The mCRL2 Toolset for Analysing
Concurrent Systems
Improvements in Expressivity
and Usability

Olav Bunte¹, Jan Friso Groote¹(✉), Jeroen J. A. Keiren¹,²,
Maurice Laveaux¹, Thomas Neele¹, Erik P. de Vink¹, Wieger Wesselink¹,
Anton Wijs¹, and Tim A. C. Willemse¹

¹ Eindhoven University of Technology, Eindhoven, The Netherlands
j.f.groote@tue.nl
² Open University of the Netherlands, Heerlen, The Netherlands

Abstract. Reasoning about the correctness of parallel and distributed
systems requires automated tools. By now, the mCRL2 toolset and lan-
guage have been developed over a course of more than fifteen years. In
this paper, we report on the progress and advancements over the past six
years. Firstly, the mCRL2 language has been extended to support the
modelling of probabilistic behaviour. Furthermore, the usability has been
improved with the addition of refinement checking, counterexample gen-
eration and a user-friendly GUI. Finally, several performance improve-
ments have been made in the treatment of behavioural equivalences.
Besides the changes to the toolset itself, we cover recent applications
of mCRL2 in software product line engineering and the use of domain
specific languages (DSLs).

1 Introduction

Parallel programs and distributed systems become increasingly common. This is
driven by the fact that Demard’s scaling theory [17], stating that every new pro-
cessor core is expected to provide a performance gain over older cores, does not
hold any more, and instead performance is to be gained from exploiting multiple
cores. Consequently, distributed system paradigms such as cloud computing have
grown popular. However, designing parallel and distributed systems correctly is
notoriously difficult. Unfortunately, it is all too common to observe flaws such as
data loss and hanging systems. Although these may be acceptable for many
non-critical applications, the occasional hiccup may be impermissible for critical
applications, e.g., when giving rise to increased safety risks or financial loss.

The mCRL2 toolset is designed to reason about concurrent and distributed
systems. Its language [27] is based on a rich, ACP-style process algebra and has
an axiomatic view on processes. The data theory is rooted in the theory of
abstract data types (ADTs). The toolset consists of over sixty tools supporting
visualisation, simulation, minimisation and model checking of complex systems.

© The Author(s) 2019
hhttps://doi.org/10.1007/978-3-030-17465-1_2
In this paper, we present an overview of the mCRL2 toolset in general, focussing on the developments from the past six years. We first present a cursory overview of the mCRL2 language, and discuss the recent addition of support for modelling and analysing *probabilistic processes*.

*Behavioural equivalences* such as strong and branching bisimulation are used to reduce and compare state spaces of complex systems. Recently, the complexity of branching bisimulation has been significantly improved from $O(mn)$ to $O(m(\log |\text{Act}| + \log n))$, where $m$ is the number of transitions, $n$ the number of states, and $\text{Act}$ the set of actions. This was achieved by implementing the new algorithm by Groote *et al.* [24]. Additionally, support for checking (weak) failures refinement and failures divergence refinement has been added.

*Model checking* in mCRL2 is based on parameterised boolean equation systems (PBES) [33] that combine information from a given mCRL2 specification and a property in the modal $\mu$-calculus. Solving the PBES answers the encoded model checking problem. Recent developments include improved static analysis of PBESs using liveness analysis, and solving PBESs for infinite-state systems using symbolic quotienting algorithms and abstraction. One of the major features recently introduced is the ability to generate comprehensive counterexamples in the form of a subgraph of the original system.

To aid novice users of mCRL2, an alternative graphical user-interface (GUI), *mcr2ide*, has been added, that contains a text editor to create mCRL2 specifications, and provides access to the core functionality of mCRL2 without requiring the user to know the interface of each of the sixty tools. The use of the language and tools is illustrated by means of a selection of case studies conducted with mCRL2. We focus on the application of the tools as a verification back-end for domain specific languages (DSLs), and the verification of software product lines.

The mCRL2 toolset can be downloaded from the website [www.mcrl2.org](http://www.mcrl2.org). This includes binaries as well as source code packages\(^1\). To promote external contributions, the source code of mCRL2 and the corresponding issue tracker have been moved to GitHub.\(^2\) The mCRL2 toolset is open source under the permissive Boost license, that allows free use for any purpose. Technical documentation and a user manual of the mCRL2 toolset, including a tutorial, can be found on the website. An extensive introduction to the mCRL2 language can be found in the textbook *Modeling and analysis of communicating systems* [27].

