Formalising the Dezyne Modelling Language in mCRL2

Rutger van Beusekom[2], Jan Friso Groote[1], Paul Hoogendijk[2], Robert Howe[2], Wieger Wesselink[1], Rob Wieringa[2], and Tim A.C. Willemse[1]

[1] Eindhoven University of Technology, The Netherlands
[2] Verum Software Tools 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 tool set, 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

Companies increasingly rely on model-driven engineering for developing their (software) systems. The benefit of this approach, in which a high-level (often domain-specific) modelling language is used for designing systems, is that it raises the level of abstraction, resulting in an increased productivity and higher dependability of the developed artefacts. Formal verification of the models may help to further reduce development costs by detecting issues early and by further increasing the overall reliability of the system. However, the success of formal verification is directly linked to the maturity of the tooling used for performing the analysis. Most of the available tooling requires highly skilled and experienced verification engineers to tackle complex industrial problems.

The company Verum has created the ASD:Suite tool suite in the past, 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. This tool suite relies on a proprietary design language and associated development methodology. The latter is built on top of the verification technology offered by the FDR tool suite [4], which offers facilities for checking deadlock, livelock and refinement. While ASD:Suite is easy to use for both novice and experienced system designers, it limits more experienced designers in constructing more complex models and accessing the full power of formal verification.

In an effort to move beyond these limitations, Verum has designed a new, open modelling language called *Dezyne*[3], that, compared to ASD, is richer in terms of constructs and facilities. FDR still is the *de facto* back-end for conducting verifications, through

[3] See `https://www.verum.com`; accessed 21 May 2017.

a non-documented proprietary translation of Dezyne models to FDR models, but the open nature of the language enables offering alternative verification technology through other back-ends. This will allow Verum and others to offer new services for expert users.

In this paper, we provide an encoding of the Dezyne modelling language in the mCRL2 process algebra [5], thus giving a formal semantics to Dezyne models. We address issues such as the transformation of Dezyne models to mCRL2 process expressions, which we describe as formal as possible without going into unnecessary detail. Moreover, we also discuss the technology that we used to program the transformation between Dezyne and mCRL2, and illustrate how the connection to mCRL2 and its analysis tool set [3] can be used as the basis for future verification services that can check for a much wider range of user-specific safety and liveness properties, and to offer advanced behavioural visualisation tooling to end-users.

The work we report on has been conducted in the context of the FP7 TTP VICTORIA. It took over 1 man-year of effort, of which a large portion was spent on uncovering details about Dezyne's (execution) semantics, but also on improving the transformation to mCRL2 so that it yields mCRL2 models for which verification scales well. Moreover, our efforts led to a few improvements in the existing FDR translation, but also to some improvements and enhancements in the mCRL2 tool set.

Structure of the Paper. We introduce the Dezyne language in Section 2 and our mCRL2 encoding of Dezyne in Section 3. In Section 4, we discuss improvements in the mCRL2 tool set that were a direct result of the project and in Section 5 we discuss experiments using two versions of our translation and we illustrate some of the technology that becomes available through our translation. Section 6 finishes with closing remarks.

2 Dezyne

Dezyne is a language and design methodology for specifying the behaviours of *interfaces* and *components* and checking the compliance between these. The language constructs for describing interfaces are, save some small details, identical to the language constructs available for describing components, and take cues from the theory of Mealy machines and borrow concepts from process algebras. Dezyne offers rudimentary facilities for using data variables of Boolean, (bounded) integer or user-defined enumerated types.

Components specified in Dezyne assume a specific execution model, in which a component deals with inputs one at a time. That is, in standard practice a single-threaded *run-to-completion* semantics is employed. A 'user' of a component can interact with the component by sending events to it; these events are handled *synchronously* in the sense that the component essentially will remain blocked for unsolicited events from lower-level components (which run concurrently with the component) until the user receives a *reply* from the component, while solicited events from lower-level components are buffered and dealt with one at a time. Unsolicited events emitted by lower-level components are dealt with in a similar fashion; such events may result in 'spontaneous' outputs emitted asynchronously by the component.

The design methodology and system architecture implemented in Dezyne is illustrated in Fig. 1. As a designer employing the Dezyne language and methodology, one is only concerned with specifying the behaviour of interfaces and components. Subsequent

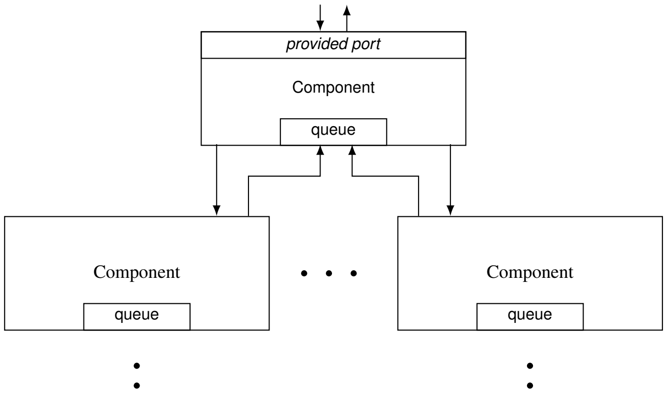

Fig. 1. Typical architecture in Dezyne. Components interact with other components through ports. Components interact with other components in a hierarchical fashion. Each component has an interface specification which formalises how its behaviour at the provided port is *expected* to behave. A compliance check verifies whether a component *actually* respects its interface.

checks compute whether the behaviour of a component as observed at its *provided* port (*i.e.* as seen by the 'user' of the component), when interacting with components through its *required* ports (*i.e.* the low-level components), formally complies with the behaviour as specified by its interface. This way one obtains a modular, hierarchical design of a software system. The modular design and compliance check are pivotal for designing large systems that are correct-by-design.

