SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **41**:143–153
Published online 29 November 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1021

Experiences in developing the mCRL2 toolset

J. F. Groote, J. J. A. Keiren[*,†], F. P. M. Stappers, J. W. Wesselink and T. A. C. Willemse

Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

SUMMARY

This paper presents practices and experiences in developing the formal methods toolset mCRL2. Findings are presented based on years of experiences in developing tools in an academic environment. Practical problems and ways to solve them are discussed. We also present the direction that we foresee for the coming years of development in formal methods tool support. Copyright © 2010 John Wiley & Sons, Ltd.

Received 1 April 2010; Revised 1 June 2010; Accepted 4 September 2010

KEY WORDS: software modelling; software engineering; software quality; verification

INTRODUCTION

mCRL2(micro Common Representation Language 2 [1]) is a language for formalizing the behaviour of concurrent systems. The language is based on the process algebra ACP [2], and has facilities for describing data, data manipulations and real-time behaviour. In general it is used to describe communicating systems and software, but the language is also suitable for *e.g.* business processes or social networks. The language is supported with a toolset enabling simulation, visualization, behavioural reduction and verification of systems. The language and its accompanying toolset are developed in an academic research group that aims to support process algebra-based verification. One of the starting points is that mCRL2 should remain usable for verifying both academic and industrial size problems.

The experiences in using the predecessor of the mCRL2 toolset, *viz.* the μCRL toolset [3], uncovered major shortcomings. Among others, μCRL lacked built-in support for pre-defined data types and support for higher order objects, such as functions, sets and quantifiers. Motivated by these shortcomings, the development of the richer modelling language mCRL2 was initiated in 2002. A position statement for the development of mCRL2 can be found in [4]. The main goal for mCRL2 is to provide a language and toolset in which academic and industrial users can specify systems in a mathematically precise and compact way. In addition the toolset should be available on all major platforms.

As the most important paradigms are shared between mCRL2 and μCRL, and a lot of code was already available in the toolset for μCRL, it was decided to reuse the existing code. Much of the functionality could be inherited, requiring minor changes to reflect the new language features. At the time this was seen as an easy way to keep development effort to a minimum.

[*]Correspondence to: J. J. A. Keiren, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.
[†]E-mail: j.j.a.keiren@tue.nl

However, by now the code has evolved to the point where most of the original code has been replaced.

The decision to reuse code fixed important design decisions, such as the use of C code and the ATerm library [5]. Along with the inherited code, the mindset that came with it was adopted. As a result, the interfaces of our new implementations had a low level of abstraction, and the code itself was quite complex, resulting in a steep learning curve for new developers.

Developing software in an academic environment introduces a number of additional problems that are interesting in their own right. Most academics do not have software development as a main goal, therefore software development gets little attention. Adopting a structured approach to programming and documentation helps to keep in control of the project.

In this paper we elaborate on our experiences and practices we adopted in developing the mCRL2 toolset. We look at both the source code and the development process.

Outline: We first introduce the language mCRL2 and the modal μ-calculus that is supported, as well as the mCRL2 toolset. We then identify the challenges in (1) the development process and (2) the existing code base. Next the development principles and coding mechanisms that we have adopted are discussed, showing how we tackle these problems. We conclude with an outlook on the future development, and the techniques we plan to employ.

OVERVIEW OF mCRL2

*The language mCRL2*is a specification language for describing the behaviour of communicating systems. The language consists of three parts: data, processes and logic. The behavioural part of the language is based on the process algebra ACP [2], and is therefore based on the same methodology as tools, such as CADP [6], μCRL [3] and FDR2 (based on CSP [7]). Like UPPAAL [8], mCRL2 allows the user to specify real-time behaviour.

The data part of the language is based on higher order abstract equational data types. It supports universal and existential quantifiers, lambda abstraction, (unbounded) integers and rational numbers, (infinite) sets and bags, structured data types and lists. The data types are designed to closely reflect their mathematical counterpart. As a result, the language is extremely expressive, but it is also easy to specify undecidable problems.

In addition, there is a property specification language based on the modal μ-calculus [9], which has been extended to treat data and time as first class citizens. To the best of our knowledge there is no equally expressive property specification formalism, especially regarding the data part. Note that toolsets like CADP also support verification of systems using modal μ-calculus. μCRL allows the user to symbolically manipulate processes. In mCRL2 similar techniques are available. In addition mCRL2 provides a symbolic encoding of verification problems using parameterized Boolean equation systems (PBESs). Symbolic techniques for simplifying PBESs are also available, such that explicit model checking can be postponed as much as possible.