The rest of the paper is structured as follows. Section 2 introduces the basics of the mCRL2 language and Sect. 3 its probabilistic extension. In Sect. 4, we discuss several new and improved tools for various behavioural relations. Section 5 gives an overview of novel analysis techniques for PBESs, while Sect. 6 introduces mCRL2’s improved GUI and Sect. 7 discusses a number of applications. Related work is discussed in Sects. 8 and 9 presents a conclusion and future plans.

---

\(^1\) The source code is also archived on [https://doi.org/10.5281/zenodo.2555054](https://doi.org/10.5281/zenodo.2555054).

\(^2\) [https://github.com/mCRL2org/mCRL2](https://github.com/mCRL2org/mCRL2).
2 The mCRL2 Language and Workflow

The behavioural specification language mCRL2 [27] is the successor of \( \mu \text{CRL} \) (micro Common Representation Language [28]) that was in turn a response to a language called CRL (Common Representation Language) that became so complex that it would not serve a useful purpose.

```plaintext
sort Content = struct bad_data | data1 | data2;

act read, deliver, get, put, pass_on : Content;

proc Filter =
    \[\sum_{c : Content} \text{get}(c) \cdot (c \approx \text{bad_data} \rightarrow \text{Filter} \circ \text{put}(c) \cdot \text{Filter})\];
    \[
    \text{Queue}(q : \text{List}(\text{Content})) = \sum_{c : \text{Content}} \text{read}(c) \cdot \text{Queue}(c \triangleright q) + \]
    \[
    q \neq [\ ] \rightarrow \text{deliver}(\text{rhead}(q)) \cdot \text{Queue}(\text{rtail}(q));
    \]

init \( \nabla \{\text{get, deliver, pass_on}\} \{\Gamma \{\text{put} \mid \text{read} \rightarrow \text{pass_on}\} \{\text{Filter} \mid \text{Queue}([])\}\} \);
```

Fig. 1. A filter process communicating with an infinite queue in mCRL2.

The languages \( \mu \text{CRL} \) and mCRL2 are quite similar combinations of process algebra in the style of ACP [8] together with equational abstract data types [19]. A typical example illustrating most of the language features of mCRL2 is given in Fig. 1, which shows a filter process (Filter) that iteratively reads data via an action get and forwards it to a queue using the action put if the data is not bad. The queue (Queue) is infinitely sized, reading data via the action read and delivering data via the action deliver. The processes are put in parallel using the parallel operator \( \parallel \). The actions put and read are forced to synchronise into the action pass_on using the communication operator \( \Gamma \) and the allow operator \( \nabla \).

The language mCRL2 only contains a minimal set of primitives to express behaviour, but this set is well chosen such that behaviour of communicating systems can be easily expressed. Both \( \mu \text{CRL} \) and mCRL2 allow to express systems with time, using positive real time tags to indicate when an action takes place. Recently the possibility has been added to express probabilistic behaviour in mCRL2, which will be explained in Sect. 3.

The differences between \( \mu \text{CRL} \) and mCRL2 are minor but significant. In mCRL2 the if-then-else is written as \( c \rightarrow p \circ q \) (was \( p<\triangleright \circ q \)). mCRL2 allows for multi-actions, e.g., \( a \mid b \mid c \) expresses that the actions \( a, b \) and \( c \) happen at the same time. mCRL2 does not allow multiple actions with the same time tag to happen consecutively (\( \mu \text{CRL} \) does, as do most other process specification
formalisms with time). Finally, mCRL2 has built-in standard datatypes, mechanisms to allow to specify datatypes far more compactly, and it allows for function datatypes, including lambda expressions, as well as arbitrary sets and bags.

The initial purpose of μCRL was to have a mathematical language to model realistic protocols and distributed systems of which the correctness could be proven manually using process algebraic axioms and rules, as well as the equations for the equational data types. The result of this is that mCRL2 is equipped with a nice fundamental theory as well as highly effective proof methods [29, 30], which have been used, for instance, to provide a concise, computer checked proof of the correctness of Tanenbaum’s most complex sliding window protocol [1].

When the language μCRL began to be used for specifying actual systems [20], it became obvious that such behavioural specifications are too large to analyse by hand and tools were required, a toolset was developed. It also became clear that specifications of actual systems are hard to give without flaws, and verification is needed to eliminate those flaws. In the early days verification had the form of proving that an implementation and a specification were (branching) bisimilar.

