From μCRL to mCRL2

Motivation and outline

Jan Friso Groote, Aad Mathijssen, Muck van Weerdenburg, Yaroslav Usenko
Department of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{J.F.Groote, A.H.J.Mathijssen, M.J.van.Weerdenburg, Y.S.Usenko}@tue.nl

Abstract

We sketch the language *mCRL2*, the successor of μCRL, which is a process algebra with data, devised in 1990 to model and study the behaviour of interacting programs and systems. The language is improved in several respects guided by the experience obtained from numerous applications where realistic systems have been modelled and analysed. Just as with μCRL, the leading principle is to provide a minimal set of primitives that allow effective specifications, that conform to standard mathematics and that allow standard mathematical manipulations and proof methodologies. In the first place the equational abstract datatypes have been enhanced with higher-order constructs and standard data types, ranging from booleans, numbers and lists to sets, bags and higher-order function types. In the second place multi-actions have been introduced to allow a seamless integration with Petri nets. In the last place communication is made local to enable compositionality.

1 The history of μCRL

In an attempt to construct a language to which all existing specification languages could be translated, a common representation language (CRL) was constructed in an EC funded project called SPECS. This language became a monstrum for which is was impossible to device a coherent semantics, let alone to be used as a basis for further theory or tool building.

Upon these findings, in 1990 a minimal language called μCRL (*micro Common Representation Language*) came into being as the simplest conceivable language to model realistic systems. The language is a process algebra with data. The data is specified using first-order equational logic which was the norm at the time. Earlier developed languages such as LOTOS [2] and PSF [8] also contained equational datatypes. However, μCRL was much simpler than these languages.

In the first research phase proof methodologies were developed to give mathematical proofs of distributed algorithms and protocols. A number of proof techniques have been uncovered such as *cones and foci*, *τ-confluence* and *coordinate transformations* (see [6] for an overview). Many systems have been verified using these techniques, but particularly noteworthy is the most complex sliding window protocol in [9] (see [3]). Verification of this protocol led to the detection of an unknown deadlock in the protocol, it showed that the external behaviour of the original protocol was prohibitively complex and catalysed the development of many proof methodologies.

In the second research phase a toolset for μCRL was developed [1]. The primary motivation for this was that industrial specifications quickly became far too large to be handled manually. Large specifications, like ordinary programs, turned out to contain flaws such as deadlocks and tools were required to ensure the absence of anomalies. For plain verifications, the tool can handle systems with more than 10^9 states. By using confluence, abstract interpretation and symbolic reasoning much larger systems, containing hundreds of components have been verified. For half a decade the tool plays an essential role in teaching the design of dependable systems at various universities.

2 Why must μCRL be changed?

It turns out to be impossible to design a complete specification language that is immediately right. In [4] time was added to the language. Furthermore, constructors were added to the specification of functions in the datatypes of μCRL to make the available induction principles explicit. And finally, the possibility to specify an initial state of a process had to be added. As time passed it became more and more obvious that the language would benefit from some more changes.

First of all changes were required in the abstract data types, although their expressive power was more than sufficient. A relatively minor problem was that in μCRL all basic datatypes, such as the naturals and the booleans had to be explicitly encoded. Much more serious was the negative effect on interhuman communication of specifications. Different persons could give widely different specifications of for instance the naturals. This meant that before getting to the gist of a specification, first the specification of the naturals had to be understood. Furthermore, because all functions in μCRL are prefix functions, standard notation, such as an infix + for addition on natural numbers could not be used. This is not a problem for small specifications but seriously decreases the readability of large ones.

In practice first-order abstract datatypes also discourage the use of higher-order objects, such as functions, sets, relations and quantifiers. For instance sets are often modelled as finite lists. This tends to make specifications more complex than necessary.