The essential part of the grammar of Dezyne is depicted in Table 1; we have omitted those parts that are required for describing a *system*; the latter is essentially a collection of components and a static description of how they are connected. Dezyne's static semantics excludes models in which there are obvious naming conflicts and consistency issues (*e.g.* multiple interface specifications with the same name are not permitted; events can be declared at most once in an interface, *etcetera*). Some constraints are there to enforce the typical tree-like architectural design pattern of Fig. 1, used in Dezyne (*e.g.* each component has at most one provided port). Most importantly for our exposition is the fact that correct interface specifications, components and recursive functions can be rewritten to a normal form where the behaviour can be represented by the following production rules:

```
BehaviourStmt             ::= [ Guard ] OnEventStmt
...
FuncDecl                  ::= ID ID ( (ID ID)? ){ (ImperativeStmt)* }
...
OnEventStmt               ::= on OnTrigger : ImperativeStmt
ImperativeStmt            ::= CompoundImperativeStmt | ReplyStmt | IllegalStmt
                              AssignmentBehaviourStmt | ActionStmt | ReturnStmt| ConditionalStmt
CompoundImperativeStmt ::= { (ImperativeStmt)* }
ConditionalStmt           ::= if Guard then ImperativeStmt else ImperativeStmt ;
```

In essence, this means that each interface and component specifies a sequence of responses and assignments for each event stimulating the interface or component.

Table 1. EBNF for (the essential part of) the Dezyne language. Terminal symbols are typeset in bold. For brevity, optional productions are enclosed within parentheses and a question mark (_)? whereas repetition, resp. positive repetition of productions are enclosed within $(\_)^*$, resp. $(\_)^+$. Nonterminal ID represents the identifiers that can be generated using standard ASCII characters; Expr represents typical expressions built from operations on data types, function calls, *etcetera*.

Model	::=	(InterfaceDecl | ComponentDecl)*
InterfaceDecl	::=	**interface** ID { (EventDir ID ID **;**)* (Behaviour)? }
ComponentDecl	::=	**component** ID { (InterfaceDir compoundName ID **;**)* (Behaviour)? }
EventDir	::=	**in** | **out**
InterfaceDir	::=	**provides** | **requires**
Behaviour	::=	**behaviour** (ID)? { (TypeDecl)* (VarDecl)* (FuncDecl)* (BehaviourStmt)* }
TypeDecl	::=	**enum** ID { ID (**,** ID)* }**;**
VarDecl	::=	ID ID **=** Expr **;**
FuncDecl	::=	ID ID **(** (ID ID)? **)**{ (BehaviourStmt)* }
BehaviourStmt	::=	GuardedStmt | CompoundBehaviourStmt | OnEventStmt | ReplyStmt | IllegalStmt | AssignmentBehaviourStmt | ActionStmt | ReturnStmt ConditionalStmt
GuardedStmt	::=	**[** Guard **]** BehaviourStmt
Guard	::=	Expr | **otherwise**
ConditionalStmt	::=	**if** Guard **then** BehaviourStmt **else** BehaviourStmt **;**
CompoundBehaviourStmt	::=	{ (BehaviourStmt)* }
OnEventStmt	::=	**on** OnTrigger **:** BehaviourStmt
ReplyStmt	::=	**reply** (**(**Expr**)**)? **;**
OnTrigger	::=	(EventInstance)$^+$ | **optional** | **inevitable**
EventInstance	::=	ID | ID**.**ID
IllegalStmt	::=	**illegal;**
AssignmentBehaviourStmt	::=	ID **=** Expr **;**
ActionStmt	::=	EventInstance **;**
ReturnStmt	::=	**return** (Expr)? **;**

Example 1. Consider the description of a controller described in Dezyne, given in Fig. 2 (left). Its interface specification (not depicted here), describing the external behaviour the component must comply with, is described by specification IController, as indicated by the provides keyword; it communicates with the 'outside world' via the port called controller. The requires keyword indicates that the controller communicates with a lower-level component, via a port named actuator, behaving in line with the IActuator interface. Events can be received via, or sent via the ports. The behaviour section prescribes the behaviour of the component, indicating, *e.g.* that when s.Off holds (which is shorthand for s == Off) and a start event occurs at port controller (indicated by the on keyword), the component invokes a start event on port actuator, assigns variable s the value State.Init and subsequently returns control via an implicit reply message via the controller port. Also, when s.Off holds, neither a shutdown event via port controller, nor a fail event via port actuator, are permitted; this is indicated by the illegal keyword. □

Using (mutually) recursive functions, one can specify a finite or infinite sequence of statements to be executed upon receiving an event. Recursion is limited to tail recursion [2], allowing for predictable and effective implementations of Dezyne models in standard programming languages such as C and C++. A typical excerpt of a recursive function is given in Fig. 2 (right).

Dezyne allows its users to read and update the values of the variables declared in the variable section of a behaviour in recursive functions. Such manipulations offer a high degree of flexibility to the modeller and are appealing to those accustomed to using iteration rather than recursion. As a consequence, the function g in Fig. 2 (right) sets

```
component Controller
{
      provides IController controller;
      requires IActuator actuator;
      behaviour
      {
            enum State { Off, Init };
            State s = State.Off;
            [s.Off]
            {
                  on controller.start(): { actuator.start(); s = State.Init; }
                  on controller.shutdown(), actuator.fail(): illegal;
            }
            [s.Init]
            {
                  on controller.start(), controller.shutdown(): illegal;
                  on actuator.fail(): { controller.failed(); s = State.Off; }
            }
      }
}
```

```
...
bool b = true;
subint CounterType 0..2;
void f()
{
      i.on(); g(2);
      i.on(); g(2);
}
void g(CounterType c)
{
      if (c == 2 && b){ i.run(); b = false; g(c); }
      else if (c == 0) { i.stop(); }
      else { i.standby(); g(c-1); }
}
...
```

Fig. 2. Left: a DEZYNE model describing a very simple controller. Right: a snippet of a recursive function in DEZYNE; i is a port over which events such as start, stop, on, run and standby are sent.

