From Dezyne to Model Checking

Rutger van Beusekom\textsuperscript{2}, Jan Friso Groote\textsuperscript{1}, Rob Howe\textsuperscript{2}, Wieger Wesselink\textsuperscript{1}, Rob Wieringa\textsuperscript{2}, and Tim A.C. Willens\textsuperscript{1}

\textsuperscript{1} Eindhoven University of Technology, The Netherlands
\textsuperscript{2} Verum B.V., The Netherlands

Abstract. Dezyne is an industrial language with an associated set of tools, allowing users to model interface behaviours and implementations of reactive components and generate executable code from these. The tool and language succeed the successful ASD.Suite toolset, which, in addition to modelling reactive components, offers a set of verification capabilities allowing users to check the conformance of implementations to their interfaces. In this paper, we describe the Dezyne language and a model transformation to the mCRL2 language, providing users access to advanced model checking capabilities and refinement checks of the mCRL2 tool set.

1 Introduction

The general consensus is that formal verification of system designs helps to reduce development costs by detecting issues early and by increasing the overall reliability of the system. Over a period of forty years, formal verification has grown from a pure academic exercise to a rich research area with promising industrial application. Many major firms have adopted these verification techniques in one way or another, using them to make better quality systems. However, the success of formal verification is directly linked to the maturity of the tooling performing the analysis and the skill of the verification engineer.

Most of the available tooling requires highly skilled and experienced verification engineers to tackle serious and complex industrial problems. Verum has created the ASD tool suite in an attempt to shield the system designer from the complexity of the verification language and technology by offering an intuitive integrated development environment for specifying complex, concurrent, industrial systems. The current ASD tool suite shields the complexity so well that it is easy to use for both novice and experienced system designers, but limits the more experienced designers in accessing the full power of formal verification. Currently the ASD tool suite offers a proprietary design language and associated development methodology that is built on top of the verification technology offered by the FDR tool suite, which offers facilities for checking deadlock, livelock and refinement.

In an effort to move beyond these limitations, Verum started to design Dezyne, a language which builds upon the concepts familiar to the users of ASD, yet richer in terms of constructs and facilities. FDR again provides the
back-end for the verification, but this time, also a prototype connection to the mCRL2 tool suite was developed in the context of the FP7 technology transfer project VICTORIA. This was done to allow Verum to offer new services to its expert users. Such new services that can be offered range from the ability to design data-dependent systems, to checking user-specific safety and liveness properties that go beyond deadlock and livelock, to offering advanced behavioural visualisation tooling to end-users.

In this paper, we report on the formalisation of Dezyne using mCRL2. We address issues such as the semantics of Dezyne programs but also the technology we used to program the connection between mCRL2 and Dezyne.

2 An Overview of mCRL2

The mCRL2 language \cite{mCRL2} consists of three distinct parts: a data language for describing the data transformations and data types, a process language for specifying system behaviours and a modal language for reasoning about the system behaviours.

2.1 Data Language

The data language of the mCRL2 language is based on higher-order abstract equational data types. The language has built-in definitions for many of the commonly used data types (and operations on these), such as Booleans, represented by sort \texttt{Bool}, (unbounded) Integer, Natural and Positive numbers, represented by sorts \texttt{Int}, \texttt{Nat} and \texttt{Pos}, respectively. Container sorts, such as (infinite) sets, bags and lists over arbitrary data sorts \texttt{D} are denoted by \texttt{Set(D)}, \texttt{Bag(D)}, and \texttt{List(D)}. New data sorts can be defined either by directly specifying the constructors of a data sort, or using structured sorts, or through aliasing. Relations and mappings on data sorts, and their rules of logic are formalised through equational rewrite rules. Defining data sorts through structured sorts introduces a few built-in mappings and relations, such as projection functions, equality and inequality. Expressions in the data language can be built by combining sort constructors, functions, relations and data variables. Example expressions are 3+4 of sort \texttt{Pos} and \texttt{b && (b || false)} of sort \texttt{Bool}.

