Sarir: A $\mathcal{R}$ebeca to mCRL2 Translator

H. Hojjat[1], M. Sirjani[1], M.R. Mousavi[2,3] and J.F. Groote[2]
(1) University of Tehran and IPM, Iran (2) Eindhoven University of Technology (TU/e), The Netherlands
(3) Reykjavík University, Iceland

Abstract

We describe a translation from $\mathcal{R}$ebeca, an actor-based language, to mCRL2, a process algebra enhanced with data types. The main motivation is to exploit the verification tools and theories developed for mCRL2 in $\mathcal{R}$ebeca. The mapping is applied to several case-studies including the tree identify phase of the IEEE 1394 standard. The results of the experiment show that the minimization tools of mCRL2 can be very effective and the outcome of the present translation outperforms that of the translation to the input language of the Spin model-checker.

1 Introduction

In this paper, we present a mapping from $\mathcal{R}$ebeca [12, 13] to mCRL2 [8] and introduce a tool, Sarir[1], which automatically carries out this translation. Two sides of the mapping are languages with a formal semantics. The source of the translation, is an object based language having its roots in the actor model [1]. The target of the translation is a process algebraic language: mCRL2, a successor to μCRL [6]. An important goal in the design of μCRL is to provide a common language for analysis of system behavior, hence the name 'micro Common Representation Language'.

At first glance, $\mathcal{R}$ebeca and mCRL2 languages seem to be inherently different. $\mathcal{R}$ebeca uses asynchronous message passing whereas communication in mCRL2 is synchronous. The events that constitute the state space in a $\mathcal{R}$ebeca program is the passing of messages between rebecs (reactive objects) and changing the internal state of the rebecs, but in mCRL2 the state space is generated by issuing atomic actions. Throughout the paper we briefly describe how these dissimilar phenomena can be mapped together. The main interest of this mapping is to investigate the applicability of various reduction and analysis tools available for mCRL2. We are also working to translate the existing and highly effective reduction techniques for mCRL2 [7] to the setting of $\mathcal{R}$ebeca.

There are a fair number of attempts that addressed the connections between actor and process algebra, most notably Aπ calculus [2] which is a typed asynchronous variant of the π-calculus. Our aim here is different from that of [2]; the main concern in [2] is semantics while here our main concern is verification. For our purpose, incorporated data types of mCRL2, such as integer, boolean and list are very handy and simplify our translation while in [2], all data types have to be defined using basic constructs of the π-calculus.
To have a better insight into the translation and its effectiveness we carried out several case studies including the tree identify phase of the IEEE 1394 firewire. The results are encouraging and show significant improvement (esp. in large case-studies) over an existing translation into Promela [9], the input language of the Spin model-checker.

2 Background

In this section, we briefly introduce the source and the target of our translation.
$\mathcal{R}$ebeca. A $\mathcal{R}$ebeca (Reactive Object Language) model contains a set of concurrent objects, rebecs instantiated from reactive classes. Rebecs may communicate only by sending and receiving asynchronous messages, and based on the received messages react to the environment. Each rebec has an infinite mail queue in which the incoming messages are enqueued. A typical example of a $\mathcal{R}$ebeca model is depicted in Figure 1.
$\mathcal{R}$ebeca uses a Java-like syntax. There are two main declaration parts in the reactive class definition: The **knownobjects** and **statevars**. The **knownobjects** part includes the reactive classes to which this reactive class is allowed to send messages. The **statevars** part declares variables that store the local state of the

[1]Sarir is a binary star in the constellation Ursa Major. The tool can be obtained via the Rebeca home page: `http://khorshid.ece.ut.ac.ir/~rebeca/`.

```
reactiveclass Example {
    knownobjects {
      Example ex;}
    statevars {
      int counter; }
    msgsrv initial() {
      ex.add(0);}
    msgsrv add(int a)   {
      if ( counter < 100)
        {counter = counter + 1;}   }   }
    main {
      Example ex1( ex2):();
      Example ex2( ex1):();  }
```

Figure 1. A simple $\mathcal{R}$ebeca model

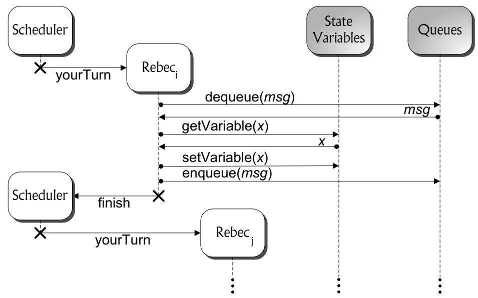

Figure 2. MSC of $\mathcal{R}$ebeca model in mCRL2

rebec. In addition to the known objects and the valuation of the variables, the local state of a rebec (in its operational semantics) includes the content of the inbox queue, i.e., its received messages. When a message is dequeued, its corresponding method is atomically executed, i.e., statements of concurrent methods cannot be interleaved. If a number of concurrent rebecs in the model have a message in their queues, messages are chosen for execution by a nondeterministic scheduler. Every reactive class definition has a method named *initial*. In the initial state, each rebec has an initial message in its queue. Available constructs for the syntax of message servers are assignment, message sending, conditional statements and sequential composition.