Boolean b to false so that the second time g is called from f, no run event is emitted from port i. Another way for functions to save part of their computation is to explicitly return a value via a return keyword.

3 An mCRL2 Semantics for DEZYNE

Our formalisation of the DEZYNE methodology includes both a transformation of the core language constructs of DEZYNE to mCRL2, and a sketch of our formalisation of the underlying execution semantics which is used to analyse the compliance of a component to its interface. We first give a cursory overview of the mCRL2 language in Section 3.1, followed by the formalisation of the DEZYNE language in Section 3.2 and its execution semantics in Section 3.3. The implementation and validation of our transformation is briefly discussed in Section 3.4.

3.1 The Process Algebra mCRL2

The mCRL2 language is a process algebra in the lineage of the *Algebra of Communicating Processes* [1]. It consists of a data language for describing data transformations and data types, and a process language for specifying system behaviours. The semantics of mCRL2 processes is given both axiomatically and operationally, associating a labelled transition system to process expressions. For a comprehensive overview of the language, we refer to [5]; for the associated tool set, we refer to [3]; due to page limits, we only informally explain the constructs essential for understanding our work.

The data language includes built-in definitions for most of the commonly used data types, such as Booleans, integers, natural numbers, *etcetera*. In addition, container sorts, such as *lists*, *sets* and *bags* are available. Users can specify their own data sorts using a basic equational data type specification mechanism.

The process specification language of mCRL2 consists of a relatively small number of basic operators and primitives. Since we are concerned with only a fragment of the language we focus on the intuition behind those operators and constructs that are essential for the current exposition. The basic observable events are modelled by parameterised (multi-)actions. Unobservable events are modelled by the constant τ, and the constant δ represents *inaction* (the process that performs no action, colloquially referred to as the deadlock process). Processes are constructed compositionally: the non-deterministic choice between processes p and q is denoted $\mathsf{p+q}$; their sequential composition is denoted $\mathsf{p{\cdot}q}$, and their parallel composition is denoted $\mathsf{p||q}$. A parallel composition of processes may give rise to *multi-actions*: actions that occur simultaneously. A *communication operator* $\Gamma_C(\mathsf{p})$ can map such multi-actions to new actions when their parameters coincide, thus modelling the synchronisation of actions. Using an abstraction operator $\tau_H(\mathsf{p})$, one can turn observable actions into unobservable actions. An *allow operator* $\nabla_A(\mathsf{p})$ can be used to only allow (multi-)actions of the set A that occur in process p.

Recursion can be used to specify processes with infinite behaviour. This is typically achieved by specifying a recursive process of the form $\mathsf{P(v{:}V) = p}$, where P is a process variable, v is a vector of typed variables (where the type is given by V), and p is a process expression that may contain process variables (and in particular variable P). Note that in the next section, we often omit the type V when specifying recursive processes.

Process behaviour can be made to depend on data using the conditional choice operator and a generalised choice operator. The process $\mathsf{b} \rightarrow \mathsf{p} \diamond \mathsf{q}$ denotes a conditional choice between processes p and q: if b holds, it behaves as process p, and otherwise as process q. Process $\sum \mathsf{d{:}D.p(d)}$ describes a (possibly infinite) unconditional choice between processes p with different values for variable d.

Example 2. A simple one-place buffer for natural numbers can be represented by a process $\mathsf{Buffer} = \sum \mathsf{m{:}Nat.read(m)} \cdot \mathsf{send(m)} \cdot \mathsf{Buffer}$, where read and send are actions that represent *storing* a value in the buffer and *loading* a buffered value from the buffer. The process below represents the same behaviour:

$$\begin{aligned}\mathsf{Buffer(n{:}Nat,b{:}Bool)} = \mathsf{b} \;\longrightarrow\; & (\mathsf{send(n){\cdot}Buffer(b = false)}) \\ \diamond\; & \textstyle\sum \mathsf{m{:}Nat.}\ (\mathsf{read(m){\cdot}Buffer(n = m,\ b = true)})\end{aligned}$$

In this alternative formalisation of the buffer, variable b is used to keep track of whether the buffer is filled, and, if so, the value currently stored in the buffer is represented by variable n. Note that $\mathsf{Buffer(b = false)}$ is shorthand notation for $\mathsf{Buffer(n,false)}$; *i.e.* in this notation, only updates to parameters are listed. □

3.2 A Formal Description of the Dezyne to mCRL2 Translation

We mainly focus on the transformation of behaviour statements that occur in Dezyne models to mCRL2; *i.e.* we focus on those statements that correspond to the $\mathsf{BehaviourStmt}$ element in the grammar. We omit details about expressions and type declarations, as these map almost one-to-one on mCRL2 types and data structures.