Often it is more convenient to prove properties about aspects of the behaviour. For this purpose mCRL2 was extended with a modal logic, in the form of the modal μ-calculus with data and time. A typical example of a formula in modal logic is the following:

\[ \nu X(n: \mathbb{N} = 0). \forall m: \mathbb{N}. [\text{enter}(m)] X(n+m) \land \forall m: \mathbb{N}. [\text{extract}(m)] (m \leq n \land X(n-m)) \]

which says that the amount extracted using actions extract can never exceed the cumulative amount entered via the action enter. The modal μ-calculus with data is far more expressive than languages such as LTL and CTL*, which can be mapped into it [13].

![Fig. 2. The mCRL2 model checking workflow](image)

Verification of modal formulae is performed through transformations to linear process specifications (LPSs) and parameterised boolean equation systems (PBESs) [25, 33]. See Fig. 2 for the typical model checking workflow. An LPS is a process in normal form, where all state behaviour is translated into data parameters. An LPS essentially consists of a set of condition-action-effect rules.
saying which action can be done in which state, and as such is a symbolic representation of a state space. A PBES is constructed using a modal formula and a linear process. It consists of a parameterised sequence of boolean fixed point equations. A PBES can be solved to obtain an answer to the question whether the mCRL2 specification satisfies the supplied formula. For more details on PBESs and the generation of evidence, refer to Sect. 5.

Whereas an LPS is a symbolic description of the behaviour of a system, a labelled transition system (LTS), makes this behaviour explicit. An LTS can be defined in the context of a set of action labels. The LTS itself consists of a set of states, an initial state, and a transition relation between states where each transition is labelled by an action. The mCRL2 toolset contains the lps2lts tool to obtain the LTS from a given LPS by means of state space exploration. The resulting LTS contains all reachable states of this LPS and the transition relation defining the possible actions in each state. The mCRL2 toolset provides tools for visualising and reducing LTSSs and also for comparing LTSSs in a pairwise manner. For more details on reducing and comparing LTSSs, refer to Sect. 4.

3 Probabilistic Extensions to mCRL2

A recent addition to the mCRL2 language is the possibility to specify probabilistic processes using the construct \( \text{dist } x:D[ \text{dist}(x) ].p(x) \) which behaves as the process \( p(x) \) with probability \( \text{dist}(x) \). The distribution \( \text{dist} \) may be discrete or continuous. For example, a process describing a light bulb that fails according to a negative exponential distribution of rate \( \lambda \) is described as

\[
\text{dist } r:\mathbb{R}.[ \text{if } (r \geq 0, \lambda e^{-\lambda r}, 0) ].\text{fail}^r
\]

where \( \text{fail}^r \) is the notation for the action \( \text{fail} \) that takes place at time \( r \).

The modelling of probabilistic behaviour with the probabilistic extension of mCRL2 can be rather insightful as advocated in [32]. There it is illustrated for the Monty Hall problem and the so-called “problem of the lost boarding pass” how strong probabilistic bisimulation and reduction modulo probabilistic weak trace equivalence can be applied to visualise the probabilistic LTS (PLTS) of the underlying probabilistic process as well as to establish the probability of reaching a target state (or set of states). We illustrate this by providing the description and state space of the Monty Hall problem here.

In the Monty Hall problem, there are three doors, one of which is hiding a prize. A player can select a door. Then one of the remaining doors that does not hide the prize is opened. The player can then decide to select the other door. If he does so, he will get the prize with probability \( \frac{2}{3} \). The action \text{prize(true)} indicates that a prize is won. The action \text{prize(false)} is an indication that no prize is obtained. A possible model in mCRL2 is given below. In this model the player switches doors. So, the prize is won if the initially selected door was not the door with the prize.
sort Doors = struct door_1 | door_2 | door_3;
init dist door_with_prize : Doors [1/3].
    dist initially_selected_door : Doors [1/3].
    prize(initially_selected_door ≠ door_with_prize) \cdot \delta;

The generated state space for this model is given in Fig. 3 at the left. From probabilistic mCRL2 processes probabilistic transition systems can be generated, which can be reduced modulo strong probabilistic bisimulation [26] (see the next section). The reduced transition system is provided at the right, and clearly shows that the prize is won with probability $\frac{2}{3}$.

Moreover, modal mu-calculus formulae yielding a probability, i.e. a real number, can be evaluated invoking probabilistic counterparts of the central tools in the toolset. For the Monty Hall model the modal formula $(\text{prize(true)}) \cdot \text{true}$ will evaluate to the probability $\frac{2}{3}$. The tool that verified this modal formula is presented in [10]. Although the initial results are promising, the semantic and axiomatic underpinning of the process theory for probabilities is demanding.