Several case studies, describing both industrial, as well as academic use cases have been carried out using mCRL2, see *e.g.* [10–12]. In recent years, several autonomous market parties in the Netherlands have started to develop and deploy formal analysis tools geared towards programming embedded systems [13]. While these companies have their proprietary specification languages, verification issues are typically delegated to academic toolsets, including FDR and mCRL2. The existence of such tools has led to the use of formal methods in some development trajectories in the industry.

The toolset mCRL2 is built to support human-guided transformations of specifications. An overview of the toolset is given in Figure 1. We briefly discuss the structure of the toolset. An mCRL2 specification can either be constructed by the user, or obtained as output of tools that read specifications in other formalisms, such as coloured Petri nets [14], χ [15], typed LySa [16] and μCRL [3].

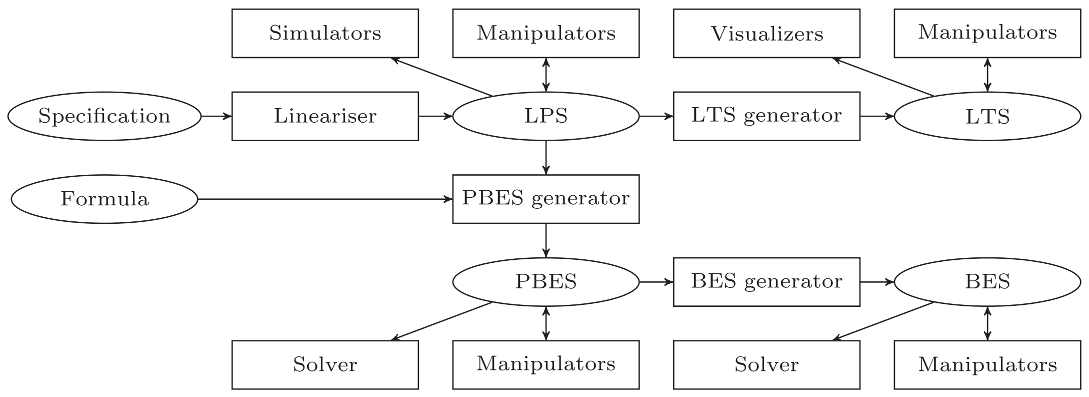

Figure 1. The structure of the mCRL2 toolset. Rectangles represent tools, ovals represent objects that can be manipulated by tools.

In general, using the linearizer, an mCRL2 specification is transformed into a linear process specification (LPS), *i.e.* a process in which all parallelism has been removed. An LPS is a compact, symbolic representation of the state space underlying the specification. Simulation tools allow the user to gain insight into the behaviour of a system described by an LPS. Linear processes can be reduced using transformations, or can be transformed into labelled transition systems (LTS), which are explicit representations of state spaces. In addition LPSs can, together with a modal μ-calculus formula expressing a functional requirement, be transformed into PBESs [17]. Solving these PBESs [18, 19] answers the encoded verification problem. Several tools for simplifying and solving PBESs are available.

Explicit state model checking is supported both through the transformation of LPSs into labelled transition systems, as well as the transformation of PBESs into Boolean equation systems (BES). The LTSs and BESs can be manipulated by other mCRL2 tools, or serve as input for external tools. Verification of LTSs is supported by reduction using behavioural equivalences, such as strong and branching bisimulation. In addition, the absence or presence of deadlock and livelock behaviour can be checked, with the ability to obtain traces to violating states. To gain insight into systems with large state spaces, several LTS visualization tools are available, supporting visual inspection of systems with millions of states, see *e.g.* [20].

In the architecture of the mCRL2 toolset, libraries are available for mCRL2 specifications, modal μ-calculus formulae, LPSs, PBESs, LTSs and BESs. By convention, the manipulations and transformations are implemented in the library corresponding to their target. As an example, linearization is implemented in the LPS library. The key algorithms are available as tools that can be used from the command line, as well as from a graphical user interface that integrates all tools.

It is our conviction that not only the tools, but also their design and implementation should be shared with the scientific community, in a similar way as scientific ideas are communicated. Therefore the toolset is provided as open source using the Boost Software Licence[‡]. This is a liberal licence that allows the free use of the software and its source code, as long as the original authors are acknowledged.

