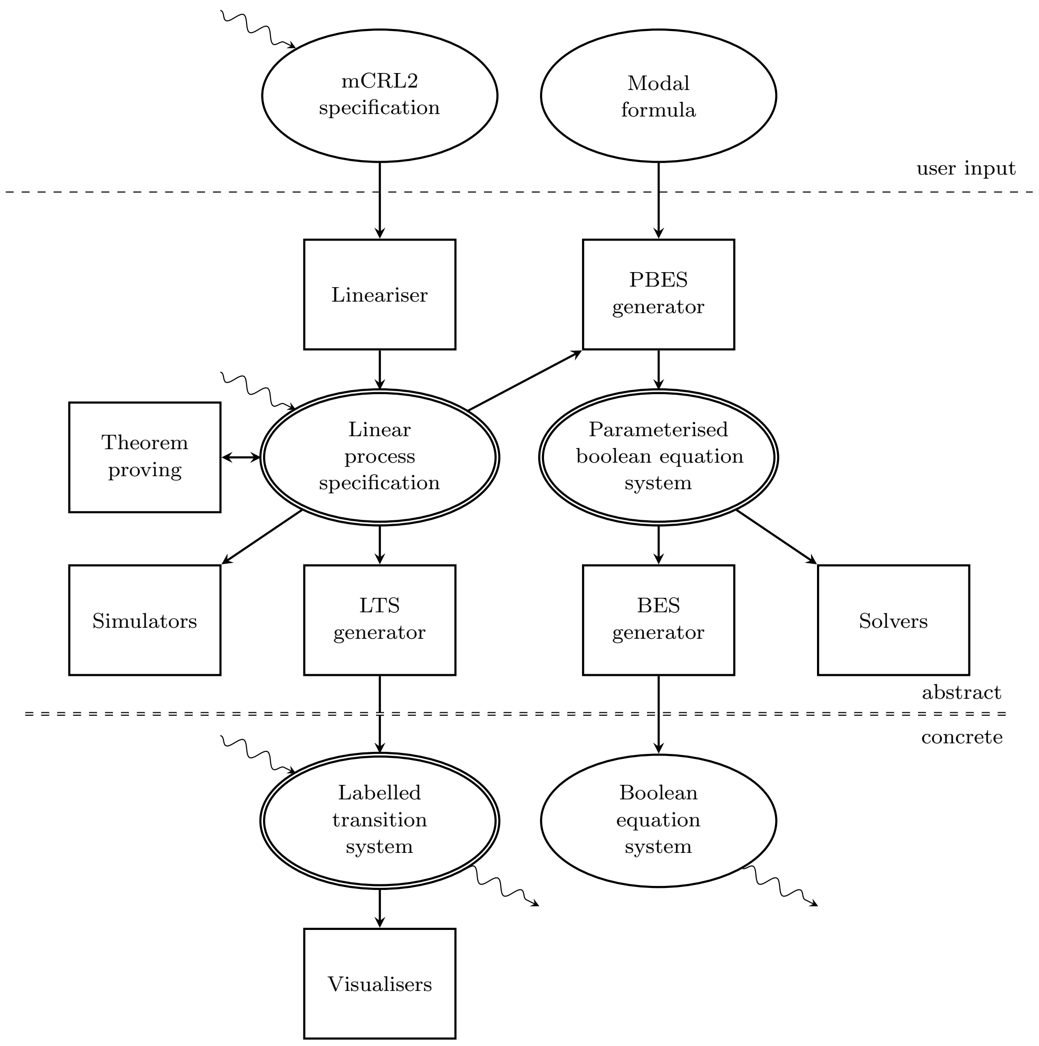

Figure 1. The structure of the mCRL2 toolset. Rectangles represent the tools, and ovals represent the objects that are manipulated by the tools. Doubly lined ovals correspond to tools that manipulate the objects. Curved arrows indicate export and import facilities to external tools.

options to continue the analysis. Validation of LPSs is supported by *LPS simulation* tools. Using simulation one can quickly gain insight into the behaviour of a system. For instance it is possible to manually select transitions, but traces can also be generated automatically (and subsequently inspected), using breadth-first, depth-first or random exploration.

Verification is supported using *theorem proving* and *model checking*. With theorem proving technology it is possible to check invariants on an LPS. Validity of boolean data expressions can also be checked using external SAT solvers. Furthermore, it is possible to detect confluence [13], which can subsequently be used to simplify the generation of *labelled transition systems* (LTSs), which are explicit representations of the state spaces.

