An Overview of the mCRL2 Toolset and its Recent Advances

S. Cranen[1], J.F. Groote[1], J.J.A. Keiren[1], F.P.M. Stappers[1], E.P. de Vink[1,2], J.W. Wesselink[1], and T.A.C. Willemse[1]

[1] Eindhoven University of Technology
[2] Centrum Wiskunde & Informatica

Abstract. The analysis of complex distributed systems requires dedicated software tools. The mCRL2 language and toolset have been developed to support such analysis. We highlight changes and improvements made to the toolset in recent years. On the one hand, these affect the scope of application, which has been broadened with extended support for data structures like infinite sets and functions. On the other hand, considerable progress has been made regarding the performance of our tools for state space generation and model checking, due to improvements in symbolic reduction techniques and due to a shift towards parity game-based solving. We also discuss the software architecture of the toolset, which was well suited to accommodate the above changes, and we address a number of case studies to illustrate the approach.

1 Introduction

Distributed systems and parallel programs are becoming increasingly common as a result of easy access to cheap multi-core processors and the popularity of paradigms such as cloud computing. These systems are notoriously difficult to design correctly. To a large extent this is caused by the concurrency that results in a lack of insight in the global configuration of a system, and the sheer number of different configurations in which a system can be at any moment. Design flaws may result in loss of data or hanging software. Race conditions are a well-known example of such flaws. While an occasional hiccup may be tolerable for non-critical applications, this may be unacceptable if an application causes significant financial losses or increases safety risks.

The mCRL2 toolset is designed to reason about distributed and concurrent systems. mCRL2 is based on the process algebra μCRL [7] and inherits its axiomatic view on processes. In μCRL, various methodologies for manually proving correctness of processes based on axiomatic reasoning were developed; these were adopted in mCRL2. The mCRL2 language, like its predecessor, is designed in such a way that it does not restrict the expressive freedom of the user. The data theory is still rooted in the theory of ADTs, but now comes with many built-in data types. Compared to μCRL, the process language has changed slightly but crucially, so semantics can be provided to languages with a notion of *true concurrency*.

The introduction of *parameterised boolean equation systems* [23] in the mCRL2 toolset clearly marks the transition to a verification paradigm based on model checking. The model checking approach complements the axiomatic verification methodology offered in the toolset. Currently, the mCRL2 toolset consists of over 60 tools that together allow visualisation, simulation, minimisation and model checking of complex systems. This paper aims to offer an overview of the toolset and its usage. We highlight its conceptual and technical essentials, of which we illustrate the domain of application, emphasising on recent developments.

First, we provide a cursory overview of the mCRL2 language. We then explain the notions of *linear process* and *equation system*, which play a fundamental role in many of the algorithms implemented in the mCRL2 toolset. The most recent improvements and additions are highlighted, addressing amongst others tool performance, support for analysing real-time systems, and solving equation systems via parity games. To broaden the scope of application, mCRL2 interfaces with other specification languages. We report on initial investigations to reduce the work needed to keep these interfaces up-to-date.

As the code base of the mCRL2 toolset has expanded substantially over the last few years, maintainability has become an important aspect in the development of the toolset. We describe our efforts to reduce the amount of hand-written code, and to improve readability and documentation of our software. These and other concerns, such as interoperability, have led to architectural changes that we mention briefly.

The uses of the language and tools are sketched by summarising a selection of illustrative case studies conducted with mCRL2. We indicate where recently added techniques were instrumental for these case studies. Finally, we position our toolset in the broader context of verification tooling, and give an outlook on the challenges ahead.

Documentation, sources and binaries of the mCRL2 toolset can be downloaded from the mCRL2 website `www.mcrl2.org`. The toolset is open source; the associated boost license allows free use for any purpose. A user manual also containing a tutorial can be found in the user documentation section of the website. The tutorial introduces the reader to the basic concepts and syntax and provides guidance for the tools most commonly used. Lecture notes used for a master course at Eindhoven University of Technology and Delft University of Technology, approaching a final draft, are available from the mCRL2 website too.

2 mCRL2: approach, applications and challenges

The mCRL2 language consists of three different sublanguages: a data language, a process language, and a property language. Following the philosophy underlying mCRL2, convenience of modelling and expressiveness have been leading in the respective definitions. We briefly discuss the three sublanguages below. For an in-depth treatment of the language, we refer to the website and the publications and material mentioned there.

In mCRL2 data and transformations on data are described using abstract data types. This allows users to create their own data types by defining the appropriate constructors and by providing functions operating on the data types. The mCRL2 data language has built-in support for commonly used data types, like the booleans, natural numbers, integers and reals. The usual operations on these data structures are predefined. Complex types can be constructed using type constructors such as sets, lists, and functions over any data type. Notation for built-in data types stays close to mathematics: numbers are written as sequences of decimals, without a limit on the size of the numbers. Sets are written using set comprehension. Functions are first-class citizens, and can be used to obtain concise models. The language allows in-line lambda abstraction as well as function updates. For example, the function doubling every natural number can be defined using the lambda abstraction `lambda n:Nat.n+n`. The function that doubles every natural number, but maps the number 4 to 0 can be defined using a function update `( lambda n:Nat.n+n )[ 4->0 ]`.