For our transformation, we assume that every statement s in a concrete Dezyne model has a unique index (*e.g.* a program counter) given by $index(s)$. This index can easily be assigned while parsing the model. Every mCRL2 process equation for a given Dezyne component (resp. interface specification), generated by our transformation, shares the same list v of typed process parameters. This list contains all variables declared in a Dezyne component (resp. interface specification). In particular, it includes all global and local variables of the behaviours, all function parameters and local function variables, and a small number of additional variables that are needed as context for the translation. The list of variables v over-approximates the list of variables that may be in scope at any point in the execution of a component (resp. interface specification). The typed list v can also be constructed while parsing the model. We assume that name conflicts have been resolved using appropriate α-renaming.

Our translation of a behaviour statement s is given by $Tr(s, \mathsf{v}, i, j, g)$, where mapping Tr yields a set of mCRL2 process equations, defined by the rules in Table 2 (for basic statements and events), and in Table 3 (for function statements). Here i is always equal to $index(s)$, and j is the index corresponding to the statement that is executed after termination of s, or -1 if there is no such statement; *i.e.* j points to the next continuation. Each statement s with index i has a corresponding process equation $\mathsf{P}_i(\mathsf{v})$, where v is the list of typed process parameters. The parameter g determines the current scope in which statement s resides; g can either be the name of a function (in which case s is in the function body of g), or it can have the value $\perp$ (in which case s is not in the scope of any function). The actions inevitable, optional and illegal correspond to the triggers and statement with the same name in Dezyne. The parameterised actions snd_r and rcv_r are used to send and receive a value t that is set in a reply(t) statement; the snd_r action marks the end of an on e:s_1 statement. The snd_e and rcv_e actions correspond to sending and receiving of events.

In order to bridge the semantic gap between the Dezyne language and the mCRL2 language, we have added a few statements that are not part of the Dezyne language. A send_reply statement is inserted at the end of each on e: s_1 statement, to make it explicit that the value that is set using a reply(t) statement inside s_1 is eventually returned. In the Dezyne language, sending the reply remains implicit. Dezyne has the requirement that a reply value is set exactly once in an on e:s_1 statement. It is straightforward to extend the translation of Table 2 to check for this by recording the number of executed reply(t) statements in a process parameter. Several other checks, such as *out-of-bounds* checks can be added equally straightforward to our transformation. The choice statement $s_1 \oplus s_2$ and the sequential statement $s_1; s_2$ were introduced to make it explicit that a compound statement that is directly in the scope of an on e:s_1 statement is different from a compound statement inside a behaviour section. The first one acts like a choice between statements, while the latter acts as a sequential composition of statements. Finally the skip statement corresponds to an empty compound statement.

The translation of a behaviour s of a component (resp. an interface specification) is given by $Tr(s, \mathsf{v}_0, i, i, \perp)$, where $i = index(s)$ and v_0 contains the initial values of the global variables of the behaviour, and default values for all other parameters. The continuation variable j is set to i. The effect of this is that the behaviour s will be repeated indefinitely. To reduce the size of the underlying state space, in our implementation of our encoding we reset all non-global variables to their default value at the end of the execution of an on e:s_1 statement.

Table 2. Mapping Tr, describing the translation of (extended) Dezyne statements in normal form to mCRL2 processes and process expressions. Note that we used the convention that $i_1 = index(s_1)$ and $i_2 = index(s_2)$, t is a data expression, b is a Boolean expression, e is an event, x is a variable name, T is a type and $\mathsf{T_x}$ is the type of x. The process parameter r is an element of v and may contain any value t that is set using a reply(t) statement.

statement s	**translation** $Tr(s, v, i, j, g)$
	basic statements
skip	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{P}_j()\}$
$s_1; s_2$	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{P}_{i_1}()\} \cup Tr(s_1, \mathsf{v}, i_1, i_2, g) \cup Tr(s_2, \mathsf{v}, i_2, j, g)$
$\{s_1; s_2; \cdots ; s_n\}$	$Tr(s_1; (s_2; (\cdots ; s_n)), \mathsf{v}, i, j, g)$
$s_1 \oplus s_2$	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{P}_{i_1}() + \mathsf{P}_{i_2}()\} \cup Tr(s_1, \mathsf{v}, i_1, j, g) \cup Tr(s_2, \mathsf{v}, i_2, j, g)$
$\{s_1 \oplus s_2 \oplus \cdots \oplus s_n\}$	$Tr(s_1 \oplus (s_2 \oplus (\cdots ; s_n)), \mathsf{v}, i, j, g)$
if b then s_1 else s_2	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{b} \rightarrow \mathsf{P}_{i_1}() \diamond \mathsf{P}_{i_2}()\} \cup Tr(s_1, \mathsf{v}, i_1, j, g) \cup Tr(s_2, \mathsf{v}, i_2, j, g)$
x = t	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{P}_j(\mathsf{x} = \mathsf{t})\}$
T x = t	$Tr(\mathsf{x} = \mathsf{t}, \mathsf{v}, i, j, g)$
illegal	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{Illegal}()\}$ where $\mathsf{Illegal}(\mathsf{v}) = \mathsf{illegal} \cdot \mathsf{Illegal}()$
	event related statements
[b] s_1	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{b} \rightarrow \mathsf{P}_{i_1}() \diamond \delta\} \cup Tr(s_1, \mathsf{v}, i_1, j, g)$
on e:s_1	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{rcv\_e(e)} \cdot \mathsf{P}_{i_1}()\} \cup Tr(s_1, \mathsf{v}, i_1, j, g)$
reply(t)	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{P}_j(\mathsf{r} = \mathsf{t})\}$
send_reply(e)	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{snd\_r(e, r)} \cdot \mathsf{P}_j()\}$
x = e	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{snd\_e(e)} \cdot \sum \mathsf{x}' : \mathsf{T_x}.\mathsf{rcv\_r(e, x')} \cdot \mathsf{P}_j(\mathsf{x} = \mathsf{x}')\}$
e	$\begin{cases} \{\mathsf{P}_i(\mathsf{v}) = \mathsf{snd\_e(e)} \cdot \mathsf{rcv\_r(void)} \cdot \mathsf{P}_j()\} & \text{if e is an `in' event from a required port} \\ \{\mathsf{P}_i(\mathsf{v}) = \mathsf{snd\_e(e)} \cdot \mathsf{P}_j()\} & \text{otherwise} \end{cases}$