To apply model checking, a modal formula must be provided that states some functional requirement on the mCRL2 specification. Modal formulae can be specified in a variant of the modal μ-calculus extended with regular expressions [14], data and time (see Section 4.1 for examples). In combination with the LPS this formula is transformed into a parameterised boolean equation system (PBES) [14,15] by the *PBES generator*; solving this PBES answers the encoded problem. Several tools are available for solving PBESs.

Verification and validation can be proceeded at a more concrete level by mapping LPSs to LTSs, and PBESs to boolean equation systems (BESs). This is achieved by dedicated generators. The resulting LTSs and BESs can be further manipulated by mCRL2 tools, or serve as input to external toolsets.

Verification at the level of LTSs is possible by means of equivalence checks between two LTSs using equivalences such as strong and branching bisimulation. Furthermore the presence or absence of deadlocks or certain actions can be checked, together with witnessing traces if desired.

Validation of LTSs is supported by the *LTS visualisation tools*. With such tools one can gain insight into systems up to millions of states large. Each tool has a different way of visualising an LTS: either by using automatic positioning algorithms or by clustering states based on state information. These visualisation tools have proven to be useful in detecting complex properties such as symmetry and invariance. Figure 2 shows a visualisation of an LTS using a technique from [16]. Based on their distance from the initial state, states are clustered and positioned in a 3D structure similar to a cone tree [17], with the emphasis on symmetry.

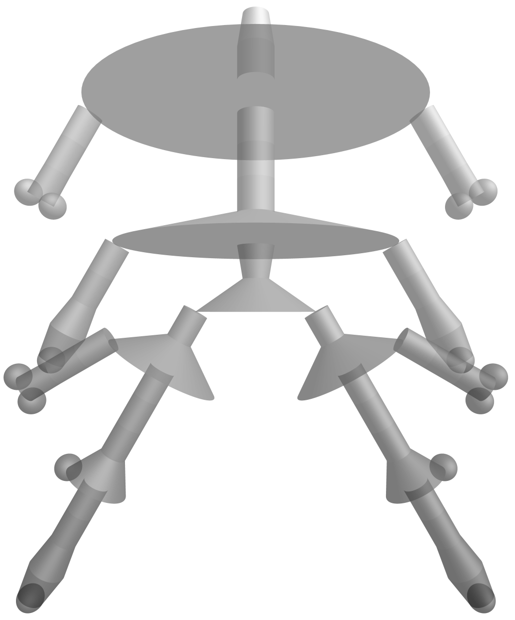

Figure 2. Visualisation of the structure of an LTSs of the alternating bit protocol [18,19]. This instance of the protocol is capable of transferring up to two different data elements. The structure is completely symmetric: the left and right part precisely correspond to the transfer of the two different data elements.

To make verification and validation feasible often reductions and simplifications must be applied. There are several LPS, PBES and LTS manipulation tools available for this purpose. For instance LPSs and PBESs can both be simplified by removing unused or constant parameters, and LPSs can be simplified by removing or instantiating summation variables [20], renaming actions, or removing time. LTS manipulation consists of minimisation with respect to various equivalences (e.g. trace equivalence, and strong and branching bisimilarity) and the conversion to different formats.

3. Design of the toolset

The mCRL2 toolset is developed in the C++ programming language. In mCRL2 the functionality is put into libraries with well-defined interfaces, which enables a high degree of code reuse. This is a departure from the tool-centered design of μCRL, the predecessor of mCRL2. The mCRL2 tools are a thin user interface layer on top of functionality provided by libraries. There are four main libraries:

– the Data library, containing functionality related to the data language like a parser, a type-checker and a rewriter
– the LPS library, containing algorithms for transforming linear processes and for the generation of state spaces
– the PBES library, containing algorithms for computing and solving PBES equations
– and the LTS library, containing functionality like property preserving reductions and equivalence checking.

Primary motivations for the choice of C++ were advanced language facilities for creating library interfaces, the availability of the C++ Standard Library, and facilities for generic programming.

Generic programming not only reduces the duplication of code, it also makes it easier to adapt algorithms. For example, a common operation is to determine equality between two data expressions. This is usually done using a rewriter, but sometimes a dedicated prover can give better results. Such operations are best implemented using a template argument, thus abstracting from the actual implementation. As a side effect, this increases the reusability of algorithms even outside the toolset.