The toolset is supported on Windows, Linux and MacOSX, allowing for choice and flexibility for the user.

ANALYSIS OF THE DEVELOPMENT PROCESS

Around the year 2007 the development of mCRL2 was growing out of hand. Managing development of roughly 150 000 lines of code (LOC) (see Figure 4) required serious thought. We have been

[‡]More information about Boost can be found at http://www.boost.org.

working to improve the development process ever since. We analyse the problems that have been observed in the development process in the period from 2005 to 2010. The analysis is based on key process areas (KPAs), as can be found in process maturity standards such as the CMM standard [21]:

1. requirements management;
2. project planning;
3. project tracking;
4. software quality assurance.

Software requirements were often implicitly written down in scientific publications. Turning publications into code required substantial effort from developers. Requirements that were only uncovered during implementation were never made explicit, nor were they documented in a central place. This made it hard to track design decisions when doing maintenance.

mCRL2 development is based on a half-yearly release cycle, with releases in January and July each year. This means that development is done in short iterations. A formal planning and roadmap for the project were never made. This made it impossible to track the progress of the project towards a next release. As a result, desired features did not make it into a release where they were expected, but also features that were not expected were put into a release.

Short iterations sometimes give rise to a desire to ship experimental software. Because of this a *good*, *bad and ugly* style classification is given to tools to denote their stability and reliability. An objective way to assign the good, bad and ugly classification to tools was however never developed. It could therefore occur that tools were marked as good, when in practise they were not good at all.

Owing to the lack of a systematic approach to software testing, changes in core components of the mCRL2 toolset introduced bugs. The ‘tests’ that were carried out mainly consisted of running tools on a number of example specifications, and manually inspecting the result; unit tests *e.g.* were hardly available. Additionally, small changes have shown to cause dramatic changes in the performance of the tools. Often, bugs and performance issues only surfaced after long periods of time, making it hard to identify the root cause. This shows the need for a systematic way of improving and guarding software quality. Now that we have seen a number of KPAs, and the ways in which mCRL2 development did not pay much attention to them in 2007, we try to identify the deeper causes of these problems.

The issues raised above were in many cases symptoms of the underlying problems and the development setting. The mCRL2 toolset is developed in an academic environment, hence we can roughly distinguish three types of developers: (1) scientific programmers, (2) students (both BSc and MSc level) and (3) other staff, *i.e.* PhD students and (associate) professors. Figure 2 displays the periods in which developers have been actively contributing to the code base[§], as measured by the commits in the versioning repository. The two topmost lines on the vertical axis denote the scientific programmers. The lower 15 values on the vertical axis denote students and staff who are not normally involved in the development. The other lines denote PhD students and staff who are actively involved in the development. The figure nicely illustrates the differences in time spans and the number of commits in that time.

Scientific programmers are responsible for the majority of code in mCRL2. They have anywhere between half to all of their time available for development. Note that their main tasks are implementation and documentation; the design of new algorithms is a task for researchers. For everyone except scientific programmers, the focus is not on writing code. Instead, people focus on writing a thesis or scientific papers, and some of the code is a mere side effect of such works. As an additional complication, people not part of the permanent staff tend to come and go in quick succession, leading to *scattered knowledge* about the code base. Temporary staff has productive

[§]Observe that the figure does not indicate anything about the amount of code that has been contributed; especially authors 1 and 2 contributed only a single small bugfix.

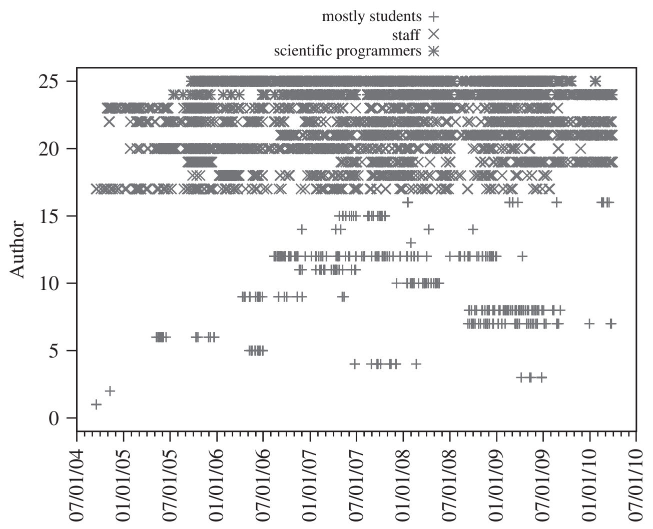

Figure 2. Developer activity over time as measured by the commits in the versioning repository.

periods of roughly 3 months (BSc students), 6 months (MSc students) and 3 years (PhD students) respectively. As time passes by, it turns out that some pieces of code *lack active developers or even maintainers*. Some components have been substantially modified by 10 or more developers in disjoint time spans because of this.