The behaviour of a system is described by processes, composed from a set of user-defined actions and a set of operators on actions and processes. These operators include multi-action composition, sequential, alternative and parallel composition and abstraction operators. The language also offers primitives to model real-time systems. Processes are defined in the context of data definitions describing the data types that are used and the operations upon them. This permits the modelling of systems whose behaviour crucially depends on the data that is exchanged: actions can be parameterised by data and *if-then-else* constructs allow for specifying conditional process behaviour. The semantics of processes is defined using a structural operational semantics, which associates with every expression in the language a *labelled transition system* (LTS). Such a labelled transition system is viewed as a graph consisting of vertices and edges, where each edge is labelled with an action, which in turn can have data parameters. The information contained in vertices is represented by a process expression and a valuation of its data parameters, but is unobservable; behaviour is determined by the actions.

High-level properties can be described using an extension of Kozen's propositional modal μ-calculus. Least and greatest fixpoint operators, which may be nested arbitrarily, can be used in combination with modal operators to describe requirements of increasing complexity. In this manner it is for instance possible to specify fairness properties, thereby staying true to the design philosophy that the modeller should not be restricted in his or her expressive freedom. The property language is equipped with constructs for reasoning about timed processes. Semantically, expressions in the property language identify a set of states in a given labelled transition system (namely, those states that satisfy the property).

Although unrestrictive, the μ-calculus is an intricate formalism. Its usability is improved by providing a set of powerful, intuitive macros, inspired by the regular expressions found in PDL. In many practical situations, this eliminates the need for fixpoint operators. For instance, safety properties asserting that a system should not exhibit a sequence of actions matching the regular expres-

sion r simply becomes `[`r`]false`. The existence of such a sequence is expressed as `<`r`>true`. By mixing regular expressions and fixpoints, one can build more complex formulae that are still easy to read. For instance, the expression `nu X.<`r`>X` asserts that there is an infinite path of action sequences matching the regular expression r. If, for instance, r = `a*.b`, it says that there is a path consisting of an infinite number of `b` actions, interrupted by finite sequences of `a` actions.

The ability to use parameterised actions in the process specification language requires similar capabilities in the property language. Like processes, properties are therefore interpreted in the context of a data specification. Fixpoint variables and actions can be parameterised with data, boolean expressions may contain data variables, and universal and existential quantification over (possibly infinite) data types are allowed. Action formulae denote (potentially infinite) sets of parameterised actions. For example, one may write `true` to denote the set of all actions, or `exists n:Nat.val(n>5)&& s(n)` to denote the set of `s(n)` actions, where `n>5`. The property `[true*.exists n:Nat.val(n>5)&& s(n)]false` then expresses that such an action never occurs.

The expressiveness of the mCRL2 property language makes it well-suited for reasoning about complex distributed systems. Its expressivity is witnessed by the fact that one can easily encode the counting μ-calculus [26] in it, which is known to be strictly more expressive than the propositional μ-calculus. The incorporation of data even enables succinct transformations from popular temporal logics. In [12], we reported on a linear transformation from CTL* to our μ-calculus; the transformation of CTL* to the equational propositional modal μ-calculus is exponential [5].

The expressive power of the mCRL2 language also has serious consequences as far as automation is concerned. Heuristics are required to work around the general undecidability of the data theory. Quantifier elimination cannot simply rely on exhaustive enumeration of all elements of a data type in case the carrier of the latter is of infinite size. The ability to use unrestricted mixing of least and greatest fixpoints in the μ-calculus may lead to computationally intractable decision problems. In the past years, we have made significant improvements in the mCRL2 toolset to cope with the consequences of the expressive power of the mCRL2 language.

3 The mCRL2 toolset

The mCRL2 toolset consists of over 60 tools that together allow for analysing complex system designs formally described in the mCRL2 language. Internally, the toolset relies on two types of objects, viz. *linear processes* [21] and *parameterised boolean equation systems* [23]. The toolset offers full control over these objects, equipping users with tools to manipulate and transform them. Below, we explain these concepts in more detail, and we indicate what progress was made in recent years.