Example 1. A sort representing the Cartesian product of integer numbers can be defined by aliasing it to a structured sort \texttt{struct coord(x:Int,y:Int)}, \texttt{i.e.,}

\texttt{Point = struct coord(x:Int,y:Int).} The mappings \texttt{x} and \texttt{y} are the projection functions: given an expression \texttt{coord(0,1)} of sort \texttt{Point}, the expression \texttt{x(coord(0,1))} is an integer number (the number 0 in this case).

Of particular importance for the formalisation of SML programs is lambda abstraction, which is well-known from \textlambda-calculus and type theory. Functions, specified through lambda abstraction or otherwise, can be updated concisely and intuitively, as illustrated by the following example.
Example 2. Let $f: \text{Nat} \rightarrow \text{Nat}$ be a user-defined mapping of sort \text{Nat} to \text{Nat}; the function $f$ can be thought of as an infinite array. Using lambda abstraction, we can define $f$ to map each natural number to its doubled number: $f = \lambda n: \text{Nat}. n+n$. Using function updates, function $f$ can be modified: $f[0\rightarrow 2]$ agrees with function $f$, save for $f[0\rightarrow 2](0)$, which has value 2, whereas $f(0)$ has value 0.

The built-in data types are designed to reflect their mathematical counterparts, contributing to the accessibility of the data language. The support for universal and existential quantifiers further facilitates conventional mathematical reasoning.

2.2 Process Language

The process specification language of mCRL2 consists of only a small number of basic operators and primitives. The language is inspired by process algebras from the ACP family [?], and has both an axiomatic and an operational semantics. We forego a formal exposition of its semantics, for which we refer to [?, ?]; instead, we restrict ourselves to introducing its syntax, sketch its meaning informally and illustrate its use through small examples.

Processes are constructed compositionally using alternative composition and quantification, sequential and parallel composition, hiding, communication, and recursion. The basic behavioural elements of a process are the deadlock process delta and (parameterised) actions. The latter represent atomic, observable events, such as receiving status updates or the sending of commands; the parameters can be used to represent the data that is linked to such events. If read is an action name, and $e$ is some data expression of the sort of action name read, then read($e$) is a parameterised action.

Suppose $p$ and $q$ are processes, then their alternative composition is denoted $p+q$. Intuitively, process $p+q$ behaves as either process $p$ or $q$, dependent on which of the two processes executed the first action. Since process delta can never execute actions, process delta+p will simply behave as process $p$. Alternative quantification generalises alternative composition: if a data variable $d$ of some sort $D$ occurs in process $p$, then \text{sum} $d$:$D. p$ denotes the (possibly infinite) choice between the set of processes obtained by instantiating variable $d$ in process $p$ with all possible values it can attain.

Example 3. Suppose read(n+n) is an action parameterised with natural number expressions, then process \text{sum} $n$:$\text{Nat}. \text{read}(n+n)$ denotes the infinite set of processes offering read actions with even values as their parameters.

Using a binary if-then construct, denoted $b \rightarrow p$, or a ternary if-then-else construct, denoted $b \rightarrow p <> q$, processes can be made data-dependent: if $b$ evaluates to true, then processes $b \rightarrow p$ and $b \rightarrow p <> q$ behave as process $p$, and the latter behaves as process $q$ otherwise, whereas the former behaves as process delta.
The sequential composition of processes \(p\) and \(q\), denoted \(p.q\), behaves as process \(p\), and, upon successful termination of \(p\), continues to behave as process \(q\). Note that the deadlock process \(\delta\) never terminates successfully; hence, process \(\delta.p\) is indistinguishable from process \(\delta\).

**Example 4.** Let \(\text{read}\) and \(\text{send}\) be two parameterised actions, taking natural number expressions as their arguments, and let \(n\) of sort \(\text{Nat}\) be a data variable. Then process \(\text{read}(n).\text{send}(n)\) represents a system in which first a value represented by data variable \(n\) is read, which is then sent via action \(\text{send}\). The process \(\text{sum } n: \text{Nat}. \text{read}(n).\text{send}(n)\) combines sequential composition with alternative quantification, modelling that any natural number read through action \(\text{read}\) is sent via action \(\text{send}\).