Example 3. We exemplify the translation on a small part of the Dezyne model of Fig. 2 (left), using fictitious numbers as statement indices. We assume that all events are void events, meaning that these do not return a value.

$$
\begin{array}{ll}
\mathsf{Controller_1(s{:}State)} & \mathsf{= Controller_2(s) + Controller_{12}(s);} \\
\mathsf{Controller_2(s{:}State)} & \mathsf{= (s == Off) \longrightarrow Controller_3(s)} \diamond \delta; \\
\mathsf{Controller_3(s{:}State)} & \mathsf{= Controller_4(s) + Controller_8(s);} \\
\mathsf{Controller_4(s{:}State)} & \mathsf{= rcv\_e(controller.start) \cdot Controller_5(s);} \\
\mathsf{Controller_5(s{:}State)} & \mathsf{= snd\_e(actuator.start) \cdot rcv\_r(void) \cdot Controller_6(s);} \\
\mathsf{Controller_6(s{:}State)} & \mathsf{= Controller_7(s = Init);} \\
\mathsf{Controller_7(s{:}State)} & \mathsf{= snd\_r(controller.start, void) \cdot Controller_1(s);} \\
\mathsf{Controller_8(s{:}State)} & \mathsf{= Controller_9(s) + Controller_{11}(s);} \\
\mathsf{Controller_9(s{:}State)} & \mathsf{= rcv\_e(controller.shutdown) \cdot Controller_{10}}(s); \\
\mathsf{Controller_{10}(s{:}State)} & \mathsf{= Illegal();} \\
\mathsf{Controller_{11}(s{:}State)} & \mathsf{= rcv\_e(actuator.fail) \cdot Controller_{10}}(s); \\
\mathsf{Controller_{12}(s{:}State)} & = \ldots \\
\ldots & \\
\mathsf{Illegal(s{:}State)} & \mathsf{= illegal \cdot Illegal();}
\end{array}
$$

Note that the actual typing information for the events would be specified in the interface specifications IController and IActuator, referred to in (but not detailed in) Fig. 2 (left). Furthermore, observe that equation $\mathsf{Controller_7}$ deals with the send_reply statement which is not part of the Dezyne language, but which we need to include to signal the end of an on-event statement. □

Formalising the recursive functions of the DEZYNE language proved to be the most involved part of the translation as it required several iterations to find a translation that had a good enough performance for some industrial cases with thousands of deeply nested function calls. One of the complications is that functions can modify the global variables of a behaviour. In our first attempt, we handled these modifications using a separate register process, but it turned out that the additional communication needed for this could cause an unacceptable blow up of the state space for some examples.

Our final solution was to introduce a process parameter c that contains the function call stack, and process parameters $\mathsf{rvar_T}$ for each function return type T that contain function call results. Both c and $\mathsf{rvar_T}$ are elements of the list of variables v we maintain in our translation. In each return statement of a function with return type T, the function result is stored in the parameter $\mathsf{rvar_T}$. In an assignment statement $\mathsf{x{=}f(t)}$, the function result is retrieved from this parameter $\mathsf{rvar_T}$. We ensure that each function body is translated only once. At first sight this may seem problematic, since the translation of a function call depends on the statement where the execution should continue after termination, which is encoded in the parameter j. This problem has been solved by moving the actual mapping of a function call statement with index i to the corresponding continuation j in a separate Return process. The Return process contains a summand $(\mathsf{c} \neq [] \wedge \mathsf{head(c)} = i) \longrightarrow \mathsf{P}_j(\mathsf{c{=}tail(c), x{=}rvar_T}) \diamond \delta$ for each assignment statement $\mathsf{x{=}f(t)}$ with index i. Note that the indices of the function call statements are stored in the function call stack c. In case of a nested function call between mutually dependent tail-recursive functions, it is known that the continuation statement will not change. So in this particular case we do not add the index of the statement to the function call stack c. We determine whether functions are mutually dependent by checking that they are in the same strongly connected component of the function call graph. The restriction to tail-recursive functions ensures that it is not needed to put copies of local function variables on the stack, see *e.g.* [2]. Details of the formalisation of function call statements can be found in Table 3. For completeness, the translation $Tr(s_f, \mathsf{v}, i_f, -1, f)$ of a function body s_f is added to the translation of each function call $\mathsf{f(t)}$. In our implementation it is generated only once. Note that the continuation parameter j is set to the undefined value -1, since the actual continuation value of a function call is stored in the Return process.