ANALYSIS OF THE CODE

Our problems were not confined to the development process, but also surfaced in our source code. We analyse our code base as it was in 2007; observe that some of the issues we describe have not yet been fully resolved. We perform our analysis using the following characteristics, which are taken from the ISO 25000 standard for the evaluation of software quality [22]:

1. *functionality*: does the software contain the required functions;
2. *efficiency*: how much resources does the software use;
3. *maintainability*: how easy is it to modify the software;
4. *portability*: how easy is it to transfer the software to another environment.

A lot of functionality in mCRL2 was initially adopted from the μCRL code base. mCRL2 itself grew organically from here. The code was unstructured, such that, while programming, it was not clear what functionality to use, or even whether certain functionality was available. The lack of documented interfaces ensured that it could not be judged whether the required functionality was available. Documentation of interfaces and algorithms was lacking, and the interfaces were inflexible.

Unstructured code was not the only factor impairing maintainability. The existing C code had a low level of abstraction (every term in the library had the same type, *viz.* `ATerm`). In addition the code suffered from code duplication and bad coding practices like unsafe type-casts. As an additional complication, it was assumed that the mCRL2 language would need no further changes, and that therefore the internal ATerm format on which the implementation was based would remain fixed.

The use of the ATerm library had serious implications for maintenance, but also portability was affected. For instance, all existing code solely compiled on Linux based systems, and used platform specific functionality. The most striking example of unportable code are the rewriters.

Example 1
To gain performance in rewriting, it is possible to generate and compile dedicated rewriters given a specification. This generation is done on the fly when running tools. Even at this time this functionality is only available on platforms that support the GNU C compiler.

The existing code increasingly impaired the day-to-day development in a number of ways. New developers suffered from a steep learning curve, and often required assistance from more experienced developers. The lack of documentation, and the incomprehensible and inflexible interfaces led developers to introduce their own *ad hoc* solutions for general problems, leading to code duplication. Finally, porting code to other platforms proved to be hard.

ADOPTED PRACTICES AND GUIDELINES

Although no decision was made to change the process and code to formally adhere to well-documented standards, such as CMM and ISO 25 000, work has been ongoing since 2007 to improve both the process and the source code. In this section we give an insight into the practices and guidelines we adopted into our software engineering process, as well as the ways in which we have changed the code. We have instated development guidelines, covering:

- coding standard;
- documentation guidelines;
- testing guidelines;
- performance monitoring.

The source code is managed using the version control software Subversion. For bug tracking purposes this is integrated with Trac[¶]. A coding standard has been constructed to improve the readability and maintainability of code by giving a uniform look and feel. It provides developers with handles for coding style, as well as requirements for standards compliance of the code. The development guidelines also describe the way in which to document and fix bugs and tasks, enabling progress tracking towards new releases. Furthermore, requirements are provided for each level in the good, bad, ugly style classification that is given to tools.

To combat the lack of documentation and traceability of design decisions, developers are required to deliver four levels of documentation:

1. tool user documentation (for users of the toolset);
2. library user documentation (for developers using the libraries);
3. pseudo-code of algorithms;
4. source code documentation.

All documentation is reviewed by at least one other developer. The tool user documentation is mainly aimed at users of the toolset, and is stored on a wiki page[‖]. It describes the theory and usage of the tools from a user perspective.

The other types of documentation are aimed at developers. The library user documentation (written in QuickBook, which is part of Boost) serves as a high level overview of a library, giving an indication of functionality in the library. The usage of a library is illustrated by small examples. Requirements and design decisions for complex algorithms are mainly documented in LaTeX, which allows for a human readable presentation, mixing text and precise mathematical notation. The algorithms are described using pseudo-code. This separates the design of the algorithm from the actual implementation. In general, this leads to cleaner algorithm descriptions, that can be readily understood and reviewed by other developers. Experience shows that bugs in implementations based on pseudo code are either bugs in the specifications, or the result of sloppiness in implementing.

