Mapping From Timed Rebeca to Erlang

Árni Hermann Reynisson Matteo Cimini

Translating Timed Rebeca to Erlang

Abstract Syntax

We first provide an abstract syntax for Timed Rebeca, in order to give a more clear presentation of the encoding. Moreover, only relevant syntactic categories are taken into account, for instance encoding of if statements and almost all expressions are not shown since their implemention is trivial or subsumed by the encoding.

$\langle Model \rangle$	::=	$r_1 \dots r_i\ m$
$\langle ReactiveClass \rangle$	::=	`reactiveclass` c `(`n`){`$k\ s\ i\ m$`}`
$\langle KnownRebecs \rangle$	::=	`knownrebecs` $\{t_1 v_1 \dots t_i v_i\}$
$\langle StateVars \rangle$	::=	`statevars` $\{t_1 v_1 \dots t_i v_i\}$
$\langle MsgSrvInit \rangle$	::=	`msgsrv initial(`$t_1 v_1 \dots t_i v_i$`)` $\{s_1 \dots s_i\}$
$\langle MsgSrv \rangle$	::=	`msgsrv` n`(`$t_1 v_1 \dots t_i v_i$`)` $\{s_1 \dots s_i\}$
$\langle Stm \rangle$	::=	$v = e$;
	|	$r.m(e_1 \dots e_i)$;
	|	$r.m(e_1 \dots e_i)$ `after(`e_a`)` `deadline(`e_d`)`;
	|	`delay(`e`)`;
	|	$\{s_1 \dots s_i\}$
$\langle Exp \rangle$	::=	`now()`
$\langle Main \rangle$	::=	`main`$\{d_1 \dots d_i\}$
$\langle InstanceDecl \rangle$	::=	$t_r v_r$ `(`$t_1 v_1 \dots t_i v_i$`):(`$k_1 \dots k_i$`)`

We provide the encoding by means of several mappings, one for each syntactic category of Timed Rebeca. Some of the encodings are parametrized by the information contained in a structure we call $conf$. The structure $conf$ contains three fields: $knownrebecs$, $statevar$ and $localvars$. Intuitively, the field $knownrebecs$ contains the set of known rebecs, and $statevar$ and $localvars$ contains the pairs variable-value for both state and local variables. The structure $conf$ is created in a preprocessing step and then passed to the encoding mappings. In our encodings we use the standard dot notation in order to access the structure $conf$, e.g. $conf.statevar$ to access the field $statevar$ of the structure $conf$. Below we give a brief description of the relevant parts of the mappings.

$\mathcal{MO}(r_1 \dots r_i\ m)$: **Encoding for** $\langle Model \rangle$**.** Where $r_1 \dots r_i$ are rebecs and m is the code in *main*. This function, computes the structure *conf* for each rebec, encodes each rebec passing this structure as parameter. Next, it encodes the code in *main*.

$\mathcal{R}(\texttt{reactiveclass}\ c\ \texttt{(}n\texttt{)\{}k\ s\ i\ m_1\ m_2 \dots m_i\texttt{\}})$**: Encoding for** $\langle ReactiveClass \rangle$**.** Where c is the name of the reactive class, n is a natural number denoting the size of the bag, k is a set of knownrebecs, s is a set of state variables, i is the *initial* method and $m_1 \dots m_i$ are methods. This function, encodes the reactive class in 3 erlang functions with same name, but accepting different formal parameters, so with different signatures.

1. the first erlang function accepts the known rebecs and call the second function,
2. the second function accepts the initial message, and once arrived, it runs the corresponding code, obtained with the mapping $\mathcal{B}$. This mapping returns a new set of state variables, indeed variables might have been changed during the execution of a *inital*. Since in erlang structures are immutable, they cannot be modified directly and in our encoding, following standard solutions, we create a new structure and return it as value. After the execution of the code for *initial*, this function calls the third function, passing this new set of state variables as well as the set of known rebecs.
3. the third function waits for incoming messages, which correspond to method calls. Once arrived, it runs the corresponding code, obtained again with $\mathcal{B}$. After that, it put itself ready to accept again messages by function calling itself, but with the modified set of state variables. Indeed variables might have been changed during the execution of a method.