mCRL2. The specification formalism μCRL extends the Algebra of Communicating Processes (ACP) [3] with abstract data types. mCRL2 is a successor to μCRL, and includes features such as true concurrency (in terms of multi-actions), real time, higher-order functions and concrete data types. We refer to [8] for a more elaborate description of the new features.

$$p ::= a(d_1, \ldots, d_n) \mid \tau \mid \delta \mid p{+}p \mid p{\cdot}p \mid p \parallel p \mid \tau_I(p) \mid \partial_H(p) \mid \nabla_V(p) \mid \Gamma_C(p) \mid \textstyle\sum_{d:D} p \mid c \to p \diamond p$$

The summarized syntax of a process in mCRL2 is given by the above grammar. A basic action a of a process may have a number of arguments $d_1, \ldots, d_n$. These arguments correspond to the data elements. There are two designated basic actions τ and δ which do not take any parameter. Action τ denotes an internal action. Internal actions, as their name suggests, cannot be observed from the external world. Process δ denotes the deadlock process in which no further transition is possible. Non-deterministic choice between two processes is denoted by the "+" operator. Processes can be composed sequentially and in parallel by means of "$\cdot$" and "$\parallel$". It is often useful to abstract from some actions in the system and declare them as internal actions. This is performed by the abstraction operator $\tau_I(p)$. The subscript I is the set of actions that are to be hidden. To enforce synchronization, the encapsulation operator $\partial_H(p)$ specifies the set of actions H which are not allowed to occur. Reversely, the allow operator $\nabla_V(p)$ indicates the actions that are only allowed to occur. To show possible communications in a system and the resulting actions, communication operator $\Gamma_C(p)$ is used. The elements of set C are of the form $a_1 \mid a_2 \mid \cdots \mid a_n \to c$ (for $n \geq 0$), which intuitively means that action c is the result of the multi-party synchronization of actions a_1, a_2, ..., and a_n.

There are a number of built-in data types in mCRL2, such as (unbounded) integers, (uncountable) reals, lists, sets and functions. In particular, the list data type is quite useful for us. Concatenation($++$), head extraction (*head*, *rhead*) and tail extraction (*tail*, *rtail*) are some of the predefined operations on lists.

3 Translation: Mapping Sketch

There are two main factors responsible for constructing the local state of a rebec: valuation of local variables and the contents of the inbox q that stores the incoming messages [12]. The rebec's message servers are means for manipulating and changing the states of rebecs. The execution of message servers take place in an atomic step, i.e., the execution of a statement from one message server cannot be interleaved with the execution of another statement from another message server. So, the set of rebecs can be viewed as a set of coarse grained parallel components. This helps us to replace the concurrent behavior of the rebecs with nondeterministic sequential interleaving, and also reducing the number of generated processes. Instead of introducing a set of processes for rebecs and run them in parallel, we take the advantage of atomic execution of message servers to use a single enabled process for each message server being executed. Figure 2 shows a message sequence chart presenting an abstract view of the translation. In this figure, each box represents an mCRL2 process. Gray boxes (Queues and State variables) are always present in the system but from the white boxes (Scheduler and Rebecs) at all times only one instance exists.

There are two processes that hold the state of the program: one of them for the state variables and the other for the internal queues. These two processes always exist in the system and the state of the model can be retrieved and changed by exchanging messages with them. Furthermore, there is a scheduler process in the system.

The scheduler should determine which rebec has the turn to run. For this purpose it non-deterministically chooses a rebec and starts the process corresponding to that rebec. The rebec process starts running by dequeueing a message from the queue process. The message requires the rebec process to perform a series of actions, which include getting and setting variables and sending messages to other rebec processes. After completing the message server task, the rebec process informs the scheduler process and it finishes. The scheduler continues the overall execution by again initiating a rebec. The sequence of creating and destroying rebec processes will continue forever.

The mCRL2 translation of the $\mathcal{R}$ebeca code in Figure 1 is presented in Figure 3. Due to lack of space, we cannot give a complete explanation of the translation rules in this paper.