4 Behavioural Relations

Given two LTSs, the \texttt{ltscompare} tool can check whether they are related according to one of a number of equivalence and refinement relations. Additionally, the \texttt{ltsconvert} tool can reduce a given LTS modulo an equivalence relation. In the following subsections the recently added implementations of several equivalence and refinement relations are described.

4.1 Equivalences

The \texttt{ltscompare} tool can check simulation equivalence, and (weak) trace equivalence between LTSs. In the latest release an algorithm for checking ready simulation was implemented and integrated into the toolset [23]. Regarding bisimulations, the tool can furthermore check strong, branching and weak bisimulation
between LTSs. The latter two are sensitive to so-called internal behaviour, represented by the action $\tau$. Divergence-preserving variants of these bisimulations are supported, which take the ability to perform infinite sequences of internal behaviour into account. The above mentioned equivalences can also be used by the ltsconvert tool.

Recently, the Groote/Jansen/Keiren/Wijs algorithm (GJKW) for branching bisimulation \cite{groote2005}, with complexity $O(m(\log|Act| + \log n))$, was implemented. When tested in practice, it frequently demonstrates performance improvements by a factor of 10, and occasionally by a factor of 100 over the previous algorithm by Groote and Vaandrager \cite{groote1993}.

The improved complexity is the result of combining the process the smaller half principle \cite{groote1998} with the key observations made by Groote and Vaandrager regarding internal transitions \cite{groote1993}. GJKW uses partition refinement to identify all classes of equivalent states. Repeatedly, one class (or block) $B$ is selected to be the so-called splitter, and each block $B'$ is checked for the reachability of $B$, where internal behaviour should be skipped over. In case $B$ is reachable from some states in $B'$ but not from others, $B'$ needs to be split into two subblocks, separating the states from which $B$ can and cannot be reached. Whenever a fixed-point is reached, the obtained partition defines the equivalence relation.

GJKW applies process the smaller half in two ways. First of all, it is ensured that each time a state $s$ is part of a splitter $B$, the size of $B$, in terms of number of states, is at most half the size of the previous splitter in which $s$ resided. To do this, blocks are partitioned in constellations. A block is selected as splitter iff its size is at most half the number of states in the constellation in which it resides. When a splitter is selected, it is moved into its own, new, constellation, and when a block is split, the resulting subblocks remain in the same constellation.

Second of all, it has to be ensured that splitting a block $B'$ takes time proportional to the smallest resulting subblock. To achieve this, two state selection procedures are executed in lockstep, one identifying the states in $B'$ that can reach the splitter, and one detecting the other states. Once one of these procedures has identified all its states, those states can be split off from $B'$.

Reachability checking is performed efficiently by using the notion of bottom state \cite{groote1993}, which is a state that has no outgoing internal transitions leading to a state in the same block. It suffices to check whether any bottom state in $B'$ can reach $B$. Hence, it is crucial that for each block, the set of bottom states is maintained during execution of the algorithm.

GJKW is very complicated due to the amount of book keeping needed to achieve the complexity. Among others, a data structure by Valmari, called refinable partition \cite{valmari1990} is used, together with three copies of all transitions, structured in different ways to allow fast retrieval in the various stages of the algorithm.

Besides checking for branching bisimulation, GJKW is used as a basis for checking strong bisimulation (in which case it corresponds to the Paige-Tarjan algorithm \cite{paige1987}) and as a preprocessing step for checking weak bisimulation.

For the support of the analysis of probabilistic systems, a number of preliminary extensions have been made to the mCRL2 toolset. In particular, a new
algorithm has been added to reduce PLTSs – containing both non-deterministic and probabilistic choice [44] – modulo strong probabilistic bisimulation. This new Paige-Tarjan style algorithm, called GRV [26] and implemented in the tool ltsphism, improves upon the complexity of the best known algorithm so far by Baier et al. [2]. The GRV algorithm was inspired by work on lumping of Markov Chains by Valmari and Franceschini [47] to limit the number of times a probabilistic transition needs to be sorted. Under the assumption of a bounded fan-out for probabilistic states, the time complexity of GRV is $O(n_p \log n_a)$ with $n_p$ equal to the number of probabilistic transitions and $n_a$ being the number of non-deterministic states in a PLTS.

4.2 Refinement

In model checking there is typically a single model on which properties, defined in another language, are verified. An alternative approach that can be employed is refinement checking. Here, the correctness of the model is verified by establishing a refinement relation between an implementation LTS and a specification LTS. The chosen refinement relation must be strong enough to preserve the desired properties of the model, but also weak enough to allow many valid implementations.