Apart from the C++ Standard Library the mCRL2 toolset relies heavily on the ATerm Library [21] and on the Boost C++ libraries [8]. The ATerm Library (short for Annotated Terms Library), provides "an abstract data type designed for the exchange of tree-like data structures between distributed applications" [22]. An important property of the ATerm Library is maximal sharing of subterms, which allows memory-efficient storage of large terms. This is essential for the toolset, since memory is often the limiting factor when dealing with large state spaces. Boost is a diverse collection of peer-reviewed C++ libraries, some of which have been adopted in the proposal for the next generation of the C++ Standard Library.

The shift in focus towards the development of libraries instead of tools prompted for a change in working procedures. Following the Boost model, library design and public interfaces are required to be properly documented. Changes to library interfaces and associated documentation are peer-reviewed. The Boost testing framework has been adopted for testing library functionality with different platforms and compilers.

4. Applications

The mCRL2 toolset has been used in a number of academic and industrial case studies. We give an example of a simple academic case study, and give an overview of some industrial case studies.

4.1. *Academic case study: dining philosophers*

A classic example of a concurrent system is the *dining philosophers problem* [23]. It tells the story of a group of philosophers sitting at a table at which each philosopher has its own plate. In between every pair of neighbouring plates there is precisely one fork. The dish they are served requires two forks to be eaten. In other words, each pair of neighbouring entities (philosophers) share one resource (fork). For simplicity we only consider three philosophers. This situation is depicted in Fig. 3.

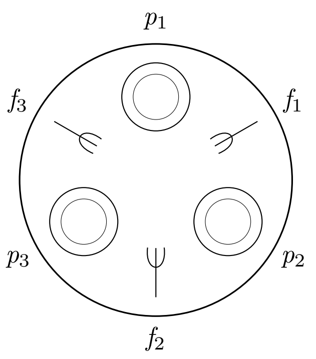

Figure 3. Dining table for three philosophers.

An mCRL2 model of the problem is presented below.

$$\begin{array}{ll}
\textbf{sort} & \mathit{PhilId} = \textbf{struct}\ p_1 \mid p_2 \mid p_3; \\
 & \mathit{ForkId} = \textbf{struct}\ f_1 \mid f_2 \mid f_3; \\[1ex]
\textbf{map} & \mathit{lf}, \mathit{rf} : \mathit{PhilId} \to \mathit{ForkId}; \\
\textbf{eqn} & \mathit{lf}(p_1) = f_1;\ \mathit{lf}(p_2) = f_2;\ \mathit{lf}(p_3) = f_3; \\
 & \mathit{rf}(p_1) = f_3;\ \mathit{rf}(p_2) = f_1;\ \mathit{rf}(p_3) = f_2; \\[1ex]
\textbf{act} & \mathsf{get}, \mathsf{put}, \mathsf{up}, \mathsf{down}, \mathsf{lock}, \mathsf{free} : \mathit{PhilId} \times \mathit{ForkId}; \\
 & \mathsf{eat} : \mathit{PhilId};
\end{array}$$

$$\begin{array}{l}
\textbf{proc}\ \mathsf{Phil}(p : \mathit{PhilId}) = \ (\mathsf{get}(p, \mathit{lf}(p)) \cdot \mathsf{get}(p, \mathit{rf}(p)) + \mathsf{get}(p, \mathit{rf}(p)) \cdot \mathsf{get}(p, \mathit{lf}(p))) \cdot \mathsf{eat}(p) \cdot \\
\qquad\qquad\qquad\qquad (\mathsf{put}(p, \mathit{lf}(p)) \cdot \mathsf{put}(p, \mathit{rf}(p)) + \mathsf{put}(p, \mathit{rf}(p)) \cdot \mathsf{put}(p, \mathit{lf}(p))) \cdot \mathsf{Phil}(p); \\
\qquad \mathsf{Fork}(f : \mathit{ForkId}) = \mathsf{sum}\, p : \mathit{Phil}.\mathsf{up}(p, f) \cdot \mathsf{down}(p, f) \cdot \mathsf{Fork}(f);
\end{array}$$

$$\begin{array}{ll}
\textbf{init} & \mathsf{allow}(\{\mathsf{lock}, \mathsf{free}, \mathsf{eat}\}, \mathsf{comm}(\{\mathsf{get} \mid \mathsf{up} \to \mathsf{lock}, \mathsf{put} \mid \mathsf{down} \to \mathsf{free}\}, \\
 & \qquad \mathsf{Phil}(p_1) \parallel \mathsf{Phil}(p_2) \parallel \mathsf{Phil}(p_3) \parallel \mathsf{Fork}(f_1) \parallel \mathsf{Fork}(f_2) \parallel \mathsf{Fork}(f_3)));
\end{array}$$

In this model the sort *PhilId* contains the philosophers. The nth philosopher is denoted by p_n. Similarly, the sort *ForkId* contains the forks (f_n denotes the nth fork). The functions *lf* and *rf* designate the left and right fork of each philosopher, respectively.