A parallel composition of processes \(p\) and \(q\), which is denoted by process \(p|\!|q\), behaves as the interleaving of both processes involved: the first action may come from either process \(p\) or \(q\), after which the process that “remains” is composed in parallel with the process that did not execute the first action. In addition, both processes may execute their first actions simultaneously, producing a multi-action, after which the processes that remain are again composed in parallel.

**Example 5.** Consider the process \(p\) defined as \(\text{sum } n: \text{Nat}. \text{read}(n).\text{send}(n)\) and \(q\), defined as \(\text{sum } m: \text{Nat}. \text{send}(m).\text{read}(m+m)\). The parallel composition of \(p\) and \(q\), may first execute a \(\text{read}(n)\) action, after which it will behave as the parallel composition \(\text{send}(n)|\!|q\), it may first execute a \(\text{send}(m)\) action, after which it will behave as the parallel composition \(p|\!|\text{read}(m+m)\), or both processes may execute their first actions simultaneously, denoted by the multi-action \(\text{read}(n)|\!|\text{send}(m)\), after which the remaining process behaves as \(\text{send}(n)|\!|\text{read}(m+m)\).

Exchanging information between processes by synchronising on specific events is achieved using the \(\text{comm}\) operator. This operator takes a set of communication rules and a process as its argument; the communication rules specify which multi-actions communicate successfully. Exchange of information and successful communications are only achieved if the involved actions agree on all the values of their parameters.

**Example 6.** Consider again processes \(p\) and \(q\) from the previous example. Suppose actions \(\text{read}\) and \(\text{send}\) can communicate, yielding a new parameterised action \(\text{sync}\). Then process \(\text{comm}([\text{read}|\!|\text{send}\rightarrow\text{sync}], p|\!|q)\) will convert the multi-action \(\text{read}(n)|\!|\text{send}(m)\) to \(\text{sync}(n)\) whenever \(n=m\), and leave the multi-action intact in all other cases.

Process behaviours can be restricted to a use a set of atomic actions and multi-actions only, using the \(\text{allow}\) construct. This provides the means to enforce that non-successful synchronisations of two parallel processes are not considered. Effectively, the process \(\text{allow}(A, p)\), where \(A\) is a finite set of atomic action names and multi-actions, behaves as process \(p\), except that any action or multi-action in \(p\) that is not in \(A\) is replaced by the deadlock process \(\delta\).
Example 7. Let $p$ and $q$ again be the processes from the previous example. The process $\text{allow}([\text{sync}], \text{comm}([\text{read}|\text{send} \rightarrow \text{sync}], p||q))$ will behave as $\text{sum } n:\text{Nat}. \text{sync}(n).([\text{sum } m:\text{Nat}. (n=m+m) \rightarrow \text{sync}(n) <> \text{delta}])$. That is, if a non-zero value is communicated between processes $p$ and $q$, then no further communication happens (since $n=m+m$ evaluates to $\text{true}$ only if both $n$ and $m$ are zero) and the process locks; if the value zero is exchanged, then this is done once more, after which the process successfully terminates.

Finally, recursive equations of the form $X(d_1:D_1, \ldots, d_n:D_n) = p$, allow for specifying infinite behaviours. Intuitively, each occurrence of a parameterised process variable $X(e_1, \ldots, e_n)$ in some process term $q$ behaves as process $p$, in which the variables $d_i$ have been replaced by expressions $e_i$.

Example 8. Consider process equation $X=$ \text{sum } $m:\text{Nat}. \text{read}(m).X$ and process equation $Y(n:\text{Nat})=$ \text{sum } $i:\text{Nat}. (i > n) \rightarrow \text{send}(i).Y(i)$. Process $X$ can execute a $\text{read}$ action with an arbitrary natural number parameter, after which it again behaves as process $X$. Process $Y(0)$ can execute a $\text{send}$ action with an arbitrary non-zero natural number parameter $i$, after which it behaves as process $Y(i)$. Effectively, process $Y(i)$ specifies a system that sends ever-increasing numbers.