For refinement relations the ltscompare tool can check the asymmetric variants of simulation, ready simulation and (weak) trace equivalence between LTSs. In the latest release, several algorithms have been added to check (weak) trace, (weak) failures and failures-divergences refinement relations based on the algorithms introduced in [48]. We remark that weak failures refinement is known as stable failures refinement in the literature. Several improvements have been made to the reference algorithms and the resulting implementation has been successfully used in practice, as described in Sect. 7.1.

The newly introduced algorithms are based on the notion of antichains. These algorithms try to find a witness to show that no refinement relation exists. The antichain data structure keeps track of the explored part of the state space and assists in pruning other parts based on an ordering. If no refinement relation exists, the tool provides a counterexample trace to a violating state. To further speed up refinement checking, the tool applies divergence-preserving branching bisimulation reduction as a preprocessing step.

5 Model Checking

Behavioural properties can be specified in a first-order extension of the modal $\mu$-calculus. The problem of deciding whether a $\mu$-calculus property holds for a given mCRL2 specification is converted to a problem of (partially) solving a PBES. Such an equation system consists of a sequence of parameterised fixpoint equations of the form $(\sigma X(d_1:D_1, \ldots, d_n:D_n) = \phi)$, where $\sigma$ is either a least ($\mu$) or greatest ($\nu$) fixpoint, $X$ is an $n$-ary typed second-order recursion
variable, each \(d_i\) is a parameter of type \(D_i\) and \(\phi\) is a predicate formula (technically, a first-order formula with second-order recursion variables). The entire translation is syntax-driven, i.e., linear in the size of the linear process specification and the property. We remark that mCRL2 also comes with tools that encode decision problems for behavioural equivalences as equation system solving problems; moreover, mCRL2 offers similar translations operating on labelled transition systems instead of linear process specifications.

5.1 Improved Static Analysis of Equation Systems

The parameters occurring in an equation system are derived from the parameters present in process specifications and first-order variables present in \(\mu\)-calculus formulae. Such parameters typically determine the set of second-order variables on which another second-order variable in an equation system depends. Most equation system solving techniques rely on explicitly computing these dependencies. Obviously, such techniques fail when the set of dependencies is infinite. Consider, for instance the equation system depicted below:

\[
\begin{align*}
\nu X(i, k:N) &= (i \neq 1 \lor X(1, k + 1)) \land \forall m: N. Y(2, k + m) \\
\mu Y(i, k:N) &= (k < 10 \lor i = 2) \land (i \neq 2 \lor Y(1, 1))
\end{align*}
\]

Observe that the solution to \(X(1, 1)\), which is \textit{true}, depends on the solution to \(X(1, 2)\), but also on the solution to \(Y(2, 1 + m)\) for all \(m\), see Fig. 4. Consequently, techniques that rely on explicitly computing the dependencies will fail to compute the solution to \(X(1, 1)\).

![Dependencies of second-order recursion variables on other second-order recursion variables in an equation system.](image)

Fig. 4. Dependencies of second-order recursion variables on other second-order recursion variables in an equation system.

Not all parameters are 'used' equally in an equation system: some parameters may only influence the truth-value of a second-order variable, whereas others may also influence whether an equation depends on second-order variables. For instance, in our example, the parameter \(i\) of \(X\) determines when there is a dependency of \(X\) on \(X\), and in the equation for \(Y\), parameter \(i\) determines when there is a dependency of \(Y\) on \(Y\). The value for parameter \(k\), however, is only of interest in the equation for \(Y\), where it immediately determines its solution when \(i \neq 2\): it will be \textit{true} when \(k < 10\) and \textit{false} otherwise. For \(i = 2\), the value of \(k\) is immaterial. As suggested by the dependency graph in Fig. 4, for \(X(1, 1)\), the
only dependency that is ultimately of consequence is the dependency on \( Y(1, 1) \), i.e., \( k = 1 \); other values for \( k \) cannot be reached.

The techniques implemented in the \texttt{pbesstategraph} tool, and which are described in [37], perform a liveness analysis for data variables, such as \( k \) in our example, and reset these values to default values when their actual value no longer matters. To this end, a static analysis determines a set of control flow parameters in an equation system. Intuitively, a control flow parameter is a parameter in an equation for which we can statically detect that it can assume only a finite number of distinct values, and that its values determine which occurrences of recursion variables in an equation are relevant. Such control flow parameters are subsequently used to approximate the dependencies of an equation system, and compute the set of data variables that are still live. As soon as a data variable switches from live to not live, it can be set to a default, pre-determined value.