[¶]See `http://subversion.tigris.org/` for information of Subversion, and `http://trac.edgewall.org/` for Trac.
[‖]Note that all online resources related to mCRL2 can be found through http://www.mcrl2.org.

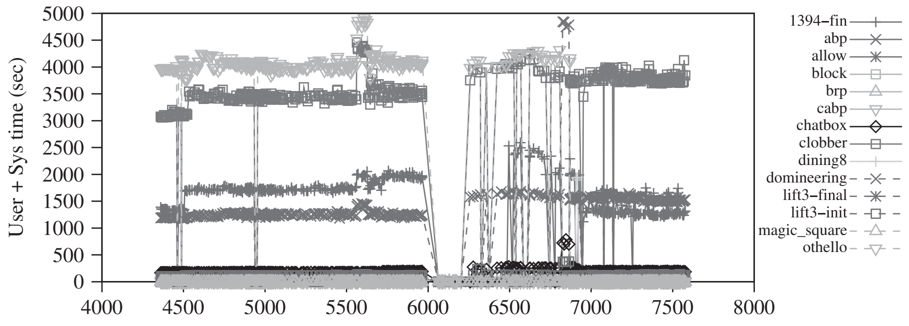

Figure 3. Historical performance measurements of the tool lps2lts with default options, each line represents an input specification, the vertical axis shows the running times, the horizontal axis the SVN revision.

Because of the availability of pseudo-code both can be easily spotted and repaired, reducing maintenance overhead. The interfaces in the implementation are documented using comments that can be processed by Doxygen**. This provides an overview that contains all functions, with a description of their purpose, as well as their arguments, and any constraints on input if applicable. All three types of library documentation are made available through a single web page, providing a single entry point for all available documentation.

Especially in the development of a formal methods toolset correctness, reliability and performance are key issues. We put the following two mechanisms in place to monitor reliability and performance:

- unit tests;
- performance measurements.

The test guidelines require developers to add unit tests for components they construct. In addition, bug fixes must be accompanied with tests that show that indeed the bug is fixed. The unit tests are implemented using the Boost Test framework, and are automatically run on a host of supported platforms on a daily basis through the CTest component of the CMake build system [23] on all supported platforms. Results are made available on a website using the CDash dashboard [23].

Because we experienced that changes in the core of the toolset sometimes had a surprising effect on the performance of individual tools, we have put automated performance measurements in place. The core tools in our toolset are automatically run on a selection of examples of varying complexity on a daily basis, using various settings for the tools. Historical running times are accumulated for comparison. All measurements (including the historical ones) are also made available through a website, enabling monitoring of trends in the performance of tools. We illustrate this with an example.

Example 2
Figure 3 shows an example of the historical performance measurements of one tool on all available examples. Tool failures, as well as cases where a predetermined time limit is exceeded are represented in the graph with -1. The figure clearly reflects issues with performance and stability between SVN revisions 6000 and 7000. It can also be observed from the figure that since revision 7000 the measurements are relatively stable, and that it is similar to the performance prior to revision 6000. Around revision 5500 there is an increase in running time caused by changes in the underlying libraries. Because of the measurements this was immediately observed and fixed.

In addition to the guidelines described so far, we have taken the following measures to improve maintainability and reliability of our code:

- switch from procedural programming in C to object-oriented programming in C++;
- use existing third-party libraries with cross-platform support.

**More information about Doxygen can be found at http://www.doxygen.org.

Switching from C to C++ as an implementation language allowed to create a framework of classes and interfaces on top of the ATerm library for the data structures and algorithms in mCRL2. The C++ classes now hide the actual ATerms, effectively turning the ATerm library into an implementation detail. This allows the compiler to perform stricter checking on types whenever possible and desirable. As a result, the developer is assisted with compile time checks, whereas earlier we could only rely on runtime checks. Also, code written on top of this framework has a higher level of abstraction and is closer to pseudo code. As a bonus the plethora of common functionality provided by the C++ standard library can be readily used.

A lot of existing code has gradually been adapted to use this framework, or, whenever necessary, rewritten from scratch. By now much of the code has been migrated, and we plan on migrating any remaining code in the next two years. In the migration process a lot of duplicate code has been removed. The most striking example of this is the new implementation of our tools. Every tool in our toolset is a class which inherits from an abstract base class that implements parsing command line arguments, printing help messages, as well as some default options. This reuse reduced the code base by thousands of LOC, and enabled a uniform treatment of command-line arguments across the tools.