Data Types. In the first part, new data types are defined. *Message* is defined as a triple: sender, message and receiver. *Queue* is a list of messages. The rebec names that are used in the model are included in *Rebecs*, with two default rebecs: *sched* and *null* (the former is responsible for sending the initial messages and the latter is used for the sender of the dummy null message marking the end of a queue). The last data type definition is *MsgType*, which contains the message server names of the model. If a message server has parameters, we import the parameters in the mCRL2 code by introducing a set of arguments for that message server definition in *MsgType*. For example, the message server add(int a) is translated to $add\_msg(a : Int)$. The message server *null_msg* is used in null messages.

State Variables. Each state variable is represented by a parameter of the state variable process. There are two rebecs in our example: ex1 and ex2, each of which has one state variable. So, the state process has two parameters, x_0 and x_1. The state variable process offers two actions for each of its parameter, one for getting the value and one for setting a new value to it: r_get_counter and r_set_counter. To set the variable *counter* of the rebec $ex1$ to a new value $counter'$, action s_set_counter ($r2mInt(counter', ex1)$) should be issued. For getting the value, one should issue the ac-

```
%predefined types
sort R2MBool = struct r2mBool(value : Bool, rebec : Rebecs);
sort R2MInt = struct r2mInt(value : Int, rebec : Rebecs);
sort Message = struct msg(sender:Rebecs,
                          sentMessage:MsgType, receiver:Rebecs);
sort Queue = List(Message);
sort Rebecs = struct ex1 | ex2 | sched | null;
sort R2MQueue = structr2mQueue(value : Queue, rebec : Rebecs);
sort MsgType =struct initial_msg | add_msg(a:Int) | null_msg;
```

```
%state variables
act set_counter, r_set_counter, s_set_counter
    get_counter, r_get_counter, s_get_counter:R2MInt;
proc state(x0:R2MInt, x1:R2MInt) =
    ∑_{y:Int} r_set_counter(r2mInt(y, rebec(x0)))·
              state(r2mInt(y, rebec(x0)), x1)+
    r_get_counter(x0)·state(x0, x1)+
    ∑_{y:Int} r_set_counter(r2mInt(y, rebec(x1)))·
              state(x0, r2mInt(y, rebec(x1)))+
    r_get_counter(x1)·state(x0, x1);
```

```
%internal queues
act enqueue, r_enqueue, s_enqueue,
    dequeue, r_dequeue, s_dequeue:Message;
proc queue(q0:R2MQueue, q1:R2MQueue) =
    ∑_{mt:MsgType,sr:Rebecs} r_enqueue(msg(sr, mt, rebec(q0)))·
     queue(r2mQueue([msg(null, null_msg, rebec(q0)),
       msg(sr, mt, rebec(q0))]++tail(value(q0)), rebec(q0)), q1)+
    (r_dequeue(rhead(value(q0)))·
     ((value(q0) == [msg(null, null_msg, rebec(q0))])
       →queue(q0, q1)
       ◇ queue(r2mQueue(rtail(value(q0)), rebec(q0)), q1)))+
    ∑_{mt:MsgType,sr:Rebecs} r_enqueue(msg(sr, mt, rebec(q1)))·
     queue(q0, r2mQueue([msg(null, null_msg, rebec(q1)),
        msg(sr, mt, rebec(q1))] ++tail(value(q1)), rebec(q1)))+
    (r_dequeue( rhead(value(q1)))·
     ((value(q1)==[msg(null, null_msg, rebec(q1))])
     →queue(q0, q1)
      ◇ queue(q0, r2mQueue(rtail(value(q1)), rebec(q1)))));
```

```
%rebecs definition
proc rebec(r:Rebecs, ko_ex:Rebecs) =
    ∑_{mt:MsgType,sr:Rebecs} s_dequeue(msg(sr, mt, r))·
     (mt== initial_msg)
       →(s_enqueue(msg(r,add_msg(0), ko_ex)))·Scheduler
       ◇(∑_{a:Int}(mt== add_msg(a))
          → (∑_{counter:Int} s_get_counter(r2mInt(counter, r))·
           (counter < 100)
            → (∑_{counter:Int}
                s_get_counter(r2mInt(counter, r)).
                s_set_counter(r2mInt((counter+1), r)))
            ◇τ)
         )·Scheduler;
```

```
proc Scheduler = rebec(ex1, ex2)+rebec(ex2, ex1);
```

```
init ∇_{set_counter, get_counter, enqueue, dequeue}(
    Γ_{r_set_counter|s_set_counter→set_counter,
      r_get_counter|s_get_counter→get_counter,
      r_enqueue|s_enqueue→enqueue,
      r_dequeue|s_dequeue→dequeue}
    state(r2mInt(0, ex1), r2mInt(0, ex2)) ||
    queue(r2mQueue([msg(null, null_msg, ex1),
            msg(sched, initial_msg, ex1)], ex1),
          r2mQueue([msg(null, null_msg, ex2),
            msg(sched, initial_msg, ex2)], ex2)) ||
    Scheduler));
```

Figure 3. mCRL2 translation of Figure 1.

tion $\sum_{z:Int}$ $\mathsf{s\_get\_counter}(r2mInt(z, ex1))$.