Actions $\mathsf{get}(p_n,f_m)$ and $\mathsf{put}(p_n,f_m)$ are performed by the philosopher process. They model that the philosopher p_n gets or puts down fork f_m. The corresponding actions $\mathsf{up}(p_n,f_m)$ and $\mathsf{down}(p_n,f_m)$ are performed by the fork process. They model the fork f_m being taken or put down by philosopher p_n. The action $\mathsf{eat}(p_n)$ models philosopher p_n eating. For communication purposes we have added actions lock and free. These will represent a fork actually being taken, respectively put down by a philosopher.

The process $\mathsf{Phil}(p_n)$ models the behaviour of the nth philosopher. It first takes the left- and right-hand forks (in any order), then eats, then puts both forks back (again in any order), after which it repeats this behaviour.

The process $\mathsf{Fork}(f_n)$ models the behaviour of the nth fork (i.e. the fork on the left of the nth philosopher). It can be taken by any philosopher p, after which it is put down again by the same philosopher and then repeats this behaviour.

The whole system consists of three Phil and three Fork processes in parallel. By enforcing communication between actions get and up and between put and down, we ensure that forks agree on being taken or put down by the philosophers. The result of these communications are lock and free, respectively. Note that the comm operator only ensures that communication happens when possible. The allow operator makes sure that nothing else happens by blocking all actions other than lock, free or eat.

We are looking for a solution to this problem that adheres to the following requirements, expressed in the modal μ-calculus:
– deadlock freedom: $\mathsf{nu}\, X.[\mathit{true}]X\ \&\&\ \langle \mathit{true}\rangle \mathit{true}$;
– starvation freedom: $\mathsf{nu}\, X.[\mathit{true}]X\ \&\&\ \mathsf{forall}\, p : \mathit{PhilId}.\ \mathsf{mu}\, Y.([!\mathit{eat}(p)]Y\ \&\&\ \langle \mathit{true}\rangle \mathit{true})$.
The first formula expresses that in any state it should be possible to perform an action. The second formula expresses that in any state it should be possible to eventually perform an $\mathit{eat}(p)$ action, for any philosopher p.

We check the properties on the model by first linearising the model to an LPS. For each formula, we then generate a PBES from the LPS and the formula. When solving the PBESs, we find out that they are not valid. Alternatively, we check for deadlock freedom (the first requirement) while generating the LTS from the LPS. This shows us that the trace $\mathsf{lock}(p_1,f_1)\cdot\mathsf{lock}(p_2,f_2)\cdot\mathsf{lock}(p_3,f_3)$ leads to a deadlock. It represents the situation in which each of the philosophers has taken one fork and waits for the other one, without being able to put the first one down. This also shows that the second requirement is not met: no philosopher will eventually be able to eat and they will all starve.

A standard solution to the dining philosophers problem is to use scheduling, by introducing a waiter from which the philosophers must ask permission to pick up the forks. One way of modelling a waiter is as follows:

```
act  ack_get;
proc Waiter = ack_get(p1, lf(p1)) · ack_get(p1, rf(p1))·
              ack_get(p2, lf(p2)) · ack_get(p2, rf(p2))·
              ack_get(p3, lf(p3)) · ack_get(p3, rf(p3)) · Waiter;
```

The Waiter process is put in parallel with the other processes; the ack_get action is synchronised with the communication of the get and up action to enforce the schedule:

$$\textbf{init}\quad \mathsf{allow}(\{\mathsf{lock}, \mathsf{free}, \mathsf{eat}\}, \mathsf{comm}(\{\mathsf{ack\_get} \mid \mathsf{get} \mid \mathsf{up} \rightarrow \mathsf{lock}, \mathsf{put} \mid \mathsf{down} \rightarrow \mathsf{free}\},$$
$$\mathsf{Waiter} \parallel \mathsf{Phil}(p_1) \parallel \mathsf{Phil}(p_2) \parallel \mathsf{Phil}(p_3) \parallel \mathsf{Fork}(f_1) \parallel \mathsf{Fork}(f_2) \parallel \mathsf{Fork}(f_3)));$$

Using the same procedure as before, we check the requirements on the updated model and observe that they are now satisfied.