In our example, parameter \( i \) in equations \( X \) and \( Y \) is a control flow parameter that can take on value 1 or 2. Based on a liveness analysis one can conclude that the second argument in both occurrences of the recursion variable \( X \) in the equation for \( X \) can be reset, leading to an equation system that has the same solution as the original one:

\[
\nu X(i, k; N) = (i \neq 1 \lor X(1, 1)) \land \forall m: N \cdot Y(2, 1) \\
\mu Y(i, k; N) = (k < 10 \lor i = 2) \land (i \neq 2 \lor Y(1, 1))
\]

Observe that there are only a finite number of dependencies in the above equation system, as the universally quantified variable \( m \) no longer induces an infinite set of dependencies. Consequently, it can be solved using techniques that rely on computing the dependencies in an equation system. The experiments in [37] show that \texttt{pbesstategraph} in general speeds up solving when it is able to reduce the underlying set of dependencies in an equation system, and when it is not able to do so, the overhead caused by the analysis is typically small.

### 5.2 Infinite-State Model Checking

Two new experimental tools, \texttt{pbeta symbolicbisim} [40] and \texttt{pbesabsinthe} [16], support model checking of infinite-state systems. These are two of the few symbolic tools in the toolset. Regular PBES solving techniques, such as those implemented in \texttt{pbessolve}, store each state explicitly, which prohibits the analysis of infinite-state systems. In \texttt{pbeta symbolicbisim}, (infinite) sets of states are represented using first-order logic expressions. Instead of straightforward exploration, it performs symbolic partition refinement based on the information about the underlying state space that is contained in the PBES. The approximation of the state space is iteratively refined, until it equals the bisimulation quotient of that state space. Moreover, since the only goal of this tool is to solve a PBES, i.e. give the answer \texttt{true} or \texttt{false}, additional abstraction techniques can be very coarse. As a result, the tool often terminates before the bisimulation quotient has been fully computed.
The second tool, \texttt{pbesabsinthe}, requires the user to specify an abstraction mapping manually. If the abstraction mapping satisfies certain criteria, it will be used to generate a finite underlying graph structure. By solving the graph structure, the tool obtains a solution to the PBES under consideration.

The theoretical foundations of \texttt{pessymbolicbisim} and \texttt{pbesabsinthe} are similar: \texttt{pessymbolicbisim} computes an abstraction based on an equivalence relation and \texttt{pbesabsinthe} works with preorder-based abstractions. Both approaches have their own strengths and weaknesses: \texttt{pbesabsinthe} requires the user to specify an abstraction manually, whereas \texttt{pessymbolicbisim} runs fully automatically. However, the analysis of \texttt{pessymbolicbisim} can be very costly for larger models. A prime application of \texttt{pessymbolicbisim} and \texttt{pbesabsinthe} is the verification of real-time systems.

5.3 Evidence Extraction

One of the major new features of the mCRL2 toolset that, until recently, was lacking is the ability to generate informative counterexamples (resp. witnesses) from a failed (resp. successful) verification. The theory of evidence generation that is implemented is based on that of [15], which explains how to extract diagnostic evidence for $\mu$-calculus formulae via the \textit{Least Fixed-Point} (LFP) logic. The diagnostic evidence that is extracted is a subgraph of the original labelled transition system that permits to reconstruct the same proof of a failing (or successful) verification. Note that since the input language for properties can encode branching-time and linear-time properties, diagnostic evidence cannot always be presented in terms of traces or lassos; for linear-time properties, however, the theory permits to generate trace- and lasso-shaped evidence.

A straightforward implementation of the ideas of [15] in the setting of equation systems is, however, hampered by the fact that the original evidence theory builds on a notion of \textit{proof graph} that is different from the one developed in [14] for equation systems. In [49], we show that these differences can be overcome by modifying the translation of the model checking problem as an equation system solving problem. This new translation is invoked by passing the flag `-c' to the tool \texttt{1ps2pbes}. The new equation system solver \texttt{pbesolve} can be directed to extract and store the diagnostic evidence from an equation system by passing the linear process specification along with this equation system; the resulting evidence, which is stored as a linear process specification, can subsequently be simulated, minimised or visualised for further inspection.