Queues. A rebec has an internal queue, in which the incoming messages are enqueued, and when it is given a turn, it dequeues a message from this queue for execution. A message is a triple: $< sendid, i, mtdid >$, where *sendid* is the identifier of the sender, i is the identifier of the receiver, and *mtdid* denotes the message server which is intended to be called [12]. Two rebecs exist in our running example, so queue has the two parameters q_0 and q_1. For each queue variable two actions of enqueuing and dequeuing are available. If a rebec process wants to dequeue a message from $ex1$, it issues $\sum_{mt:MsgType, sr:Rebecs}$ $\mathsf{s\_dequeue}(msg(sr, mt, ex1))$, and for enqueuing the message $add\_msg(0)$ to the $ex1$ queue, it issues $\mathsf{s\_enqueue}(msg(r\_i, add\_msg(0), ex1))$ in which r_i denotes the name of the sender rebec.

Messages are enqueued from the left, and are dequeued from right. There is always a null message (with *null* as its sender and *null_msg* as its message type) at the leftmost position of the queue, indicating the queue end.

Rebecs. A rebec process dequeues a message from the top of its queue and runs its corresponding message server. The arguments of the process are the name of the rebec and its known rebecs (r and ko_ex in our example). Message $< sr, mt, r >$ is dequeued from the top of the queue. Variable sr denotes the sender of the message, and mt is the message type. Message type mt, determines the behavior of the rebec when servicing the message. (Sequential composition in $\mathcal{R}$ebeca ";" is simply translated into sequential composition in mCRL2 "·".) Assignment and message sending are the statements that result in state transitions. One important distinction between these two types of statements is that an assignment can solely change the local state of a rebec but by sending messages a rebec modifies the state of another rebec. The mCRL2 conditional operator is similar to the common if-then-else structure. In an expression $c \rightarrow p \diamond q$, if condition c is evaluated to true then p, and otherwise q, is executed. The conditional statement of $\mathcal{R}$ebeca can thus be mapped directly to mCRL2 conditionals. In mCRL2 if the "else" component is not stated ($c \rightarrow q$) it is automatically assumed to be deadlock, i.e., in an untimed setting $c \rightarrow q$ is an abbreviation for $c \rightarrow p \diamond \delta$ (with time it is $c \rightarrow p \diamond \delta$ˁ$0$). If a conditional statement in $\mathcal{R}$ebeca does not contain "else" we ought to put a τ action for the else branch to prevent deadlocks. For the Boolean condition first the variables are loaded and then the condition is translated. The translation of p and q is defined inductively, based on the translations of the individual statements.

$\mathcal{R}$ebea construct	Promela construct
reactiveclass	proctype
rebec	process
knownobjects	parameters of the process
message queue	channel
message server	atomic block
state variables of a rebec	global variables

Table 1. $\mathcal{R}$ebeca to Promela mapping

After the translation of rebec processes, the Scheduler process is defined, which is straightforward: it nondeterministically selects a rebec process. The last part is the init process, where the processes are initialized.

4 Implementation and Case-Studies

Sarir is a tool created to automate the mapping described in the previous sections. Sarir, which is implemented using the C++ language and the ANTLR parser generator, gets a $\mathcal{R}$ebeca description file as its input and generates an mCRL2 output file. The input language is the same as the $\mathcal{R}$ebeca specification which is presented in [12]; it is further enhanced by removing the requirement to specify the maximum length of the internal queue (thanks to the dynamic structure of the list data type in mCRL2). The output file can be linearized [14] and further analyzed using the mCRL2 tool-set.

To study the proposed translation in action, a series of experiments was carried out with Sarir. The same experiments were also performed by a previously available tool [13], which converts $\mathcal{R}$ebeca specifications to Promela. The details of the Promela translation can be found in [13]. Here we summarize the main points. The translation is a natural one to one mapping, presented in Table 1 and goes along the same lines as our translation to mCRL2.