A strong argument against the use of bare abstract data types came from manually proving the correctness of specifications. Given a specification, many elementary facts about the data are not self evident and proving them draws away energy from the main task, namely finding the core correctness argument for the protocol or distributed system under study. For an abstractly specified sort Nat, it is not self evident that it indeed represents natural numbers in a true way. Hence, the truth of simple identities had to be established using axioms and induction principles. For instance commutativity of addition must be established separately for each specification of natural numbers. For tools, properties like $x > y \wedge y > z \rightarrow x > z$ turned out a hurdle that was hard to overcome. By having standard data types, dedicated integer linear programming techniques can be employed with which we can prove or disprove the validity of inequality based formulas that are many orders of magnitude larger than the one above. Actually, the μCRL toolset already made a number of silent assumptions about certain data types (esp. the booleans) and certain functions (esp. it assumed that a function *eq* represented equality). This enabled the development of a very effective equality BDD prover [5] but actually violates the philosophy of abstract data types.

Despite these disadvantages, equational abstract data types were more than sufficiently expressive for any data type that needed to be specified. As the structure of data is very simple, we could device optimal algorithms to handle data with little effort. Repeated comparative experiments show that the μCRL tool set contains the most efficient state exploration tools in terms of the number of states that it can store in main memory. Comparing to for instance SPIN [7], the μCRL toolset is approximately a factor 4 slower in dealing with abstractly specified bits and bytes, which are built-in data types in SPIN.

Another issue that we ran into with μCRL is the relationship between different process specifi-

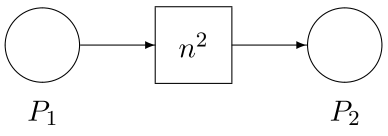

Figure 1: A simple coloured Petri net

cation formalisms. We see three main streams. There are assertional specification formalisms, Petri nets and process algebras. We would all benefit if these formalisms would be integrated. In the past we did not find any difficulties relating assertional methods and μCRL. However with Petri nets we ran into a problem. Consider the coloured Petri net in figure 1. There are two places P_1 and P_2 and a transition labeled with n^2 in the middle. The tokens in this coloured Petri net contain natural numbers and the transition squares the number in each token that it processes. The standard semantics of this system is that a token atomically leaves P_1, its value is squared and it is put in P_2.

The natural structure preserving translation of this Petri net into process algebra is the parallel process $P_1 \parallel T \parallel P_2$. Using a standard synchronous communication a token can be read from P_1 into T, and in a subsequent step be forwarded from T to P_2. But now we have translated what was a single atomic step into two atomic steps. This is bad for at least the two following reasons. In the first place this innocent looking doubling of states increases the number of states worsening the severity of the state explosion problem, which is one of the core problems we try to avoid. In the second place nice properties about Petri nets, such as state invariants do not easily carry over when introducing such intermediate states.

In order to avoid the introduction of such an intermediate state and still allow for direct structural translations, we felt forced to introduce multi-actions. In a multi-action zero or more actions can occur simultaneously. The typical notation is $a|b|c$ for a multi-action in which actions a, b and c happen at the same time. Now we can describe the transition in figure 1 by a process that reads a token with value n and in the same multi-action delivers the token with a value n^2. There is no straightforward way to do this in μCRL.

Another problem occurs when describing complex systems with non-uniform communication. In μCRL there is a global communication operator that is not compositional. To make the new language compositional, we need to define it locally.

3 The mCRL2 language

The mCRL2 language is a movement back from the bare minimum concept of μCRL towards a slightly richer language. Therefore, we propose to call it a *milli Common Representation Language*, or *mCRL*. Experience has taught that though we have designed the language with utmost care, we may still have made mistakes in its design and fundamentally new extensions such as stochastic or hybrid behaviour may be added in the future. Hence, we added a version number to the name paving the way for *mCRL3*, *mCRL4*, etc. to come. By the way, the name μCRL is not really suited for internet because of the initial Greek letter.