If a process variable occurs within the scope of its own equation, a shorthand notation for updating only part of the data parameters of that equation is available. For instance, if process variable $X(d_1, \ldots, e_i, \ldots, d_n)$ occurs in the right-hand side process $p$ of an equation $X(d_1:D_1, \ldots, d_n:D_n) = p$, then we can write $X(d_i=e_i)$ instead.

2.3 Modal Language

Whereas the process language is typically used to specify how a system achieves its behaviour, the modal language is typically used to reason about high level requirements of such systems. The modal language of mCRL2 is based on the theory of the modal $\mu$-calculus [?], extended with facilities to reason about data, see [?].

Apart from standard Boolean connectives such as conjunction and disjunction, the mCRL2 modal language permits the use of existential and universal quantification over data sorts (specified by the data language), and the use of Boolean expressions. In addition the language permits the use of modalities. The $\text{must}$ modality $[A]f$ expresses that any first action $a(v)$ executed by a process will result in a process that satisfies property $f$ if action $a(v)$ is among the actions in the set of actions described by $A$. Dually, the $\text{may}$ modality $<A>f$ asserts that among the set of first actions that can be executed by the process, there is one action that is contained in the set of actions described by $A$, and which, if executed, will result in a process satisfying property $f$. The modal language permits describing infinite sets of actions, which is needed because of the possibly infinite branching processes that can be described by the process language.
Example 9. The set of actions characterised by \textit{true} is the entire set of actions; the set of actions characterised by \textit{exists n:Nat. read(n)} is the set of read actions with a parameter taken from the set of all natural numbers. Finally, \textit{!exists n:Nat. read(n+n)} specifies the entire set of actions save those read actions with even valued natural number parameters. Thus, \textit{<true>true} asserts that a process can execute an action, and \textit{![exists n:Nat. read(n+n)]false} asserts that a process can at most execute read actions with even valued natural numbers. Lastly, \textit{forall n:Nat. <read(n)>true} asserts that a process can execute read actions with every natural number parameter. This property holds of a process such as \textit{sum m:Nat. read(m)}, but not of a process such as \textit{sum m:Nat. read(m+m)}.

Finally, least and greatest fixpoints, denoted by \textit{mu X. f(X)} and \textit{nu X. f(X)}, respectively, permit reasoning about finite and infinite runs of a system. Typically, least fixpoints are used to specify \textit{eventualities}, whereas greatest fixpoints are used for \textit{invariants}. By mixing least and greatest fixpoints, increasingly complex properties, such as particular fairness properties, can be stated.

Example 10. The property \textit{nu X. <exists n:Nat. read(n)>X} asserts that a process is capable of executing an infinite sequence of read actions, without requiring anything about the parameters these carry. In a similar vein, the property \textit{forall n:Nat. [read(n)]mu X. ![send(n)]X} asserts that a read action with some natural number parameter will inevitably result in a send action with that same natural number parameter.

The modal language of mCRL2 features \textit{parameterised} recursion, but since we will not use this construct, we will not elaborate on it.

2.4 Tooling

The language mCRL2 has a homonymously named toolset, offering tools that help to understand specifications written in the data language and the process language, and tools that can check whether properties written in the modal language hold of a process description or not. For an overview of most common tools, we refer to [?]; we here confine ourselves to give a high-level overview of the techniques that were most relevant for our purposes.