By using portable external libraries, and by strictly adhering to the C++ standard in our implementation, we have succeeded in porting our toolset to various platforms, with little effort, providing native support for both 32-bit and 64-bit versions on all of these platforms. As a bonus, we gained the advantage of having checks from different compilers, which has proven to be helpful in improving the stability of the toolset. After the initial effort the overhead required for supporting multiple platforms and compilers is relatively low.

FUTURE DIRECTIONS

Currently we are aiming to move up yet another level of abstraction, and thus reduce the amount of handwritten code, and increase the flexibility of the code. We aim to allow for changes in language and internal term representation, and make our testing more rigorous. We have recently introduced the following techniques, and expect to see an increase of their use in the (near) future:

- generic programming;
- code generation;
- test generation;
- automation of non-regression tests.

Generic programming in C++ allows to create uniform interfaces with a high degree of flexibility, while retaining strict compile time checking using concepts. In an academic environment, where people require prototypes of new or altered functionality, this is a very useful property. Generic programming is especially suited for libraries that describe data types with accompanying algorithms. It allows the algorithms to abstract from the precise implementation of the data types by only assuming minimal requirements of the data types. The main advantage of using concepts in the interfaces of algorithms is the explicit documentation of type requirements. Having clear semantics of types is a necessary step towards making correct implementations. The usage of concepts promotes the writing of clean code that corresponds even closer to pseudo code descriptions of algorithms. Use of generic programming techniques increases productivity through uniform and extensible interfaces. It also allows for easier experimentation with different implementations, and leads to decreased maintenance costs through an increased level of reuse of code. We give an example of the way in which we benefit from generic programming.

Example 3
A reoccurring operation in mCRL2 algorithms is substitution on terms. A substitution concept has been defined, and a generic substitution class that models this concept. In addition functions for applying substitutions have been defined, using a generic traversal framework for mCRL2 data types. Owing to this, dozens of existing replace functions and ad hoc solutions for substitution

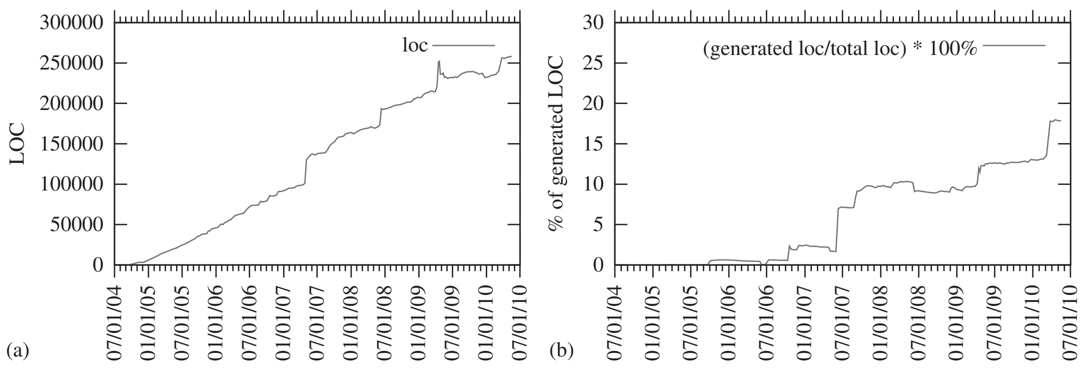

Figure 4. Lines of code in the mCRL2 toolset by date: (a) total LOC and (b) percentage of generated LOC.

problems have become obsolete. The main advantage is that substitutions now share a uniform interface, and thus are much easier to maintain and use. The shared implementation also reduces the possibility of programming errors.

In C++, generic programming is enabled by using templates. C++ concepts, a type system for templates [24], form a key ingredient of this style of programming. Using concept definitions, sets of requirements (valid expressions, associated types, semantic invariants, complexity guarantees, and so on) on template types can be defined that are type checked during compile time. Concepts are especially suitable for construction of data structures and algorithms that operate on these, see *e.g.* [25]. Language support for concepts was proposed for the next version of the C++ standard, C++0x, see *e.g.* [26], but was turned down because there were doubts about the impact on the day-to-day programming. As a result, we decided to use the portable Boost Concept library for concept checking.