Figure 5, taken from [49], gives an impression of the shape of diagnostic evidence that can be generated using the new tooling. The labelled transition system that is depicted presents the counterexample to a formula for the CERN job storage management system [43] that states that invariantly, each task that is terminated is inevitably removed. Note that this counterexample is obtained by minimising the original 142-state large evidence produced by our tools modulo branching bisimulation.
Fig. 5. Counterexamples for the requirement that each task in a terminating state is eventually removed for the Storage Management Systems. We omitted all edge labels, and the dashed line indicates a lengthy path through a number of other states (not depicted), whereas the dotted transitions are 3D artefacts.

6 User-Friendly GUI

The techniques explained in this paper may not be easily accessible to users that are new to the mCRL2 toolset. This is because the toolset is mostly intended for scientific purposes; at least initially, not much attention had been spent on user friendliness. As the toolset started to get used in workshops and academic courses however, the need for this user friendliness increased. This gave rise to the tools mcr12 gui, a graphical alternative to the command line usage of the toolset, and mcr12xi, an editor for mCRL2 specifications. However, to use the functionality of the toolset it was still required to know about the individual tools. For instance, to visualise the state space of an mCRL2 specification, one needed to manually run the tools mcr12lps, lps2lts and ltsgraph.

As an alternative, the tool mcr12ide has been added to the mCRL2 toolset. This tool provides a graphical user interface with a text editor to create and edit mCRL2 specifications and it provides the core functionality of the toolset such as visualising the (reduced) state space and verifying properties. The tools that correspond to this functionality are abstracted away from the user; only one or a few button clicks are needed.

See Fig. 6 for an instance of mcr12ide with an open project, consisting of an mCRL2 specification and a number of properties. The UI consists of an editor for mCRL2 specifications, a toolbar at the top, a dock listing defined properties on the right and a dock with console output at the bottom. The toolbar contains buttons for creating, opening and saving a project and buttons for running tools. The properties dock allows verifying each single property on the given mCRL2 specification, editing/removing properties and showing the witness/counterexample after verification.

7 Applications

The mCRL2 toolset and its capabilities have not gone unnoticed. Over the years numerous initiatives and collaborations have sprouted to apply its functionality.

7.1 mCRL2 as a Verification Back-End

The mCRL2 toolset enjoys a sustained application in industry, often in the context of case studies carried out by MSc or PhD students. Moreover, the mCRL2
Fig. 6. An instance of mcr12ide in Windows 10 with an mCRL2 specification of the alternating bit protocol. The properties in the dock on the right are (from top to bottom) true, false and not checked yet.

toolset is increasingly used as a back-end aiming at verification of higher-level languages. Some of these applications are built on academic languages; e.g., in [22] the Algebra for Wireless Networks is translated to mCRL2, enabling the verification of protocols for Mobile Ad hoc Networks and Wireless Mesh Networks. Models written in the state-machine based Simple Language of Communicating Objects (SLCO) are translated to mCRL2 to verify shared-memory concurrent systems and reason about the sequential consistency of automatically generated multi-threaded software [42]. Others are targeting more broadly used languages; e.g., in [39], Go programs are translated to mCRL2 and the mCRL2 toolset is used for model checking Go programs.

The use of mCRL2 in industry is furthermore driven by the current Formal Model-Driven Engineering (FMDE) trend. In the FMDE paradigm, programs written in a Domain-Specific Language (DSL) are used to generate both executable code and verifiable models. A recent example is the commercial FMDE toolset Desyne developed by Verum, see [9], which uses mCRL2 to check for livelocks and deadlocks, and which relies on mCRL2’s facilities to check for refinement relations (see Sect. 4.2) to check for interface compliance. Similar languages and methodologies are under development at other companies. For instance, ASML, one of the world’s leading manufacturers of chip-making equip-
ment, is developing the Alias language, and Océ, a global leading company in
digital imaging, industrial printing and collaborative business services, is de-
veloping the OIL language. Both FMDE solutions build on mCRL2.

We believe the FMDE trend will continue in the coming years and that it
will influence the development of the toolset. For example, the use of refinement
checking in the Dezyne back-end has forced us to implement several optimisations (cf. Sect. 4.2). Furthermore, machine-generated specifications are typically
longer and more verbose than handwritten specifications. This will require a
more efficient implementation of the lineariser – as implemented in mcr122lps
– in the coming years.