Data expressions can be evaluated using an interpreter. Typically, the interpreter can help in understanding why expressions can be further simplified or not. For instance, one may ask whether for a complex open Boolean data expression \textit{b}, there is an assignment to some of the variables in \textit{b} so that the expression evaluates to \textit{true}. Note that it is very easy to state undecidable properties in the language, so such tooling helps assess whether the technology used for reasoning about data is sufficiently powerful to work with the expressions used, \textit{e.g.}, in the process description of a certain system. For instance, one may wonder whether for an unknown natural number \textit{k}, the expression \textit{exists n:Nat. k < n && n < 4} evaluates to \textit{true}, \textit{false}, or some other expression; in this case, the interpreter will simplify the expression to \textit{k < 3}. 

6
The behaviour described by processes can be simulated or explored exhaustively by investigating the combinatorial possibilities of the actions that can be executed, resulting in a state space such as a labelled transition system. Such a state space can be visualised in 2D or 3D using a variety of advanced techniques. Moreover, reduction techniques allow for minimising the state space using well-known equivalence reductions such as strong bisimilarity, similarity, trace equivalence and (divergence-sensitive) branching bisimilarity.

Verification of the behaviour described by processes is supported by computing whether a given functional requirement, expressed as a modal μ-calculus formula holds of the process or not; this is known as model checking. For specific types of requirements, counterexamples that are easy to interpret can be reported in the case the requirement fails on the given process. This facilitates debugging the cause of the failure.

3 The Dezyne Language

Conceptually, the Dezyne language takes inspiration from Mealy machines and concepts offered by process algebras. The language allows for specifying the interface behaviour and the implemented behaviour of components. The language for describing both is, save some small details, identical. In this section we informally describe the main concepts and ideas behind the Dezyne language through a few examples and we sketch how it relates to mCRL2. A more formal view on some of these aspects is provided in the next section.

A first example of an interface description in Dezyne is offered by Listing 1. The interface specification of Listing 1 describes the behaviour of the interface ‘Sensors’. The keywords ‘in’ and ‘out’ declare the directions of events (in this case ‘Active’, ‘Deactivate’ and ‘DetectedMovement’) that take place at the interface: the former are input events, whereas the latter are output events. Apart from declaring events, additional data types and typed variables can be introduced: in the example, a new local data type ‘States’ is introduced, consisting of three elements, and a global (global to the interface behaviour) data variable ‘state’ of type ‘States’ is introduced and initialised to ‘Deactivated’.

Conceptually, an interface specification describes the sequence of allowed and expected events that can take place at a given interface. The model that is adopted here is one that resembles that of Mealy machines: inputs are followed by a sequence of output events and/or state updates and/or function calls and an (implicit in case of a ‘void’ input event) output return value upon ‘successful completion’ of the event. Apart from the declared interface events, the constant events ‘optional’ and ‘inevitable’ indicate ‘spontaneous’ input events. These events are used to model that the interface can itself generate a sequence of output events without first receiving an input event. There is a subtle distinction between both constants: the outputs generated as a result of an optional are potentially present, but, progress/responsiveness of an interface should not rely on the optional events being implemented; events following and inevitable constant, however, are bound to occur in the presence of the absence of any
Listing 1: An example of an interface specification.

other activity at the interface. Thus, the latter permits to model events such as timeouts, whereas the former permits to model design variabilities. The constant ‘illegal’ is used to signal that an unexpected input event occurred.

The behaviour of an interface is described in the section following the ‘behaviour’ keyword. Inside this section, again data types and locally global variables can be introduced. The core of the behaviour section is made up of the specification of the actual behaviour, which in turn is built from guarded statements, event statements, function calls and assignments. An input event is preceded and followed by zero or more guards, but no guards can occur after the first assignment statement or output event statement. Conceptually, the guards and input event act as a ‘precondition’, and the sequence of assignments, function calls and output events are the ‘effect’. Consider a statement of the form:

```
[state != Deactivated]
{
  on Activate: [state == Activated] { state = States.Activated; }
  [otherwise] { illegal; }
  on Deactivate: ...
}
```