$\mathcal{B}(\texttt{msgsrv}\ m\texttt{(}t_1 v_1 \dots t_i v_i\texttt{)\{}s_1 \dots s_i\texttt{\}})$: **Encoding for** $\langle MsgSrv \rangle$**.** Where m is the name of the method, $t_1 \dots t_i$ are type names, $v_1 \dots v_i$ are identifiers, and $s_1 \dots s_i$ are the statements of the body of the method. This function, using pattern maching, put the reactive class waiting for a message

$$\{Sender, TT, DL, m, w_1 \dots w_i\}$$

which is basically the messages exchanged in the actual Timed Rebeca. When such a message is arrived, we check if the time tag of the message is not higher than the current time, and also if the deadline of the message is not expired. In case not, we execute the statements of the method, otherwise a null action is executed. A few peculiarities of this encoding deserve a word.

- Performing a null action corresponds to what in Timed Rebeca is the discarding of the message. Indeed, in the erlang system the message has been delivered and it will be not processed again.

- The execution of statements is quite involved. Indeed, structures are immutable in erlang, but in Timed Rebeca the execution of some statements might change variable values. Successive statements then should be executed knowing the new values for variables. Our solution is to execute every statement as a function that receive the set of variables as argument, and return a new set of variable. The auxiliary function AP takes care of correctly composing these functions.
- The reader might wonder why we discard also messages whose time tag is higher than the current time. We indeed have seen that the processing of this type of messages must just be postponed. To understand this, the reader should consider that in order to encode a Timed Rebeca $after$ construct, we rely on the $after$ construct, which would actually send the message only at the right time. The erlang system takes care of this aspect for us.
- The function $tr\_now()$ recovers the current time from the erlang primitive $erlang\_now()$.

$\mathcal{S}$**: Encoding of** $\langle Stm \rangle$**.** This function encodes statements from Timed Rebeca into anonymous functions in erlang. Functions receive as input the set of variables and returns a new set of variables. The two relevant cases to discuss are the method invokation when it involves $after$ and *deadline* constructs, and the *delay* statement.

- $\mathcal{S}(r.m(e_1 \dots e_i)$ `after(`a`) deadline(`d`);`$)$: It creates a new process using the primitive *spawn*. This new process, uses a receive with an empty body and the erlang $after$ to send the message. As said above, differently from Timed Rebeca where messages are sent immediately but carrying the time tag from when they become pickable, here the erlang system care of this aspect for us, actually waiting the expected amount of time before sending the message. Thanks to the primitive *spawn* the sender process does not stop its execution by the effect of $after$, it is instead the new process to wait. The reader should also notice that the message sent by this new process contains the father process as sender, not itself.
- $\mathcal{S}($`delay(`e`);`$)$: It simply performs a receive with an empty body and $after$, in order to let the time pass, performing a null action, afterwards.

other mappings: $\mathcal{I}$ and $\mathcal{T}$ translate name of identifiers and types from Timed Rebeca to erlang. $\mathcal{E}$ encodes expression and it is implemented as expected. $\mathcal{K}$ translate constants from Timed Rebeca to erlang. $\mathcal{M}$ encodes the *main* code and $\mathcal{PC}$ and $\mathcal{L}$ encodes instance declarations.

Even if the whole encoding is not shown, the source code of an implementation of it in erlang can be found at $http://en.ru.is/icerose/applying-formal-methods/projects/TARO$.