Table 3. Mapping *Tr*, describing the translation of DEZYNE function calls and returns in mCRL2. Note that t is a data expression, s_f is the body of function f, $i_f = index(s_f)$, and d_f is the function parameter of function f. By $\mathsf{c} = i \triangleright \mathsf{c}$ we denoted that index i is prepended to list c.

statement s	**translation** $Tr(s, \mathsf{v}, i, j, g)$
function call statements	
f(t)	$\begin{cases} \{\mathsf{P}_i(\mathsf{v}) = \mathsf{P}_{i_f}(\mathsf{d}_f = \mathsf{t})\} & \text{if } f \text{ and } g \text{ are mutually dependent} \\ \{\mathsf{P}_i(\mathsf{v}) = \mathsf{P}_{i_f}(\mathsf{d}_f = \mathsf{t}, \mathsf{c} = i \triangleright \mathsf{c})\} & \text{otherwise} \end{cases}$ $\cup\ Tr(s_f, \mathsf{v}, i_f, -1, f)$
x=f(t)	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{P}_{i_f}(\mathsf{d}_f = \mathsf{t}, \mathsf{c} = i \triangleright \mathsf{c})\} \cup Tr(s_f, \mathsf{v}, i_f, -1, f)$
return t	$\{\mathsf{P}_i(\mathsf{v}) = \mathsf{Return}(\mathsf{rvar}_T = \mathsf{t})\}$ where T is the return type of f

3.3 Formalising the Execution Model

Dezyne models that are converted to executable code and subsequently deployed interact with other components following a run-to-completion regime which is guaranteed by the Dezyne code generation. A faithful analysis of the behaviour of Dezyne components therefore requires a formalisation of this execution model in mCRL2. This holds particularly true for the compliance test that is conducted, which essentially checks whether the behaviour of a component C, as can be observed from its provided port p, complies with the behaviour that is specified by C's interface specification. Formally, the compliance check decides whether or not the labelled transition system underlying the behaviour of C (when interacting with other components through its required ports r_1 up to r_n, see also Fig. 1) is a correct failures-divergence refinement [6] of the labelled transition system underlying the behaviour of C's interface specification. Relying on an assume-guarantee style of reasoning, the behaviours of the components that C interacts with through ports r_1 up to r_n, are represented by their respective interface specifications (and their underlying labelled transition systems) in all analyses of the behaviour of C in the Dezyne tool set.

Conceptually, the run-to-completion execution model ensures that component C, when interacting with other components through C's port p and ports r_1 up to r_n, is blocked for unsolicited external stimuli as long as it has not finished dealing with a previous stimulus. External stimuli that come via the required ports are queued in a queue Q. This is not the case for the replies to events submitted to a component via a required port. Unsolicited stimuli arriving at a required port are announced by an optional or inevitable trigger. The execution model furthermore defines the semantic difference between the latter two triggers, by non-deterministically deciding at any point in the execution of C's behaviour that optional triggers become disabled, whereas inevitable triggers cannot be disabled. Such nuances make the effect of the execution model on the interactions between components non-trivial.

Rather than presenting our mCRL2 formalisation of the run-to-completion semantics, we explain its workings using a high-level state diagram of a part of this formalisation, see Fig. 3. The diagram represents how unsolicited stimuli arriving via the provided port are dealt with; the part dealing with unsolicited stimuli arriving via the required port (initiated by an optional or inevitable trigger, which fills buffer Q) is largely the same but lacks, *e.g.* transitions dealing with sending reply values to the events taken from the queue. The execution model enforces that stimuli at the provided port and optional and inevitable triggers at the required ports are only accepted in state 'Idle' of Fig. 3. In mCRL2, this can be modelled by a blocking synchronisation on actions such as rcv_e, optional and inevitable, using a combination of mCRL2's parallel composition operator ||, its communication and restriction operator and its renaming operator.

The state diagram of Fig. 3 illustrates the flow of events when a stimulus via the provided port arrives. This causes a state change, leading to state 'Processing'. When the component reports that it has finished processing the event (indicated by the snd_r(e,v) action, which sets a value for reply variable r) it moves to state 'Finishing'. Once the component is in state 'Finishing', it will start processing the solicited events that may have arrived in the queue in the meantime. Executing an event e′ from the queue (indicated by the rcv_e(e′) action) takes the state diagram to state 'Finished Blocked'; when the

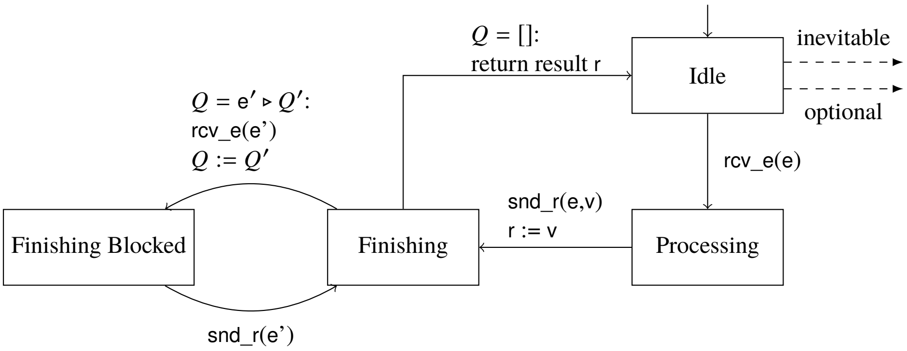

Fig. 3. Schematic overview of the run-to-completion semantics of Dezyne components.

component reports it is finished processing this event (indicated by the snd_r(e′) action), it returns to state 'Finished'. When the queue is finally empty, the component again returns to the 'Idle' state and returns the value stored in variable r that was determined during the execution of event e. In all non-'Idle' states the component may send out events via its provided port or via its required ports, and, in response to such events, other components may fill the queue with new events; we have omitted these self-loops from the diagram for simplicity.

3.4 Implementing and Validating the Transformation

The model transformation has been implemented using Python. The input of our transformation is a Dezyne model stored in Scheme format. The Scheme file is parsed into a Python class model of a Dezyne model, to which our generator is applied. The result is a Python class model of an mCRL2 model. This mCRL2 model is then pretty printed to text format, after which the mCRL2 tools are applied for further analysis.