In such a statement, the condition `state != Deactivated` and the condition `state == Activated`, together with the occurrence of the input `Activate`, act as a precondition to the state update `state = States.Activated`. 
From a high level perspective, the transformation of an interface to mCRL2 is rather straightforward. For instance, an mCRL2 process capturing the behaviour of the interface of Listing 1 is the following: The interface behaviour is thus

```ml
proc Sensor_Interface(state: States)
*
(state == Deactivated) ->
(Sensor_in(Activate). return. Sensor_Interface(state = Activated)
+ Sensor_in(Deactivate). illegal. delta)
*
(state == Activated) ->
(Sensor_in(Deactivate). return. Sensor_Interface(state = Deactivated)
Sensor_Interface(state = Triggered)
+ Sensor_in(Activate). illegal)
*
(state == Triggered) ->
(Sensor_in(Deactivate). return. Sensor_Interface(state = Deactivated)
+ Sensor_in(Activate). illegal)
```

Listing 2: Conceptual mCRL2 translation of the interface specification of Listing 1.

conceptually modelled as a recursive process, where input events (and optional and inevitable events), and sequences of output events alternate. At the end of a sequence of output events, a return is issued (possibly outputing some value that has been computed as a result of the actions executed) and a recursive call to the process describing the interface behaviour is made, and all potential state updates are applied. The actual transformation to mCRL2 is slightly more involved (even if the result is semantically equivalent to the one depicted in Listing 2). There are two major reasons for this:

1. Output events and assignments can freely mix, and there can be multiple (consecutive) assignments;
2. Function calls can have side effects on the locally global state space.

We discuss both separately.

**Assignments.** In Dezyne, an assignment is, just as in most process algebras, a non-observable event. Since we are modelling the interface behaviour as a recursive process, the only means we have to update the values of the locally global state variables is to set their values while invoking a recursive call. But we cannot simply introduce a recursive call whenever we encounter an assignment: if we do so, we need to explicitly maintain a program counter that indicates which
event/assignment we last encountered and where we should continue. Manually adding a program counter requires a lot of additional bookkeeping. Instead, we introduce the program counter implicitly by associating a separate recursive subprocess to each subsequence of events/assignments: in case we are then translating a sequence of events/assignments where the first statement is an assignment, we simply pass the updated value to the locally global state variable to the next subprocess. To illustrate this, consider the following imaginary subfragment of interface:

.. on e: {x = T.a ; x = T.b; }

Assuming this is part of the behaviour section of an interface $I$, we would obtain a process $I\_Interface(x : T)$ and subprocesses of the following form:

```
proc I\_Interface(x : T) = I\_Interface\_0(x) + ...;
proc I\_Interface\_0(x : T) = I\_in(e). I\_Interface\_1(x);
proc I\_Interface\_1(x : T) = I\_Interface\_2(x = a);
proc I\_Interface\_2(x : T) = I\_out(e'). I\_Interface\_3(x);
proc I\_Interface\_3(x : T) = I\_Interface\_4(x = b);
proc I\_Interface\_4(x : T) = I\_Interface();
```

Recursive functions. In certain situations there is a need to model the repeated execution of a certain output until a desirable situation is reached. A natural way to model this is to use iteration or recursion; in DEZYNE, only the latter is offered. In DEZYNE, recursion is restricted to tail recursion. A complicating factor, however, is that recursive functions can modify the locally global state variables; that is, assignments to such variables, specified in the function body, are persistent upon completion of the function call. This precludes simply modelling a recursive function as an mCRL2 process, as, upon termination of that process, the changes made to (copies of) the locally global state variables are not persistent, nor does mCRL2 offer facilities for passing values upon termination of a process to a new process.

The solution we apply for this is to identify all variables relevant to the locally global state of an interface and 1) copy those to the mCRL2 process modelling the recursive function, 2) upon completion of the recursive function (which is simply the end of a sequence of statements not containing a function call), communicate the values of the locally global state variables to a dedicated register process that runs parallel to the interface process, and 3) read the values of the state variables from the register before continuing executing the remainder of the statements of the interface. By enforcing successful communication with the register, we manage to capture the intended side effects of recursive function calls.

For an example, consider the following DEZYNE interface:
References