Timed Rebeca to Erlang Translation

$\mathcal{MO}(r_1 \ldots r_i\ m) =$ $\mathcal{R}(r_1)$ $conf_1$
$\vdots$
$\mathcal{R}(r_i)$ $conf_i$
$\mathcal{M}(m)$

$\mathcal{R}($`reactiveclass` c `(`n`){`k s m_1 $m_2 \ldots m_i$`})` $conf =$

```
n () ->
  receive
    {B(k) conf} ->
      n (#conf.knownrebecs{B(k) conf})
  end.
n (KnownRebecs) ->
  StateVars=#conf.statevars{},
  LocalVars=#conf.localvars{},
  {NewStateVars, _} = receive
    B(m_1) conf
  end.
  n(KnownRebecs, NewStateVars)
n (KnownRebecs,StateVars) ->
  LocalVars=#conf.localvars{},
  {NewStateVars, _} = receive
    B(m_2) conf
    ⋮
    B(m_i) conf
  end.
  n(KnownRebecs, NewStateVars)
```

$\mathcal{B}($`msgsrv` $m($`(`$t_1 v_1 \ldots t_i v_i$`)`$\{s_1 \ldots s_i\}) \ conf =$

```
{{Sender, TT, DL, m, w_1 ... w_i} ->
  TimeNow = tr_now(),
  if
    ((TimeNow >= TT) andalso (DL == ok orelse TimeNow < DL)) ->
      {NewStateVars, _} = AP(S(s_1) conf ... S(s_i) conf),
      n (KnownRebecs,NewStateVars);
  true ->
    % dropping message
  end
```

$\mathcal{S}($`v=e;`$)$ $conf =$

```
fun(StateVars, LocalVars) ->
  v ∈ conf.statevars ⟶ {StateVars conf.statevars2 = E(e)conf, LocalVars}
  v ∈ conf.localvars ⟶ {StateVars, LocalVars conf.localvars2 = E(e)conf}
  otherwise ⟶ error
end
```

$\mathcal{S}(r\,.\,m(e_1 \ldots e_i)\,;)\ conf =$ `fun(StateVars, LocalVars) ->`
`tr_send(`$\mathcal{I}(r)$`,`$\mathcal{I}(m)$`,{`$\mathcal{E}(e_1) \ldots \mathcal{E}(e_i)$`}),`
`{StateVars, LocalVars}`
`end`

$\mathcal{S}(r\,.\,m(e_1 \ldots e_i)$ `after(`a`) deadline(`d`);`$)\ conf =$
`fun(StateVars, LocalVars) ->`
`tr_sendafter(`$\mathcal{E}(e_a)$`,`$\mathcal{I}(r)$`,`$\mathcal{I}(m)$`,{`$\mathcal{E}(e_1)conf \ldots \mathcal{E}(e_i)conf$`},`$\mathcal{E}(e_d)$`),`
`{StateVars, LocalVars}`
`end`

$\mathcal{S}($`delay(`e`);`$)\ conf =$ `fun(StateVars, LocalVars) ->`
`tr_delay(`$\mathcal{E}(e)conf$`),`
`{StateVars, LocalVars}`
`end`

$\mathcal{M}($`main{`$d_1 \ldots d_i$`}`$) =$ `main() ->`
$\mathcal{PC}(d_1)$,
$\vdots$
$\mathcal{PC}(d_i)$,
$\mathcal{L}(d_1)$,
$\vdots$
$\mathcal{L}(d_i)$.

$\mathcal{PC}(t_r v_r$ `(`$t_1 v_1 \ldots t_i v_i$`):(`$k_1 \ldots k_i$`))` $= \mathcal{I}(v_r)$ `= spawn(fun() ->` $\mathcal{I}(t_r)$`() end)`

$\mathcal{L}(t_r v_r$ `(`$t_1 v_1 \ldots t_i v_i$`):(`$k_1 \ldots k_i$`))` $=$
$\mathcal{I}(v_r)$ `! {`$v_1 \ldots v_i$`} end),`
`tr_send(`$\mathcal{I}(v_r)$`, initial, {`$\mathcal{K}(k_1) \ldots \mathcal{K}(k_i)$`})`