Currently a new generic higher order rewriter is being developed. Owing to the preparations already taken it is expected to be very easy to integrate it once it has been finished. There are several other applications in the toolset where generic interfaces may lead to increased flexibility and reuse of code, most notably the LTS and BES libraries.

We also move towards a higher level of abstraction by generating code. In order to facilitate functionality such as term traversal in a uniform manner, we have been looking into generating this traversal code. This has resulted in writing abstract specifications for all mCRL2 classes in our own custom-made specification language. From these specifications the classes themselves are generated using a code generator implemented in Python, which was developed in-house. Next to the classes, also a generic traversal mechanism is generated, using the same specifications. Generating code from specifications has a number of advantages, namely, (1) all interfaces uniformly follow the same conventions, (2) bugs have to be fixed in either the specifications or the code generator, instead of in an implementation consisting of over 1000 LOC and (3) extending all classes with similar functionality can be achieved by creating an appropriate generator component. We now start to see the first effects in a decreasing number of handwritten LOC, and hence also in the number of LOC that need to be maintained. Figure 4 illustrates the evolution of code over time. In Figure 4(a) we see the total number of LOC in the toolset (including scripts for code generation). Figure 4(b) displays the percentage of these lines that has been generated. The combination of these figures shows that the amount of code that is generated has been gradually increasing since 2007. The points where the percentage of generated code drops coincides with the points where the total number of LOC rapidly increases. This can be explained by separately developed code imported into the toolset.

Besides generating code, effort has been put into investigating the automatic generation of test cases. Here, test cases are randomly generated within certain constraints. Using these test cases, and running algorithms on a large number of them (over 10 000), has uncovered bugs that so far went unnoticed. This can partly be explained by the randomness of the input, as it can be expected that a large number of code paths will be triggered by it. Because of these experiences we are convinced that the use of randomly generated test input is helpful, especially if multiple algorithms with the same interfaces are available, or some sort of sanity check can be done on the output of an algorithm that is tested with random input.

In addition to the generation of test cases, the unit tests and regression tests described earlier, there still is room for improvement in our test methodology. Especially, ways of systematically performing tests of our tools and testing according to user scenarios should be investigated. Furthermore, the use of objective measures, *e.g.* code coverage metrics, should be investigated in order to improve tests and test coverage.

CONCLUSIONS

We have described the challenges that occur in developing a formal methods toolset in an academic environment. We observed that the development process got out of hand. It seems that the effort required for developing and maintaining a formal methods toolset was underestimated. The changes that were enforced in the process are the ones that could be expected when we take the literature on quality of development processes into account.

As a result of poor documentation, and little attention towards the design of interfaces, large amounts of code have been rewritten during the past three years. Combined with the lack of systematic tests this slows down development. Most of the effort discussed in this paper is directed towards solving these problems.

Some of the issues that have arisen in the process are specific to academia. Most notably the quick turnover in development resources and the urge to experiment with different implementations, as well as the low focus on tool development seem to be specific to our development environment.

We feel that the key to more understandable and reliable implementations lie in a high level of abstraction in the code and (more) detailed design and source code documentation. For a more swift development process, well-defined process guidelines are required.