Linear Processes Any analysis on mCRL2 specifications is preceded by an automated transformation of the specification to the linear process format. Tech-

nically, a linear process is again an mCRL2 process specification adhering to a restricted grammar, which essentially is a syntactic format for the single-step transition relation that a process induces. That is, a linear process is a recursive equation, in the untimed setting, of the following form:

$$P(d{:}D) = \sum_{i \in I} \sum_{e_i:D_i} c_i(d, e_i) \;\rightarrow\; \alpha_i(d, e_i) \cdot P(f_i(d, e_i))$$

The state space is represented by variable d of sort D. In practice, this is a vector of variables of complex sorts. Each $i \in I$ describes a *condition-action-effect* expression, stating that a multi-action α_i, consisting of actions with parameters that depend on variable d and local variable e_i, can be executed, provided boolean condition c_i evaluates to true for the values for d and e_i. The result of executing this multi-action is a state transition to $f_i(d, e_i)$. The choice between the different condition-action-effect expressions from I is resolved non-deterministically. The transformation to the linear process format is based on the expansion laws of the parallel operator of the mCRL2 process specification language. User control over linear processes is one of the distinguishing advantages of the mCRL2 toolset.

Behaviour-preserving transformations on linear processes are useful for reducing their complexity by either reducing the complexity of the data types occurring in a linear process, reducing the number of data parameters of a process, or by replacing data expressions with simpler ones. In some instances these techniques even allow one to handle processes with infinite state spaces. Typical situations in which such manipulations are very effective occur when verifying data transfer protocols, where the payload of messages is not important.

More recently, an experimental tool was developed to transform linear processes with real-valued data sorts, representing infinite state spaces such as timed systems, into linear processes representing finite ones. The tool performs a form of predicate abstraction, where the predicates are limited to linear equations over the real-valued parameters of the process.

Linear processes can be simulated, and their state space can be explicitly generated and stored. State space generation from a linear process is sped up considerably by caching the evaluation of summands in the spirit of [6], and by pruning parts of the linear process that do not contribute transitions. Typically these techniques speed up state space exploration by a factor 10 to 100. Explicit state spaces can be reduced using behavioural equivalences like strong and branching bisimulation. Implementations of simulation preorders and equivalences, as well as a divergence preserving variant of branching bisimulation have also been made available. Moreover, LTSs can be analysed using a variety of advanced, interactive visualisation techniques for both small and large state spaces in 2D and 3D [24,39].

Parameterised Boolean Equation Systems (PBESs) or just equation systems, for short, are essentially systems of least and greatest fixpoint equations over predicates involving parameterised predicate variables. Typically, a single equation has the form $\mu X(d{:}D) = \varphi$ or $\nu X(d{:}D) = \varphi$. Here, X is a predicate variable,

d is a formal variable of some sort D, and φ is a predicate formula in positive form, containing boolean expressions, predicate variables, conjunctions, disjunctions and existential and universal quantifications. The μ and ν sign indicate whether, respectively, the least or largest solution for X satisfying the equation is desired. Thus, An equation system is viewed as a finite, ordered sequence of equations for distinct predicate variables.

The problem of deciding whether a given property expressed in the μ-calculus holds for a given process specification is automatically encoded in an equation system such that the property holds for the specification if and only if the solution to the equation system is true [22]. Apart from model checking problems, also the equivalence of two processes modulo a process equivalence can be decided by encoding it into an equation system, following the encoding of [9]. This transformation is interesting when comparing infinite state spaces. Comparing finite state spaces is more efficient using traditional algorithms.

We are primarily interested in the solution of a `PBES`, as it is also the answer to the encoded problem. In many cases, however, manipulations and simplifications are needed before the equation system can actually be solved within the available memory and time. In the past years, we have added new tools implementing solution-preserving manipulations. Inspired by a similar technique operating on linear processes, an algorithm has been added that removes data parameters from propositional variables if they do not affect the solution, see [34]. Other tools implement the automated detection of invariants of equation systems [35] and use these to simplify the predicates in the equations, again without affecting the solution to the encoded verification problem. The computational complexity of these techniques is low, operating at the level of the syntax, but their effects on the time needed to solve the equation systems can be tremendous. Recently, abstract interpretation technology for equation systems was added, allowing one to reduce complex, potentially infinite data types to simpler, finite data types. A recent theoretical analysis of the underlying theory [15] revealed that this technique is more powerful than model checking based on abstractions using modal transition systems, such as [38] and their generalisations using hypertransitions, see e.g. [42].

Solving a `PBES` typically proceeds by transforming it into an equation system in which all data parameters and data expressions have been eliminated [36]. Such equation systems, which are systems of fixpoint equations over propositions, are called boolean equation systems or `BES`s [31]. Solving boolean equation systems is known to be a decidable problem. The transformation process bears many similarities to the computation of a state space from a specification. An essential step in transforming equation systems to boolean equation systems is the simplification of predicates. Quantifier elimination technology is essential to make such transformations efficient. The approach taken here is that of constructor induction, as outlined in [36], which works regardless of whether data types are finite or infinite. Special rules, such as the one-point rule, help speeding up the quantifier elimination, and are often necessary to ensure termination.