3.1 Data language

The mCRL2 data language uses *higher-order* abstract data types as a core theory. To this theory, standard data types are added. We list these data types without further ado as they are commonly known. All the common operators on these are made available in normal mathematical notation. In order to get a quick idea, an expression using this datatype is provided.

- The sort $\mathbb{B}$ with constants *true, false* and all standard operators. It is also possible to use the quantifiers $\forall$ and $\exists$ ranging over any datatype. E.g. $b \wedge \mathit{false} \Rightarrow \forall n{:}\mathbb{N}.n < 3$.

- Unbounded positive, natural and integer numbers. Typical examples of expressions using numbers are $1 - 464748473698768976\ \mathbf{div}\ \mathit{exp}(3, n)$, $\mathit{succ}(m) \leq n - 1$ or $x == x * x - 1$.

- Function types. For two given sorts *A* and *B* the sort $A{\rightarrow}B$ contains all functions from domain A to B. Function application and lambda abstraction are part of the language. E.g. let $f = \lambda x{:}\mathbb{N}, b{:}\mathbb{B}.\mathit{if}(b, x, 2 * x)$. Then $f(3, \mathit{false})$ is equal to 6.

- Following functional languages, it is possible to declare structured types. These are especially useful for enumerated data types and complex data structures such as for instance trees. A sort *MS* of machine states can be declared by

  $$\mathbf{sort}\ \mathit{MS}\ =\ \mathbf{struct}\ \mathit{off} \mid \mathit{standby} \mid \mathit{starting} \mid \mathit{running} \mid \mathit{broken};$$

  The sort of binary trees with numbers as their leaves looks like

  $$\mathbf{sort}\ T\ =\ \mathbf{struct}\ \mathit{leaf}(\mathbb{N}) \mid \mathit{node}(T, T);$$

  It is possible to specify projection and recognition functions simultaneously, e.g.:

  $$\mathbf{sort}\ T\ =\ \mathbf{struct}\ \mathit{leaf}(\mathit{getnumber}{:}\mathbb{N})?\mathit{isLeaf} \mid \mathit{node}(\mathit{left}{:}T, \mathit{right}{:}T)?\mathit{isNode};$$

- Because lists are very commonly used datatypes, there is a built-in type of lists with standard operations. The list of natural numbers is $\mathit{List}(\mathbb{N})$. The following list expressions are all equivalent: $[3, 4, 5]$, $3 \triangleright [4, 5]$, $[3, 4] \triangleleft 5$ and $[\,] \mathbin{+\!\!+} [3, 4] \mathbin{+\!\!+} [5]$.

- Sets are very commonly used in mathematical specification, and as bags are a basic concept in Petri nets, both have been included in the language. Sets are denoted in the normal mathematical way. Typically, $\{1, 2, 4\}$, $\{1, 2\} \cup \{1, 4\}$ are sets. The set of primes is

  $$\{n{:}\mathbb{N} \mid \forall m{:}\mathbb{N}.(1 < m \wedge m < n\ \Rightarrow\ n\ \mathbf{mod}\ m > 0)\ \}.$$

- Bags are sets where the multiplicity of elements is recalled. For enumerations this count is appended to each element, e.g. $\{0{:}0, 1{:}1, 2{:}4\}$. For comprehensions the boolean condition is replaced by a natural number, e.g. $\{m{:}\mathbb{N} \mid m^2\}$ is the bag in which each number m occurs m^2 times.

Currently, there are discussions about the inclusion of real numbers. As functions are available, it is possible to represent real numbers. Moreover, this opens the way towards stochastic and hybrid systems where functions from reals to reals play an important role. Another interesting concept is the selector functions ε. The expression $\varepsilon x{:}S.c(x)$ equals a unique value x that satisfies condition $c(x)$. It satisfies the axiom $\exists x{:}S.c(x) \Rightarrow c(\varepsilon x{:}S.c(x))$. These extensions may show up in mCRL3.