Our preference for the general purpose programming language Python over a specialised model transformation language such as, *e.g.* QVTo, is motivated by the need to easily make changes to the generator. A scripting language like Python is ideal for that. Since there is a large gap between the Dezyne language and the process algebra mCRL2, it was clear from the start that the main effort would be to experiment with different ways to do the transformation. The generator and its supporting data structures have been revised many times. What also helped to support making changes is that we made specifications of the translation in an early stage, and kept it in sync with the implementation, ultimately resulting in the specifications of Tables 2 and 3.

Note that the class models of Dezyne and mCRL2 were stable from the start. The classes were kept very simple, and correspond in a one to one way with UML metamodels of both languages. The mCRL2 classes could even be generated from an input file containing merely 150 lines of text.

We validated the relative correctness of our transformation using a set of test cases provided by Verum, consisting of 168 component models and 224 interface models, including several models taken from industry (see also Section 5). For all these cases we were able to establish that the state spaces of the behaviours of the components using our transformation and Verum's transformation were strongly bisimilar. Moreover, using the mCRL2 tool set we could reproduce the outcomes to all checks currently performed by Dezyne on components, interfaces and their interactions under the run-to-completion semantics on these test cases.

4 Improvements and Enhancements in mCRL2

As the previous section illustrates, from a language point of view, the mCRL2 language is sufficiently expressive for describing the Dezyne models and its execution semantics. This opens up the possibility to analyse Dezyne models using the mCRL2 tool set.

The mCRL2 tool set works by parsing, type-checking and subsequently converting an mCRL2 specification to a normal form called a *Linear Process Specification* (LPS). All analyses of the mCRL2 specification are subsequently performed by tools operating on LPSs or its derived artefacts such as state spaces. Analysing the mCRL2 models obtained by translating large Dezyne models developed in industry led to several feature requests for various tools in mCRL2 but also revealed a few bottlenecks and a thus far undiscovered error in the mCRL2 tool set.

A major enhancement to the mCRL2 tool set concerns the addition of algorithms for deciding several types of refinement relations. This was needed to properly deal with Dezyne's verification methodology which relies on an assume-guarantee style of reasoning rooted in the notion of *failures-divergence refinement* [6]. While this notion is one of the hallmark features of the FDR tool set (in fact giving it its name), mCRL2 did not support this refinement notion, and it could not be mimicked by any of the many behavioural equivalences that *were* supported by mCRL2. An *anti-chain*-based algorithm, based on [7], for deciding failures-divergence refinement was added to the mCRL2 tool `ltscompare`.[4] Another enhancement to the tool set concerns the generation of witnesses to divergences—infinite sequences of internal actions—and the generation of counterexamples for failures-divergence refinement and other refinement relations.

The larger Dezyne models we ran as test cases revealed that mCRL2 was not optimised for dealing with the immense number of recursive process equations obtained from our automated translation. While the complexity of each individual equation was low (some equations just refer to other equations, *e.g.* when translating assignments), the vast number of these equations meant that some basic parts of the algorithms used to convert mCRL2 processes to LPSs needed improvement. Examples include the removal of a linear search through a list of global data variables and the addition of routines to merge similar equations. In particular, *alphabet reduction*, a preprocessing step of linearisation that analyses possible occurrences of multi-actions, has been improved in

[4] The option to check for this refinement relation, and other refinement relations such as *trace inclusion*, *weak trace inclusion*, *failures*, *weak failures* and *simulation preorder* is available from mCRL2 revision 13875 and onward. The additions weigh in at approximately 800 lines of code, which include, among others the additional algorithms and test cases for these algorithms.

a number of ways. Due to the occurrence of large blocks of interdependent equations, it turned out to be necessary to cache the alphabet of such equations. Also the sets of possible multi-actions needed to be pruned more aggressively, to deal with their huge sizes. At the same time, an error in the rules underlying the old alphabet reduction algorithm surfaced, which was subsequently fixed.

5 Experiments

In the course of formalising Dezyne in mCRL2, we have experimented with several different but semantically equivalent (modulo divergence-preserving branching bisimulation) translations. The main criterion, next to correctness, used in our search for a proper formalisation was the scalability of verifying the mCRL2 models resulting from a translation. Typical verifications that are offered by the Dezyne tool set, and which can be conducted by analysing the appropriate mCRL2 model obtained from translating a Dezyne model, are absence of deadlock and livelock, out-of-bound checks for variables, invoking events that are marked illegal, and interface compliance of components. As we mentioned before, the latter verification is essentially a check whether the behaviour as can be observed at the provided port of a component is a correct failures-divergence refinement of the behaviour as specified by the interface specification.

While it can be expected that the various ways of formalising a language will have an effect on the size of the underlying labelled transition systems of concrete Dezyne models, we had initially not expected the effects to be so dramatic. In fact, for small examples, the effects were marginal, but for the models developed in the industry, the effects were surprisingly big. This was particularly true for the compliance checks, which are computationally the most expensive checks carried out by the Dezyne tool set: the check requires computing a labelled transition system that represents the interaction between a component and the interface specifications for its required ports, given the execution model of Section 3.3. To illustrate the differences in scalability for the compliance check, we compare the effect (on time and state space size) of translating functions using a dedicated register process for recording the side effects functions can have on global variables and the translation described in Section 3.2, see Table 4.[5] These results clearly indicate that one can easily gain a factor 5 or more for the larger models in terms of speed by choosing an appropriate translation. This also holds for the other types of verification that can be conducted.