An intuitive method for solving boolean equation systems is through Gauss elimination [31]. The algorithms for solving boolean equation systems that were first offered in the toolset are based on this algorithm. While, technically, Gauss elimination is independent of the alternation depth, in practice, this method scaled poorly on verification problems obtained from fairness problems, which require μ-calculus formulae of alternation depth 2 or more. We therefore exploit the tight connection between boolean equation systems and parity games [17,20]. To efficiently generate a parity game from a `PBES`, an alternative way of generating a `PBES` from an `LPS` and a μ-calculus formula was recently introduced. Several algorithms for solving parity games have been made available to users of the toolset. Most notably, implementations of the *Small Progress Measures* [27] algorithm and the *Recursive Algorithm* [50] are available. For most model checking problems, these are very competitive, even for μ-calculus formulae of alternation depth 2 and beyond. Moreover, bisimulation-inspired reductions for boolean equation systems [29] and parity games [14] have been instrumental in solving `PBES`s where more direct approaches failed.

4 Interfacing with other languages

The state space exploration facilities and model checking capabilities of the `mCRL2` toolset can be used in combination with various other specification languages.

So-called narration and annotation of security protocols can be expressed in the process algebra `LySA`, a variant of the π-calculus that uses pattern matching to deal with encrypted data, cf. [8]. Static analysis of `LySa` processes has been applied to find authenticity and authentication issues. The conversion of a `LySa` specification into `mCRL2`, which in particular reflects the treatment of data, makes it possible to do complementary behaviour-oriented analysis.

Using the channel-based coordination language `Reo`, so-called connectors can be defined to orchestrate the interaction in a component-based system or a service-oriented application [1]. A transformation of `Reo` connectors into `mCRL2` adds model checking to the extensive tool suite for `Reo`. The synchronicity of ports that is typical for `Reo` fits well with the notion of multi-action incorporated in `mCRL2` and lies at the heart of the efficiency of the transformation.

The `mCRL2` toolkit accepts a number of other languages for input. These include the Petri net mark-up language `PNML` [48], the discretely timed part of the hybrid process algebra χ [3], a subset of executable UML [32], as well as a number of domain specific languages like `SML`, a control language based on finite state machines used at CERN [18], and `TRECS`, a language that manages resource availability [33] in the wafer steppers manufactured by ASML.

Not only the many differences between these languages, but also the evolution of their syntax and their semantics makes it difficult to maintain the dedicated tools that implement the various transformations. In fact, some of the front-ends mentioned have been marked deprecated in the latest releases of the `mCRL2`

toolset. To alleviate part of the burden, we are investigating a generic method to transform external specification formalisms into mCRL2 using Plotkin's structural operational semantics (SOS) as a common representation format.

Using this method, specifications in any language with a structural operational semantics can be transformed into a linear process. This is done by transforming the SOS into an mCRL2 data specification, and the specification under study into an mCRL2 data structure, which are then embedded in a process. This results in an mCRL2 process that encodes the semantics of the specification, and that can be analysed with all the means provided by the mCRL2 toolset. In [44], the underlying algorithm is explained for rules in the De Simone format [16], which is one of the most elementary rule formats for SOS. Extensions to the rule format, e.g. to include predicates, look-aheads and negative premises, can be handled in a similar manner [43].

While the approach is promising from a maintenance point of view, the encoding described above yields models that currently require too much time to verify in practice. Further research is therefore needed to make the technique usable on a larger scale.

5 Architecture and implementation

The mCRL2 toolset is a collection of tools written in portable C++. Development started around eight years ago, and the code base has steadily grown since then. At present it has more than 200K lines of code, is open source, is supported on 32-bit and 64-bit platforms and runs on most popular operating systems, including Linux, FreeBSD, Windows and Apple Mac OS X. Over the years development and testing of the mCRL2 toolset has matured. The code has been refactored and set up as a collection of libraries with well-defined interfaces. Code has been documented, and regression and performance tests are now run on a daily basis. Recently, commercial spin-off activities based on the mCRL2 toolset have started.

The toolset accommodates two kinds of users. End-users use the toolset for verification and validation of formal models, while the toolset also serves as a vehicle for experimental research. For end-users, correctness of the code and high-performance are the most important. Experimental researchers on the other hand require a high degree of flexibility, since they frequently want to test new ideas and algorithms. Many algorithms have been (re-)written to make the code correspond closely to pseudo-code specifications of the algorithms. This greatly improves the communication between experimental researchers and developers, which is often challenging in academic environments. The pseudo-code is also instrumental in establishing correctness of the algorithms, and in localising bugs.