4.2. *Industrial case studies*

For a comprehensive list of industrial case studies using the mCRL2 toolset we refer to the mCRL2 website [9]. Below we present a description of some of them.

4.2.1. *Automated Parking Garage*

Parking garages that stow and retrieve cars automatically are becoming viable solutions for parking shortages. However, these are complex systems and a number of severe incidents involving such garages have been reported. Many of these are related to safety issues in software.

In the Automated Parking Garage case study we have applied verification techniques to the development of a software design for an automated parking garage. In order to focus on the safety aspect, we have split the design of the software into three separate layers: an algorithmic layer and a hardware layer with a safety layer in between.

The safety layer has been formally specified, and subsequently validated and verified. A custom visualisation plug-in for the LPS simulator was developed to assist in the validation, and led to the discovery of some serious design flaws. Initally, verification was not feasible since the original specification would have resulted in an estimated 640 billion states. After a number of abstractions of the specification, the state space went down to 3.3 million states and 98 million transitions. On this state space we have been able to verify all safety requirements.

For more information we refer to [24]. The full mCRL2 specification is available as an appendix of [25].

4.2.2. *Aia ITP load-balancer*

Aia Software is one of the world's leading companies for software for print job distribution over document processors (high volume printers). The core of the software consists of 7.5 thousand lines of C code. In order to understand the job distribution process better, a large part of this software system has been modelled and analysed using mCRL2. Six critical issues were discovered. Since the model was close to the code, all problems that were found in the model, could be traced back to the actual code resulting in concrete suggestions for improvement of the code. All in all, the analysis significantly improved the quality of this system and led to its certification by the Laboratory for Quality Software [26].

The following concrete actions were performed in the project. The session layer of the load-balancer implementation was modelled in mCRL2 based on the C code. The underlying network socket administration layer and the upper application layers were modelled in an abstract way.

By means of state-space exploration (breadth-first search) the system was checked for deadlocks and violations of safety properties. For the configuration consisting of 3 clients and 1 server, 1.9 billion states were generated. Large experiments were performed on an 8-core AMD 64-bits machine with 128Gb RAM. For more information on this case study we refer to [27]. The full mCRL2 model is available as an appendix of [28].

4.2.3. *Pacemaker*

Vitatron (Medtronic SQDM/Vitatron in full) develops medical appliances such as pacemakers. The embedded software of a pacemaker is a complex composition of collaborating and interacting processes.

In this project the firmware design of Vitatron's DA+ pacemaker has been checked using both mCRL2 and UPPAAL. Unfortunately, UPPAAL could only be used for the initial models because it was unable to cope with the full complexity of the software.

In mCRL2, most requirements have been verified by explicit state space generation using breadth-first search. One of the requirements was validated by symbolic model checking using a PBES solver. The size of the state space depends on the configuration of the formal heart model. For the initial model of the heart, the state space contained far more than a billion states. Verifications have been carried out on restricted models with state space sizes ranging from several thousands to approximately 500 million states. The model in which we found a known violation of the requirement contained 714.464 states.

5. Conclusion

In this paper we have given an overview of the mCRL2 toolset for behavioural specifications, and some case studies.

The toolset has grown rapidly over the years. Maintainability of the software has become an important issue. To deal with this, practices from the Boost C++ Libraries have been adopted.

Quite a number of practical studies with the toolset have been done, and most of them revealed serious design flaws. In most cases existing software was modelled. In other cases the models were designs which have been transformed into working systems, of which no serious problems have been reported up till now. Expressing software models in the mCRL2 language is quite doable after some proper training. However, it still requires great skill to design models in such a way that properties about them can be efficiently checked using the toolset.

In the future we intend to further develop the system. Not only do we want to increase its general capabilities by developing improved algorithms and data structures, we are also on our way to modularise and document the toolset to open it up for other languages than mCRL2.

References

[1] H. Garavel, R. Mateescu, F. Lang, W. Serwe, CADP 2006: A toolbox for the construction and analysis of distributed processes, in: Proceedings of CAV, Vol. 4590 of LNCS, Springer, 2007, pp. 158–163.

[2] J. F. Groote, M. A. Reniers, Algebraic process verification, in: J. A. Bergstra, A. Ponse, S. A. Smolka (Eds.), Handbook of Process Algebra, Elsevier Science Publishers B.V., Amsterdam, 2001, Ch. 17, pp. 1151–1208.