3.2 Multi-actions and local communication

In order to facilitate the connection with Petri nets, multi-actions are introduced. A multi-action is a collection of ordinary actions that happen at the same time. A few examples of multi-actions are $a,\ a|b,\ b|a,\ a|b|c,\ a|b|a$ and $a(t)|b(u)|a(v)$.

In mCRL2 parallel composition does not communicate. Instead, it introduces multi-actions, e.g. the composition $a \parallel b$ of actions a, b is equal to $a \cdot b + b \cdot a + a|b$. As a result the number of multi-actions can increase exponentially in the size of the number of parallel compositions. Hence, we also need operators to restrict this behaviour. First of all we have the *blocking* operator ∂_H (which was called encapsulation in μCRL) that blocks all multi-actions of which a part occurs in the action set H, e.g. $\partial_{\{a\}}(a + b \cdot (a|c)) = b \cdot \delta$. On the other hand, we have the visibility operator ∇_V called *allow* that specifies precisely which multi-actions are allowed, namely the ones in V. For instance $\nabla_{\{\,a,b\,\}}(a \parallel b) = a \cdot b + b \cdot a$, $\nabla_{\{\,a|b\,\}}(a \parallel b) = a|b$, and $\nabla_{\{\,a,b|c\,\}}(a \parallel b \parallel c) = a \cdot (b|c) + (b|c) \cdot a$.

Communication of actions is defined using the concept of multi-actions. The *local* communication operator Γ_C realises communication of multi-actions with equal data arguments. Unlike μCRL, communication does not block. For instance, if $t = u$ and $t \neq v$, then $\Gamma_{\{\,a|b\to c\,\}}(a(t)|b(u)) = c(t)$, $\Gamma_{\{\,a|b\to c\,\}}(a(t)|b(v)) = a(t)|b(v)$ and $\Gamma_{\{\,a|b|c\to d\,\}}(a|b|c|d) = d|d$, but also $\sum_{d:D} \Gamma_{\{\,a|a\to a\,\}}(a(d)|a(t)) = \sum_{d:D} d = t \to a(t), a(d)|a(t)$, i.e. if $d = t$ then $a(t)$ and if $d \neq t$ then $a(d)|a(t)$ for a certain d.

4 Epilogue

The language mCRL2 is an attempt to make μCRL more applicable in practise and to facilitate hierarchical Petri nets. The language is extended with higher-order datatypes, standard datatypes, multi-actions and local communication. Because the new language has essentially the same structure as its predecessor, all current μCRL specifications can be easily expressed in the new language and all proof methodologies, theorems and tools carry over with only minor modifications.

References

[1] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In proceedings CAV'01. LNCS 2102, pages 250–254, 2001.

[2] P.H.J. van Eijk, C.A. Vissers and M. Diaz, editors. *The formal description technique LOTOS*. Elsevier Science Publishers B.V., 1989.

[3] W. Fokkink, J.F. Groote, J. Pang, B. Badban and J.C. van de Pol. Verifying a sliding window protocol in μCRL. In C. Rattray, S. Maharaj and C. Shankland (eds), proceedings of the 10th International Conference on Algebraic Methodology and Software Technology, Stirling, Scotland, LNCS 3116, Springer-Verlag pp. 148-163, 2004.

[4] J.F. Groote. The syntax and semantics of timed μCRL. Technical report SEN-R9709, CWI, Amsterdam, 1997.

[5] J.F. Groote and J.C. van de Pol. Equational Binary Decision Diagrams. In M. Parigot and A. Voronkov, *Logic for Programming and Reasoning, LPAR2000*, Lecture Notes in Artificial Intelligence, volume 1955, Springer Verlag, pages 161-178, 2000.

[6] J.F. Groote and M. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse and S.A. Smolka. Handbook of Process Algebra, pages 1151-1208, Elsevier, Amsterdam, 2001.

[7] G.J. Holzmann. The spin model checker: Primer and reference manual. Addison-Wesley, 2003.

[8] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[9] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.