A number of techniques are employed to support these different kinds of usage. Generic programming is applied to improve adaptability of the code. Notably, a universal framework for traversing the tree-like data structures in mCRL2 has been developed, which lies at the heart of many algorithms in the toolset. This framework uses static polymorphism, both for efficiency reasons and to support a modular design. Code generation from concise specifications makes it

easier to incorporate changes, and increases code reuse, which in turn reduces errors. Most of the traversal framework, and many classes and their operations consist of generated code. Currently about 17% of the code is generated, and this number is expected to increase further.

The mCRL2 toolset has a highly expressive input language. Therefore, test coverage has always been a problem. Recently, random testing has been applied to increase coverage. Randomly generated PBESs have proven to be successful in discovering otherwise hard to find bugs, like subtle cases where name clashes between quantifier variables in formulas were handled incorrectly. Currently the random generation of LPSs and state spaces is under development.

In the backend, mCRL2 provides interfaces to other tools. On the one hand, standardised file formats such as Aldebaran (.aut) and Binary Coded Graphs (.bcg) are used to export labelled transition systems to other tools such as CADP [19]. In the mCRL2 toolset, stable interfaces are provided for state space exploration. These have been designed in such a way that compile and link dependencies of tools using an interface can be kept to a bare minimum, to prevent API breakage. In close collaboration with its developers a coupling has been established with LTSmin [6], that enables symbolic and parallel state space generation of LPSs. Recently, an interface has also been added that enables instantiation of equation systems into parity games using LTSmin. As a result, the parallel and symbolic exploration techniques from LTSmin can now also be used to solve PBESs.

6 Applications and case studies

The purpose of the mCRL2 toolset is twofold. On the one hand, it aims to provide a set of state-of-the-art tools for the analysis of distributed systems. On the other hand, it serves as a platform to test research ideas in practice.

Below, we briefly report on three case studies conducted using the toolset, to offer a glimpse into the application domains of mCRL2. The first case study illustrates that the recent integration with LTSmin tool can help to reduce verification times substantially. The second case study illustrates that the mCRL2 multi-action can be essential for modelling systems and that parity game reduction techniques can be crucial for conducting the verification. The third case study demonstrates that case studies can be instrumental in improving the quality of the toolset.

DIRAC: a distributed community grid solution The high-energy experiments conducted at the large hadron collider of CERN generate a massive amount of raw data. A computing grid solution called DIRAC offers users uniform and reliable access to storage and computing resources. Despite a decade of continuous investment in developing and maintaining DIRAC, parts of the system occasionally enter inconsistent states, leading to a loss of efficiency and a potential loss of data. In an effort to tackle the problem at its root, the critical DIRAC subsystems have been modelled and analysed in mCRL2 [40]. The models of the subsystems

were verified using model checking. Modal μ-calculus formulae expressing liveness and safety requirements were formalised. Typical requirements stated, for example, that jobs are always processed once submitted, and that jobs never enter an inconsistent state. Violations of these requirements revealed livelocks and race conditions, explaining phenomena observed in the actual system.

The technology enabling the verification was the symbolic exploration (using the equation system interface with `LTSmin`, see [28]) and solving of the equation systems encoding the model checking problems. This allowed for a full verification of the system in under 60 seconds on a 64 bit Intel Core Duo (1.6GHz) machine with 2 GB RAM. For comparison, the model checking problem for a single property required more than 50 hours when conducted using explicit state space generation approaches, exploring well over $1.5 \cdot 10^8$ states. Attempts to employ compositional verification, relying on equivalence reductions to minimise state spaces, failed due to the fact that the individual processes that make up the subsystems have infinite state spaces.

FlexRay is a communication protocol that was developed by a consortium of automotive companies. Its final version was published in 2012. The protocol is designed to provide a reliable, high-bandwidth communication channel between nodes, with predictable timing properties. The protocol is *time-triggered*, that is, the protocol relies on nodes (senders and receivers of messages) to have synchronised clocks, and operates by allocating bandwidth to senders based on a global, cyclic schedule. Using `mCRL2`, the `FlexRay` startup procedure, which ensures that activated nodes will find each other and will correctly initialise their local view on the global schedule, was modelled and checked for correctness [11]. The rich data language, and the modularity of the process language of `mCRL2` allowed to specify the `FlexRay` protocol closely. In the protocol, there is a notion of *macroticks*, clock ticks that are generated by one process and communicated to the other processes using events. To model the synchronisation that these macroticks induce, multi-actions were used to create a form of barrier synchronisation.

To review the robustness of the protocol, faults that might occur in the system were modelled, which could mostly be done by making small, local changes to the fault-free model. The property language of `mCRL2` showed itself conveniently expressive to define relatively complicated properties. For instance, the property that eventually all nodes in the network will keep sending messages according to their schedule was expressed as a μ-calculus formula that uses fixpoints parameterised with data variables representing sets, and user-defined functions to specify the schedule. The properties were verified by creating a `PBES`, expanding it and solving the resulting `BES`. Solving time for these (large) equation systems was reduced by interpreting the `BES` as a parity game, reducing that game using a notion of stuttering equivalence tailored to parity games, and then solving the reduced game [13].