It is noteworthy that the verification times we obtain using the mCRL2 model are currently roughly 2-5 times slower than the verification times reported by Verum on the same models. This difference may be due to hardware differences, but we expect that FDR's different state space exploration technique is a main factor, which explores and minimises individual parallel processes before combining these, whereas mCRL2 explores a monolithic model. Indeed, manually mimicking FDR's compositional approach in mCRL2 shows an additional speed-up of a factor 5-10 can be achieved.

Finally, we note that the translation to mCRL2 opens up the possibility to use advanced technology for visually inspecting state spaces and tools to verify more complex

[5] Unfortunately, we cannot disclose the origin of, nor further details about these industrial models.

Table 4. The effect on the size of the state space and the time to generate the transition system and run the compliance test when translating Dezyne functions using either a dedicated register process for recording side effects on global variables (translation I) and when translating functions using the rules in Section 3.2 (translation II). Time is in seconds; a dash indicates that the computation did not finish within the available time or memory. The models are embedded software control models, developed (and deployed) in industry using Verum's software engineering tool suite. The lines of code for mCRL2 correspond to translation II.

Model	Time (s)		Speedup	# States		Reduction	Lines of code	
	I	II		I	II		Dezyne	mCRL2
Model 1	155	13	11	715, 049	110, 773	6	3, 133	2, 157
Model 2	83	13	6	984, 167	43, 281	22	2, 808	3, 616
Model 3	37	10	3	33, 488	6, 700	4	2, 382	2, 838
Model 4	27	11	2	822	226	3	2, 904	2, 482
Model 5	45	11	4	443, 379	182, 367	2	1, 751	2, 114
Model 6	135	17	7	1, 039, 654	323, 023	3	4, 145	3, 114
Model 7	–	18	–	–	74, 654	–	4, 328	3, 161
Model 8	–	21	–	–	101, 948	–	4, 931	4, 434
Model 9	–	35	–	–	215, 727	–	5, 721	4, 645
Model 10	2, 069	275	7	36, 140, 140	10, 967, 862	3	8, 169	8, 474

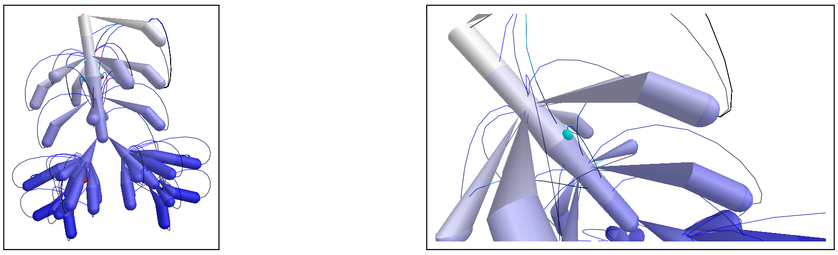

Fig. 4. Visualisations of the state space underlying an interface specification used in 'Model 10'. The symmetry in the two branches at the bottom in the left picture is a telltale sign of symmetry in the behaviour of the interface specification.

properties than the generic ones currently offered by the Dezyne verification tool set. For instance, for 'Model 10', which models a complex piece of software control in an embedded device of one of Verum's customers, we have verified typical properties relevant in this context such as:

- Invariantly, whenever the system receives an initialisation event, it remains possible to successfully stop production;
- There is an infinite execution in which production is never stopped;
- It is impossible to initialise the system when it is already initialised unless production is stopped.

Such properties are expressed in mCRL2's modal μ-calculus with data, and all three properties listed above are readily verified to hold on 'Model 10'. Moreover, we have verified several liveness properties that are true of the interface specification of 'Model 10' but not of the component itself. Through such properties, the relation between a component and its interface specification can be better understood.

Fig. 4 depicts a graphical simulation of a 3D depiction of the state space of one of the interface specifications used in ‘Model 10’, giving an impression of the type of visualisations that one can use to inspect the state space. Such a visualisation help to, *e.g.* confirm expectations (such as an expected symmetry in the system behaviour).

6 Concluding Remarks

Modelling languages used in the context of model driven engineering have gained traction among industry over the last years. Such languages are predominantly used to generate executable code, but tool sets supporting these languages rarely offer forms of formal verification of the models. The Dezyne language and associated tool set, developed by Verum, is one of these rare exceptions, with formal verification support offered through a non-documented, proprietary mapping to the FDR tool set [4].

We have described a formalisation of the Dezyne language in terms of mCRL2 [5], providing a first publicly accessible formal semantics of Dezyne models and their execution semantics. The formalisation and implementation of the transformation, which was developed in a period of 2 years and took well in excess of 1 man-year of effort, led to improvements and additions in both mCRL2 and the existing Dezyne to FDR translation, and served as an independent validation of the ideas behind the methodology behind Dezyne. Moreover, the transformation we developed is a first step to adding more advanced verification and visualisation possibilities to the Dezyne tool set.

Acknowledgements. Wieger Wesselink and Tim Willemse were funded by the EU-FP7 TTP VICTORIA project (project grant agreement 609491).

References

1. J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2010.
2. W.D. Clinger. Proper tail recursion and space efficiency. In *PLDI*, pages 174–185. ACM, 1998.
3. S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, W. Wesselink, and T.A.C. Willemse. An overview of the mCRL2 toolset and its recent advances. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2013.
4. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A.W. Roscoe. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, 18(2):149–167, 2016.
5. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.
6. A.W. Roscoe. On the expressive power of CSP refinement. *Formal Asp. Comput.*, 17(2):93–112, 2005.
7. T. Wang, S. Song, J. Sun, Y. Liu, J.S. Dong, X. Wang, and S. Li. More anti-chain based refinement checking. In *ICFEM*, volume 7635 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2012.