Experience in developing the mCRL2 toolset

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

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{j.f.groote, j.j.a.keiren, f.p.m.stappers, j.w.wesselink, t.a.c.willemse}@tue.nl

Introduction mCRL2 is a language with a toolset [3] for formal analysis of behaviour of concurrent systems. It is developed in an academic research group to support process algebra based verification in an academic as well as an industrial setting. This paper provides an insight into the experiences and practises in developing and maintaining the mCRL2 toolset.

History. Experience in applying the μCRL toolset [1] to real world systems uncovered major shortcomings. These shortcomings led to the development of the richer modelling language mCRL2, with its associated toolset, in 2002. A motivation for the development of mCRL2 is described in [2]. In an attempt to keep development effort to a minimum, code from μCRL was reused with minimal changes required to support the extended language.

The decision to reuse μCRL code fixed important design decisions like the use of C code in the project, and the use of ATerm library [4] based data structures for the purpose of maximal subterm sharing.

In this paper we identify the challenges in the development process, and describe the development principles and coding mechanisms that we have adopted.

Development and maintenance issues In an academic environment, people tend to come and go in quick succession. Additionally writing scientific papers instead of writing code is core business. This introduces the following maintenance problems in the development process:

- documentation solely exists in the minds of the developers;
- code written by other developers needs to be maintained and debugged;

In general these issues turn out to be time consuming, especially if one takes into account that the code base consists of 273864 lines of code in 1214 files, contributed by over 20 developers. Some components have been substantially modified by 10 or more developers in disjoint time spans. Changes in core components of the mCRL2 toolset have caused bugs that remained undetected for extensive periods of time. Similarly, small changes have shown to cause dramatic decreases in performance of the tools.

We can identify the following problems in our code base:

- we adopted existing C code, based on the ATerm library, with untyped interfaces, low abstraction level and heavy use of unsafe type casts;
- interfaces of the μCRL code were not well-documented;

- portability issues in the existing code obstructed our ambition to have the mCRL2 toolset available as a cross-platform tool.

The combination of these issues induces a steep learning curve for new developers, often requiring assistance from more experienced developers. The lack of documentation, a high level interface, and general purpose algorithms inadvertently led each developer to introduce his own ad-hoc solution for general problems, and this led to a copy-paste-adjust mentality. Generalising these ad-hoc implementations into a single implementation has shown to be virtually impossible without major changes to the algorithms that use them. On the other hand generalised implementations are required to make progress in implementing new or overhauled parts of the code. Overcoming these issues badly constrains development of new features, and hence the ability to closely follow developments in the theory. An additional issue introduced by the large amount of code duplication is lack of consistent behaviour between tools.

Adopted process and coding guidelines Based on the issues that we have observed, we give an insight in the development guidelines we have since adopted.

One of the main issues that we have run into process-wise is the lack of documentation. To address this issue, developers are required to explicitly design their components prior to implementing. Furthermore the implementation needs to be accompanied with extensive source code and interface documentation.[1] To make sure that all documentation is of sufficient quality, it is now reviewed by at least one other developer. Style differences among code from various authors are reduced using coding guidelines.

In the development of a formal methods toolset correctness, reliability and performance are key issues. Therefore developers are required to write unit tests for the components they implement. Furthermore bug fixes must always be accompanied with tests that show the bug is fixed. All tests are run on a daily basis on all supported platforms. We experienced that changes in the core of the toolset can have an unanticipated but vast effect on the performance of individual tools, hence we run the core tools in our toolset on a selection of examples on a daily basis. Historical running times are accumulated for comparison. This data is used to examine tool performance trends over time.

On source code level we have run into the following issues: (1) the compiler not assisting us in finding straightforward mistakes in the code because of the untyped interfaces, (2) duplication of code, and (3) the lack of cross-platform support. These issues have been addressed by transitioning the code to C++. This addressed our problems as follows. A framework of C++ classes and interfaces was designed for the most common data structures and algorithms. This allows the compiler to do stricter checking on types. All new code has been written using this framework, leading to more comprehensible code at a higher abstraction level. A lot of existing code has gradually been adapted to use this framework, or whenever necessary been rewritten from scratch. By now there are only a few pieces of code that still need to be migrated. In this 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

[1] Note that all documentation, and test and performance measurement results that we mention are available through the project website (http://www.mcrl2.org)

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 several thousand lines of code, and enabled a uniform treatment of command-line arguments across the tools. By using external libraries that are available for all supported platforms, and by adhering to the C++ standard in our implementation, we have succeeded in porting our toolset to Windows, Linux and MacOSX. This offers the advantage of having different checks from different compilers, which again helps in improving stability of the toolset. In practice the overhead required for supporting multiple platforms is relatively low.

Currently interfaces are generalised using techniques from generic programming. This allows to create uniform interfaces with a high degree of flexibility. In an academic environment, where people require prototypes of new or altered functionality, this is a very useful property, as more time needs to be spent on experimenting, and less on coding.

Conclusions We have described the challenges that occur in developing a formal methods toolset in an academic environment. Some of the issues discussed are widely known in the software development process in general, and not only in academia. Others, like the quick turnover in development resources and the urge to experiment with different implementations, as well as the low focus on tool development, are more specific to an academic environment, and require generalised solutions.

For the issues that were raised we have discussed the solutions that we have implemented so far, as well as changes that are currently going on.

The key to more understandable and reliable implementations, as well as a more swift development process lie in a high level of abstraction in the code and (more) detailed design and source code documentation.

References

1. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. v. Langevelde, B. Lisser, and J.C. v.d. Pol. μCRL: A toolset for analysing algebraic specifications. In *Proc. of CAV'01*, volume 2102 of *LNCS*, pages 250–254, 2001.
2. J.F. Groote, A.H.J. Mathijssen, and Y.S. Usenko M.J. v. Weerdenburg. From μCRL to mCRL2: Motivation and outline. In *Proc. of APC 25*, volume 162 of *ENTCS*, pages 191–196, 2006.
3. J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. v. Weerdenburg. Analysis of distributed systems with mcrl2. In M. Alexander and W. Gardner, editors, *Process Algebra for Parallel and Distributed Processing*, pages 99–128. Chapman Hall, 2009.
4. M.G.J. v.d. Brand, H.A. d. Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000.