Domain Specific Languages Domain specific languages, or `DSL`s, have become increasingly popular with high-tech industry to speed up their design and de-

velopment cycles. Although DSLs provide an easy way to design software for a specific domain, they do not guarantee correctness of the designs. Through DSLs, however, techniques from the mCRL2 toolset can be made available to industry.

If an SOS-style operational semantics is available for a DSL, the transformation technique discussed in Section 4 can be used to analyse it using mCRL2. However, many domain specific languages are still defined informally. In [45] we report on a case study of the formalisation of an industrial DSL, called TRECS. The execution semantics was implicitly defined by the implementation of the TRECS interpreter. By formalising the language, that is, by creating an SOS for its syntactic constructs and subsequent application of the semantic transformation, we were able to discover—and improve upon—sub-optimal design decisions using the mCRL2 toolset.

To further investigate the applicability of the approach, we took the formal definition of the mCRL2 language itself and encoded it into mCRL2 again by applying the same procedure [46]. The SOS consisted of 43 deduction rules and resulted in an mCRL2 specification of slightly over 1000 lines of code. Our effort revealed a number of subtle differences between the specified, intended and the implemented semantics. In particular, the definition of the mCRL2 language allows for the use of existential quantifiers within a set comprehension scheme, but this possibility was overlooked in the actual implementation of the linear specification generator. The exercise led to improvements in the toolset and the documentation of the language.

7 Related work

The mCRL2 toolset was originally based on the toolset associated with μCRL [7]. As such, a lot of the functionality of the μCRL toolset can still be found in the mCRL2 toolset.

The toolset that—in terms of functionality—most resembles the mCRL2 toolset is CADP, developed in Grenoble [19]. It uses the specification language Lotos NT, which, like the process language of mCRL2, has its roots in process algebra; it has a property language that is, like the mCRL2 property language, based on a variant of the propositional μ-calculus, and, like in the mCRL2 toolkit, verification is conducted using equation systems. Both toolsets offer the basic functionality of minimising explicit labelled transition systems and visualising these; CADP offers a slightly richer set of equivalences that can be used to reduce with, whereas mCRL2 offers more advanced interactive 2D and 3D visualisation tooling. There are a few key differences between the two toolsets. While the mCRL2 toolset is fully open source, CADP's license imposes more restrictions. Model checking in CADP is essentially limited to alternation-free μ-calculus formulae, with limited support for alternation depth 2 formulae, whereas potentially mCRL2 can verify μ-calculus formulae of arbitrary alternation depth. Unlike CADP, mCRL2 can be used to specify and analyse real-time systems. On the other hand, CADP provides features to support performance evaluation, which are lacking in mCRL2. Finally, there are differences in the philosophy between CADP and mCRL2: the latter pro-

vides full control over objects such as linear processes and equation systems, whereas in CADP objects fulfilling similar roles are hidden from the user.

Process algebras from the CSP family are less closely related to mCRL2. For example, the FDR2 toolset [41] is based on checking refinement relations such as failure-divergence inclusion between specifications and implementations. It has support for static analysis and compositional reasoning; facilities for model checking are limited to a predefined set of properties such as (the absence of) livelock, deadlock and determinism. The PAT toolset [47] provides similar features, but additionally supports specifying and analysing real-time systems and it is capable of LTL-based model checking. Furthermore, it comes with advanced techniques such as partial order reduction and symmetry reduction.

Prominent tools focussing on model checking include SPIN [25] and nuSMV [10]. The languages supported by these tools have more restricted data types (generally booleans or bits, limited range integers and finite arrays). SPIN uses a C-like process specification language Promela for the analysis of parallel programs. It primarily focusses on LTL model checking. Properties can be established by augmenting the specification with assertions and so-called 'never claims', which are either obtained from LTL formulae or constructed manually. The tool is most famous for its use of partial order reduction and bit hashing technology. The tool DiVinE is an LTL model checker built for grid and multi-core platforms [2]. It is an automaton-based tool providing a high-performance parallel computing engine. The nuSMV toolset exploits clever data structures such as BDDs to compactly represent large state spaces. Model checking in nuSMV is currently limited to CTL and LTL properties. It also offers support for bounded model checking using SAT solving.

Several toolsets are optimised for verifying specifications with predominantly quantitative aspects. These include real-time and probabilistic model checking, with tools such as Uppaal [4] and Prism [30] The tool Uppaal is based on the notion of timed automata and uses graphs to draw behaviour which can be used to describe timed behaviour. Model checking of a restricted temporal logic is solved elegantly relying on efficient representations and manipulations of time regions. The tool Prism targets discrete and continuous-time Markov chains and decision processes. It supports simulation and model checking of PCTL and CSL.