REFERENCES

1. Groote JF, Mathijssen AHJ, Reniers MA, Usenko YS, van Weerdenburg MJ. Analysis of distributed systems with mCRL2. *Process Algebra for Parallel and Distributed Processing*, Alexander M, Gardner W (eds.). Chapman & Hall: London, 2009; 99–128.
2. Baeten JCM, Basten T, Reniers MA. Process algebra: Equational theories of communicating processes. *Cambridge Tracts in Theoretical Computer Science*, vol. 50. Cambridge University Press: Cambridge, 2009.
3. Blom SCC, Fokkink WJ, Groote JF, van Langevelde I, Lisser B, van de Pol JC. μCRL: A toolset for analysing algebraic specifications. *Proceedings of CAV'01*, Paris, France (*Lecture Notes in Computer Science*, vol. 2102), 2001; 250–254.
4. Groote JF, Mathijssen AHJ, van Weerdenburg MJ, Usenko YS. From μCRL to mCRL2: Motivation and outline. *Proceedings of APC 25*, *ENTCS*, Bertinoro, Italy, vol. 162, 2006; 191–196.
5. van den Brand MGJ, de Jong HA, Klint P, Olivier PA. Efficient annotated terms. *Software*: *Practice & Experience* 2000; **30**:259–291.
6. Fernandez JC, Garavel H, Kerbrat A, Mateescu R, Mounier L, Sighireanu M. CADP: A protocol validation and verification toolbox. *Proceedings of the Eighth Conference on CAV*, New Brunswick, NJ, U.S.A. (*Lecture Notes in Computer Science*, vol. 1102). Springer, 1996; 437–440.
7. Hoare CAR. *Communicating Sequential Processes*. Prentice-Hall International: Englewood Cliffs, NJ, 1985.
8. Behrmann G, David A, Larsen KG. A tutorial on UPPAAL. *Proceedings of SFM-RT'04* (*Lecture Notes in Computer Science*, vol. 3185), Bernardo M, Corradini F (eds.). Springer: Berlin, 2004; 200–236.
9. Kozen D. Results on the propositional mu-calculus. *Theoretical Computer Science* 1983; **27**:333–354.
10. Mathijssen AHJ, Pretorius AJ. Verified design of an automated parking garage. *Proceedings of FMICS'06 and PDMC'06* (*Lecture Notes in Computer Science*, vol. 4346). Springer: Berlin, 2007; 165–180.
11. van Eekelen M, ten Hoedt S, Schreurs R, Usenko YS. Analysis of a session-layer protocol in mCRL2. Verification of a real-life industrial implementation. *Proceedings of FMICS'07* (*Lecture Notes in Computer Science*, vol. 4916). Springer: Berlin, 2007.
12. Stappers FPM, Reniers MA. Verification of safety requirements for program code using data abstraction. *Proceedings of AVoCS'09, ECEASST*, Swansea, vol. 23, 2009.

13. Hilderink GH. Software specification refinement and verification method with I-Mathic Studio. *Proceedings of CPA'06, Concurrent Systems Engineering Series*, Welch PH, Kerridge JM, Barnes FRM (eds.), vol. 64. IOS Press: Amsterdam, 2006; 297–310.
14. Jensen K. Coloured petri nets. *EATCS Monographs on Theoretical Computer Science*. Springer: Berlin, 1992.
15. van Beek DA, Man KL, Reniers MA, Rooda JE, Schiffelers RRH. Syntax and consistent equation semantics of hybrid chi. *Journal of Logic and Algebraic Programming* 2006; **68**(1–2):129–210.
16. Bodei C, Buchholtz M, Degano P, Nielson F, Riis Nielson H. Static validation of security protocols. *Journal of Computer Security* 2005; **13**(3):347–390.
17. Groote JF, Willemse TAC. Model-checking processes with data. *Science of Computer Programming* 2005; **56**:251–273.
18. van Dam A, Ploeger SCW, Willemse TAC. Instantiation for parameterized Boolean equation systems. *Proceedings of ICTAC'08* (*Lecture Notes in Computer Science*, vol. 5160). Springer: Berlin, 2008; 440–454.
19. Groote JF, Willemse TAC. Parameterised Boolean equation systems. *Theoretical Computer Science* 2005; **343**:332–369.
20. van Ham F, van de Wetering H, van Wijk JJ. Interactive visualization of state transition systems. *IEEE Transactions on Visualization and Computer Graphics* 2002; **8**(4):319–329.
21. Paulk MC, Weber CV, Curtis B, Chrissis MB. *The Capability Maturity Model*: *Guidelines for Improving the Software Process/CMU/SEI*. Addison-Wesley: Reading, MA, 1995.
22. *Software Engineering—Software Product Quality Requirements and Evaluation* (*SQuaRE*)—*Guide to SQuaRE*. ISO: Geneva, 2005.
23. Martin K, Hoffman B. *Mastering CMake*: *A Cross-Platform Build System*. Kitware, Inc.: Colombia, Latin America, 2010.
24. Dos Reis G, Stroustrup B. Specifying C++ concepts. *Proceedings of POPL'09*. ACM Press: New York, 2006; 295–308.
25. Garcia R, Lumsdaine A. MultiArray: A C++ library for generic programming with arrays. *Software*: *Practice & Experience* 2005; **35**(2):159–188.
26. Gregor D, Järvi J, Siek J, Stroustrup B, Dos Reis G, Lumsdaine A. Concepts: Linguistic support for generic programming in C++. *Proceedings of OOPSLA'06*. ACM Press: New York, 2006; 291–310.