7.2 Software Product Lines

A software product line (SPL) is a collection of systems, individually called pro-
ducts, sharing a common core. However, at specific points the products may show
slightly different behaviour dependent on the presence or absence of so-called
features. The overall system can be concisely represented as a featured transi-
tion system (FTS), an LTS with both actions and boolean expressions over a set
of features decorating the transitions (see [12]). If a product, given its features,
fulfils the boolean expression guarding the transition the transition may be taken
by the product. Basically, there are two ways to analyse SPLs: product-based
and family-based. In product-based analysis each product is verified separately;
in family-based model checking one seeks to verify a property for a group of
products, referred to as a family, as a whole.

Traditionally, dedicated model checkers are exploited for the verification of
SPLs. Examples of such SPL model checkers are SNIP and ProVeLlines by the
team of [12] that are derived from SPIN. However, the mCRL2 toolset as-is,
without specific modifications, has also been used to compare product-based
vs. family-based model checking [3,5,7]. For this, the extension of the modal
µ-calculus for the analysis of FTSes proposed in [4], that combines actions
and feature expressions for its modalities, was translated into the first-order
µ-calculus [25], the property language of the mCRL2 toolset. As a result, verifica-
tion of SPLs can be done using the standard workflow for mCRL2, achieving
family-based model checking without a family-based model checker [18], with
running times slightly worse than, but comparable to those of dedicated tools.

8 Related Work

Among the many model checkers available, the CADP toolset [21] is the clos-
est related to mCRL2. In CADP, specifications are written in the LOTOS NT
language, which has been derived from the E-LOTOS ISO standard. Similar to
mCRL2, CADP relies on action-based semantics, i.e., state spaces are stored as
an LTS. Furthermore, the verification engine in CADP takes a µ-calculus formula
as input and encodes it in a BES or PBES. However, CADP has limited sup-
port for µ-calculus formulae with fixpoint alternation and, unlike mCRL2, does
not support arbitrary nesting of fixpoints. Whereas the probabilistic analysis
tools for mCRL2 are still in their infancy, CADP offers more advanced analy-
sis techniques for Markovian probabilistic systems. The user-license of CADP
is restrictive: CADP is not open source and a free license is only available for
academic use.

Another toolset that is based on process algebra is PAT [45]. This toolset has
native support for the verification of real-time specifications and implements on-
the-fly reduction techniques, in particular partial-order reduction and symmetry
reduction. PAT can perform model checking of LTL properties.

The toolset LTSMIN [36] has a unique architecture in the sense that it is
language-independent. One of the supported input languages is mCRL2. Thus,
the state space of an mCRL2 specification can also be generated using LTSMIN's
high-performance multi-core and symbolic back-ends.

Well-known tools that have less in common with mCRL2 are SPIN [34],
NuSMV [11], PRISM [38] and UPPAAL [6]. Each of these tools has its own
strengths. First of all, SPIN is an explicit-state model checker that incorporates
advanced techniques to reduce the size of the state space (partial-order reduction
and symmetry reduction) or the amount of memory required (bit hashing). SPIN
supports the checking of assertions and LTL formulae. Secondly, NuSMV is a
powerful symbolic model checker that offers model checking algorithms such
as bounded model checking and counterexample guided abstraction refinement
(CEGAR). The tools PRISM and UPPAAL focus on quantitative aspects of
model checking. The main goal of PRISM is to analyse probabilistic systems,
whereas UPPAAL focusses on systems that involve real-time behaviour.

9 Conclusion

In the past six years many additions and changes have been made to the mCRL2
toolset and language to improve its expressivity, usability and performance.
Firstly, the mCRL2 language has been extended to enable modelling of prob-
abilistic behaviour. Secondly, by adding the ability to check refinement and to
do infinite-state model checking the mCRL2 toolset has become applicable in
a wider range of situations. Also, the introduction of the generation of coun-
terexamples and witnesses for model checking problems and the introduction of
an enhanced GUI has improved the experience of users of the mCRL2 toolset.
Lastly, refinements to underlying algorithms, such as those for equivalence reduc-
tions and static analyses of PBESs, have resulted in lowered running times when
applying the corresponding tools.

For the future, we aim to further strengthen several basic building blocks of
the toolset, in particular the term library and the rewriter. The term library is
responsible for storage and retrieval of terms that underlie mCRL2 data expres-
sions. The rewriter manipulates data expressions based on rewrite rules speci-
fied by the user. Currently, these two components have evolved over time but
are rather limitedly documented. It has proven to be difficult to revitalise the
current implementation or to make amendments to experiment with new ideas.
For this, one of the aims is to investigate the benefits of multi-core algorithms, expecting a subsequent speed-up for many other algorithms in the toolset.

References

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.