In Section 5, we already mentioned the LTSmin toolset [6] as one of the back-ends for mCRL2. Contrary to the toolsets listed above, LTSmin has no dedicated language. Instead, it provides highly optimised state space generation tools employing multi-core, parallel and symbolic reachability analysers and model checkers, and it is used as back-ends for, e.g., DiVinE, SPIN and mCRL2.

8 Closing remarks

The mCRL2 language and toolset provide end-users with state-of-the-art tools for analysing complex, distributed systems. In developing the mCRL2 toolset we aim to uphold a consistent and reliable user experience across the various supported operating systems, viz., Linux, Windows, Apple Mac OS X and FreeBSD. For

instance, we recently ported all our graphical tools from wxWidgets to Qt for this reason. On the other hand, the toolset serves as a platform for testing research ideas in practice. This requires flexible code that is easy to adapt. Some of the older parts of the toolset have not been written with adaptability in mind, making it harder to experiment with these. Efforts are being made to change this. For example, the type checker of the language is scheduled for replacement by a much more generic and modularised version. While we consider such maintenance to be necessary for the progress of the toolset, it distracts from more fundamental research.

Several challenges lie ahead. Underlying many of the algorithms for manipulating linear processes and equation systems in the `mCRL2` toolset is a rewrite engine. The rewriter enables automated reasoning about data expressions found in the linear processes and equation systems. Therefore, the efficiency of our tools depends, to a large extent, on the performance of the rewriter. Currently, we use just-in-time rewriting [37], which has been improved using strategy trees and matching trees [49]. These are in essence techniques that reduce the number of checks that have to be done in the rewrite engine. Nonetheless, the current first-order rewriter sometimes causes performance problems when dealing with more advanced language constructs such as lambda expressions, which we expect to be able to solve using a generic higher-order rewriter. Such a rewrite engine is currently under development.

At the same time, a few of our algorithms rely on a theorem prover based on binary decision diagrams with equations. It may be beneficial to use dedicated provers like SMT solvers for some problems instead. Limited support for integrating SMT solvers is already present in several experimental tools. Integrating them more robustly in the toolset and using them in more places is part of our ongoing investigations. In particular, we are investigating possible ways to connect SMT solvers with the abstraction tooling for `PBES`s [15].

References

1. F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
2. J. Barnat, L. Brim, and P. Rockai. DiVinE multi-core–a parallel LTL model-checker. In S.D. Cha et al., editor, *ATVA 2008*, pages 234–239. LNCS 5311, 2008.
3. D.A. van Beek et al. Syntax and consistent equation semantics of hybrid Chi. *Journal of Logic and Algebraic Programming*, 68(1–2):129–210, 2006.
4. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *SFM 2004*, pages 200–236. LNCS 3185, 2004.
5. G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *LICS*, pages 304–312. IEEE Computer Society, 1996.
6. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Proc. CAV 2010*, pages 354–359. LNCS 6174, 2010.
7. S. Blom et al. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *CAV 2001*, pages 250–254. LNCS 2102, 2001.

8. C. Bodei et al. Automatic validation of protocol narration. In *CSFW 2003*, pages 126–140. IEEE, 2003.
9. T. Chen et al. Equivalence checking for infinite systems using parameterized boolean equation systems. In L. Caires and V. Thudichum Vasconcelos, editors, *CONCUR 2007*, pages 120–135. LNCS 4703, 2007.
10. A. Cimatti et al. nuSMV2: An opensource tool for symbolic model checking. In E. Brinksma and K.G. Larsen, editors, *CAV 2002*, pages 359–364. 2404, 2002.
11. S. Cranen. Model checking the `FlexRay` startup phase. In M. Stoelinga and R. Pinger, editors, *Proc. FMICS 2012*, pages 131–145. LNCS 7437, 2012.
12. S. Cranen, J.F. Groote, and M.A. Reniers. A linear translation from CTL* to the first-order modal μ-calculus. *Theoretical Computer Science*, 412:3129–3139, 2011.
13. S. Cranen, J.J.A. Keiren, and T.A.C. Willemse. Stuttering mostly speeds up solving parity games. In M. Gheorghiu Bobaru et al., editor, *NFM 2011*, pages 207–221. LNCS 6617, 2011.
14. S. Cranen, J.J.A. Keiren, and T.A.C. Willemse. A cure for stuttering parity games. In A. Roychoudhury and M. D'Souza, editors, *ICTAC 2012*, pages 198–212. LNCS 7521, 2012.
15. S. Cranen et al. Abstraction in parameterised boolean equation systems. Technical Report 13-01, Eindhoven University of Technology, 2013.
16. R. De Simone. Higher-level synchronising devices in Meije-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.
17. E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy. In *FOCS 1991*, pages 368–377. IEEE, 1991.
18. B. Franek and C. Gaspar. SMI++ object-oriented framework for designing and implementing distributed control systems. *IEEE Transactions on Nuclear Science*, 52(4):891–895, 2005.
19. H. Garavel et al. CADP 2010: A toolbox for the construction and analysis of distributed processes. In P.A. Abdulla and K.R.M. Leino, editors, *TACAS 2011*, pages 372–387. LNCS 6605, 2011.
20. E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*. LNCS 2500, 2002.
21. J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1–2):39–70, 2001.
22. J.F. Groote and T.A.C. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56(3):251–273, 2005.
23. J.F. Groote and T.A.C. Willemse. Parameterised Boolean equation systems. *Theoretical Computer Science*, 343(3):332–369, 2005.
24. F. van Ham, H. van de Wetering, and J.J. van Wijk. Visualization of state transition graphs. In K. Andrews et al., editor, *INFOVIS 2001*, pages 59–63, 2001.
25. G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley, 2003.
26. D. Janin and G. Lenzi. Relating levels of the mu-calculus hierarchy and levels of the monadic hierarchy. In *LICS 2001*, pages 347–356. IEEE, 2001.
27. M. Jurdziński. Small progress measures for solving parity games. In H. Reichel and S. Tison, editors, *STACS 2000*, pages 290–301. LNCS 1770, 2000.
28. G. Kant and J. van de Pol. Efficient instantiation of parameterised boolean equation systems to parity games. In *Proc. GRAPHITE 2012*, pages 50–65. EPTCS 99, 2012.
29. J.J.A. Keiren and T.A.C. Willemse. Bisimulation minimisations for boolean equation systems. In K.S. Namjoshi et al., editor, *HVC 2009*, pages 102–116. LNCS 6405, 2011.