The $\mathcal{R}$ebeca sources of most of the cases studied here can be found in [11]. The new case study is the identify phase of the communication standard IEEE 1394 [10], which is intended for hot pluggable high performance buses. Due to the lack of space, we only refer to the Rebeca home page for a complete source and the mCRL2 translation.

After the translation, several internal actions are hidden. First of all, the queues of rebecs are hidden in $\mathcal{R}$ebeca, so there is no need to let the enqueue and dequeue messages to be visible. The situation is similar for the get actions (i.e., valuation of variables), because we only have to track the changes of the variables. We hide the setting actions of the variables not

	mCRL2				Spin	
	St.	Trans.	St. min.	Trans. min.	St.	Trans.
Tree Identify	6464	10810	47	90	11818	41917
Bridge Controller	467	616	54	99	911	1975
CSMA-CD	28	40	6	7	62	163
ProducerConsumer	44	51	20	27	48	77

Table 2. Model Checking Statistics

relevant for the external behavior. As rebecs execute deterministically, we use prioritization of these hidden actions when generating the state space [7]. After generating the state space, we also check our desired property using the evaluator tool in the CADP tool-set [5]. Some statistics of the generated state space are shown in Table 2. The results after reducing the model with branching bisimulation are also given. It can be observed that even without reduction, the state space generated by the outcome from Sarir is substantially smaller than the state spaces of the earlier tool. After an investigation of the state spaces, it turns out that statements such as sending a message with parameters results in a number of transitions in Spin while the same statements are translated into a single (parameterized) basic action in mCRL2. Note that both Spin and mCRL2 use partial order reduction. Moreover, branching bisimulation reduction substantially decreases the size of the state spaces generated from the mCRL2 code.

More reductions are still possible. As stated before, we are hiding all actions but those assigning a new value to an externally relevant variable. If the desired property contains only a proper subset of variables, actions that change the value of the other variables can also be hidden. Furthermore, we can reduce modulo weak trace equivalence. In the tree identify protocol, this reduces the state space to 12 states and 18 transitions. This last state space can easily be inspected using the visualization tools in the mCRL2 tool-set.

5 Conclusion and Future Work

In this paper, we defined and implemented a translation from an actor-based language ($\mathcal{R}$ebeca) to process algebra (mCRL2) in order to benefit from the verification tools for the target language. Experiments reveal that the mCRL2 reduction tools result in substantially smaller models when compared to the models generated by the translation to Promela. This is mainly due to the possibility of abstraction from the internal queue contents and the actions concerning reading internal variables. The existing reduction techniques for mCRL2 are then very effective on the resulting models. We are currently working to exploit the reduction algorithms for mCRL2 directly on $\mathcal{R}$ebeca models.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha and P. Thati. An Algebraic Theory of Actors and its Application to a Simple Object-Based Language. In *Ole-Johan Dahl's Festschrift*, volume 2635 of LNCS, pages 26–57, Springer, 2004.

[3] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[4] S. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specification. In *Proceedings of CAV'01*, volume 2102 of LNCS, pages 250-254, Springer, 2001.

[5] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu and M. Sighireanu. CADP - A Protocol Validation and Verification Toolbox. In *Proceedings of CAV '96*, volume 1102 of LNCS, pages 437–440, Springer, 1996.

[6] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Proceedings of ACP'94*, Springer, pages 26-62, 1994.

[7] J.F. Groote and M.A. Reniers. Algebraic Process Verification, In *Handbook of Process Algebra.* Chapter 17, pages 1151-1208, Elsevier, 2001.

[8] J.F. Groote, A.H.J. Mathijssen, M.J. van Weerdenburg and Y.S. Usenko. From μCRL to mCRL2: motivation and outline. In *volume 162 of ENCTS*, pages 191–196. Elsevier, 2006. See also `www.mcrl2.org`.

[9] G.J. Holzmann. The Model Checker SPIN. IEEE Trans. Softw. Eng. 23(5):279–295, 1997.

[10] IEEE CS. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, 1996.

[11] M. Sirjani. Formal Specification and Verification of Concurrent and Reactive Systems. Ph.D. Thesis, Sharif Univ. of Tech., 2004.

[12] M. Sirjani, A. Movaghar, A. Shali and F.S. de Boer. Modeling and Verification of Reactive Systems using Rebeca. Fundam. Inform. 63(4):385–410, 2004.

[13] M. Sirjani, A. Movaghar, A. Shali and F.S. de Boer. Model Checking, Automated Abstraction, and Compositional Verification of $\mathcal{R}$ebeca Models. J. UCS. 11(6):1054–1082, 2005.

[14] Y.S. Usenko. Linearization in μCRL. Ph.D. Thesis, TU/Eindhoven, 2002.