[3] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[4] G. Behrmann, A. David, K. G. Larsen, A tutorial on UPPAAL, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004, No. 3185 in LNCS, Springer–Verlag, 2004, pp. 200–236.

[5] G. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2004.

[6] M. van Weerdenburg, An account of implementing applicative term rewriting, ENTCS 174 (10) (2007) 139–155.

[7] D. Kozen, Results on the propositional mu-calculus, Theor. Comput. Sci. 27 (1983) 333–354.

[8] Boost website, http://www.boost.org/.

[9] mCRL2 website, http://www.mcrl2.org/.

[10] K. Jensen, Coloured Petri Nets, EATCS Monographs on Theoretical Computer Science, Springer, 1992.

[11] D. Beek, K. Man, M. Reniers, J. Rooda, R. Schiffelers, Syntax and consistent equation semantics of hybrid χ, J. Log. Algebr. Program. 68 (1-2) (2006) 129–210.

[12] Y. S. Usenko, Linearization in μCRL, Ph.D. thesis, Technische Universiteit Eindhoven (TU/e) (Dec. 2002).

[13] J. F. Groote, M. P. A. Sellink, Confluence for process verification, Theoretical Computer Science 170 (1-2) (1996) 47–81.

[14] J. F. Groote, R. Mateescu, Verification of temporal properties of processes in a setting with data, in: A. M. Haeberer (Ed.), Proc. Algebraic Methodology And Software Technology (AMAST 1998), Vol. 1548 of LNCS, Springer, 1999, pp. 74–90.

[15] J. F. Groote, T. A. C. Willemse, Parameterised boolean equation systems (extended abstract), in: P. Gardner, N. Yoshida (Eds.), Proc. CONCUR 2004, Vol. 3170 of LNCS, Springer, 2004, pp. 308–324.

[16] F. van Ham, H. van de Wetering, J. J. van Wijk, Interactive visualization of state transition systems, IEEE Transactions on Visualization and Computer Graphics 8 (4) (2002) 319–329.

[17] G. G. Robertson, J. D. Mackinlay, S. K. Card, Cone trees: animated 3D visualizations of hierarchical information, in: CHI '91: Proceedings of the SIGCHI conference on human factors in computing systems, ACM, New York, USA, 1991, pp. 189–194.

[18] K. A. Bartlett, R. A. Scantlebury, P. T. Wilkinson, A note on reliable full-duplex transmission over half-duplex links, Communications of the ACM 12 (5) (1969) 260–261.

[19] W. C. Lynch, Reliable full duplex file transmission over half-duplex telephone lines, Communications of the ACM 11 (6) (1968) 407–410.

[20] J. F. Groote, B. Lisser, Computer assisted manipulation of algebraic process specifications, SIGPLAN Notices 37 (12) (2002) 98–107.

[21] M. G. J. van den Brand, H. A. de Jong, P. Klint, P. A. Olivier, Efficient annotated terms, Software Practice and Experience 30 (3) (2000) 259–291.

[22] Annotated terms (ATerms), Journal of Logic and Algebraic Programming 59 (2004) 1–4.

[23] E. W. Dijkstra, Hierarchical ordering of sequential processes, Acta Informatica 1 (1971) 115–138.

[24] A. Mathijssen, A. J. Pretorius, Verified design of an automated parking garage, in: L. Brim, B. R. Haverkort, M. Leucker, J. van de Pol (Eds.), Proc. FMICS and PDMC 2006, Vol. 4346 of LNCS, Springer, 2007, pp. 165–180.

[25] A. H. J. Mathijssen, A. J. Pretorius, Specification, analysis and verification of an automated parking garage, Tech. Rep. 0525, Technische Universiteit Eindhoven (2005).

[26] P. M. Heck, M. C. van Eekelen, Laquso software product certification model: (LSPCM), Tech. Rep. 0803, Technische Universiteit Eindhoven (2008).

[27] M. van Eekelen, S. ten Hoedt, R. Schreurs, Y. S. Usenko, Analysis of a session-layer protocol in mCRL2. Verification of a real-life industrial implementation, in: P. Merino, S. Leue (Eds.), Proc. FMICS 2007, Vol. 4916 of LNCS, Springer, 2008, pp. 182–199.

[28] M. van Eekelen, S. ten Hoedt, R. Schreurs, Y. S. Usenko, Analysis of a session-layer protocol in mCRL2. Verification of a real-life industrial implementation, Tech. Rep. ICIS-R07014, Radboud Universiteit Nijmegen (2007).