30. M.Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *TACAS 2002*, pages 52–66. LNCS 2280, 2002.
31. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Technische Universität München, 1997.
32. S.J. Mellor and M.J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesly, 2002.
33. N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-reactive Scheduling*. PhD thesis, Eindhoven University of Technology, 2004.
34. S.M. Orzan, J.W. Wesselink, and T.A.C. Willemse. Static analysis techniques for parameterised boolean equation systems. In S. Kowalewski and A. Philippou, editors, *TACAS 2009*, pages 230–245. LNCS 5505, 2009.
35. S.M. Orzan and T.A.C. Willemse. Invariants for parameterised Boolean equation systems. *Theoretical Computer Science*, 411(11–13):1338–1371, 2010.
36. B. Ploeger, W. Wesselink, and T.A.C. Willemse. Verification of reactive systems via instantiation of parameterised Boolean equation systems. *Information and Computation*, 209(4):637–663, 2011.
37. J. van de Pol. JITty: A rewriter with strategy annotations. In S. Tison, editor, *RTA 2002*, pages 367–370. LNCS 2378, 2002.
38. J.C. van de Pol and M.V. Espada. Modal abstractions in μCRL. In C. Rattray et al., editor, *AMAST 2004*, pages 409–425. LNCS 3116, 2004.
39. A.J. Pretorius and J.J. van Wijk. Bridging the semantic gap: Visualizing transition graphs with user-defined diagrams. *IEEE Computer Graphics and Applications*, 27:58–66, 2007.
40. D. Remenska et al. Using model checking to analyze the system behavior of the LHC production grid. In *CCGrid 2012*, pages 335–343. IEEE, 2012.
41. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
42. S. Shoham and O. Grumberg. 3-valued abstraction: more precision at less cost. *Information and Computation*, 206(11):1313–1333, 2008.
43. F.P.M. Stappers. *Bridging Formal Models: An Engineering Perspective*. PhD thesis, Eindhoven University of Technology, 2012.
44. F.P.M. Stappers, M.A. Reniers, and S. Weber. Transforming SOS specifications to linear processes. In G. Salaün and B. Schätz, editors, *Proc. FMICS 2011*, pages 196–211. LNCS 6959, 2011.
45. F.P.M. Stappers et al. Formalizing a domain specific language using SOS: An industrial case study. In A.M. Sloane and U. Aßmann, editors, *Proc. SLE 2011*, pages 223–242. LNCS 6940, 2011.
46. F.P.M. Stappers et al. Dogfooding the formal semantics of `mCRL2`. In J. Bowen and Huibiao Zhu, editors, *35th SEW*. IEEE, 2012. To appear.
47. Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. In A. Bouajjani and O. Maler, editors, *CAV 2009*, pages 709–714. LNCS 5643, 2009.
48. M. Weber and E. Kindler. The Petri net markup language. In H. Ehrig et al., editor, *APN 2003*, pages 124–144. LNCS 2472, 2003.
49. M. van de Weerdenburg. *Efficient Rewriting Techniques*. PhD thesis, Eindhoven University of Technology, 2009.
50. W. Zielonka. Infinite games on finitely coloured graphs with applications to automat a on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.