# Supplementary Material- Transforming *AbC* models into UMC models

No Author Given

No Institute Given

### 0.1 Overview

At first glance, it seems that we can model each AbC component into an UMC class where attributes are local variables and component's behaviour are transition rules. In AbC, the communication among components is based on predicates over others other components attributes, which implicitly requires a global view of all system components. Since UMC objects do not share internal states and there is no concept of global data, one could address this issue by employing another UMC class playing the role of an central broker for keeping track the global state. This approach, although possible, would lead to inefficiency in that extra transitions are needed for synchronizing local states with the global state when components update their attributes, and that would result in greater complexity of the design. Our strategy is to gather all the component behavioral descriptions, together with their attribute environments to a single UMC class. In this way, components can access attributes of each other directly.

Before presenting the translation in details, we describe the structures used for input and output.

**Input structure** We use a template that looks like the one in Fig. 1 to specify an AbC system.

Here the component type is defined in a block starting at `component` and ending at `end`. Components have their `Name`s, a set of `attributes` and a `behaviour` section. The behaviour of a component is described by a set of process expressions, thus one first defines sub-processes inside the block `let` $\{\ldots\}$ and initiate the behaviour after the keyword `in` where the behaviour could be either a previous defined process name or a process expression composed from sub-processes. Each AbC component can have multiple instances, and they are written after the component definition section. When instantiating, the attribute environment is also defined by assigning corresponding values to attribute names. Finally, a system is a parallel composition from those component instances, written by putting component names after the key word `SYS`. It has been proved that AbC can encode $b\pi$-calculus, which implies the Turing completeness of AbC. For writing specifications conveniently, we introduce in AbC a few additional syntactic notations.

– Expressions can be vectors and their relevant operators in UMC style

```
component Name
   attributes: att1, att2, ...
   behaviour:
        let {
        -- process definitions in AbC syntax
        }
        in
        --a process call or process expression
end
-- other components declaration ...

-- an instantiate of component Name1
C0 : Name1 (att1 -> value1, att2 -> value2 ...)
-- other instances ...

-- declare the system from component instances
SYS ::= C0 || ...
```

**Fig. 1.** A template for AbC specification

- Predicates can contain tests of membership relation between an element and a vector (denoted by $\in, \notin$).
- Inaction process is written as *nil*, variables are prefixed by $

Specifically, our AbC-like specification allows expressions containing vectors and their relevant operators in UMC style, e.g., $v.head, v.tail$ or selector (denotes the first element, the rest and the $i$th element of a vector $v$, respectively. Predicates can contain tests of membership relation between an element and a vector (denoted by $\in, \notin$). In addition, to guarantee a finite representation of AbC specifcations and generated UMC models, we further make some assumptions.

- the specified system consists of a fixed number of components,
- the parallel operator does not occur inside a recursive definition. The only allowed exception is the definition of a process with the structure: $P \coloneqq Q|P$, where $|$ is replaced by its bounded version $|^n$, with $n$ the number of parallel instances to be created. For example: $P := Q \ |^2 \ P$ is interpreted as three processes $P := Q_1 \mid Q_2 \mid Q_3$.

To distinct different components inside the section `Transitions`, each component is assigned an index. Moreover, each component can have different processes (e.g., created by parallel composition), and each process is also assigned index . More formally, we consider an AbC specification as a pair $< D, S >$, where $D$ contains all process definitions and $S$ is an aggregation of the AbC components.

$$S ::= \Gamma_1 : P_1 \parallel \Gamma_2 : P_2 \parallel \ldots \parallel \Gamma_n : P_n$$

That is, $P_i$ are process names whose definitions are stored in $D$.

Components are chosen non-deterministically to perform output actions by relying on an unique signal as trigger, while input actions of one component are triggered by the corresponding sender.

### 0.2 Translation rules

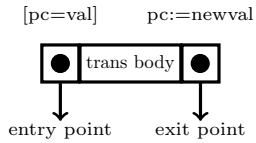
Our top level translation $\mathcal{F}$ takes an AbC specification and translates it into a single UMC class whose general structure is depicted in Fig. 2. It comprises fixed parts such as necessary signals and data structures to model AbC input and output actions, besides generated attributes vectors to store component environments at hand and generated transition rules to model component behaviour.

```
F[[⟨D, S⟩]] =
Class AbCSystem is
Signals: allowsend(i:int),
         bcast(tgt,msg,j:int);
Vars:
   /* necessary variables */
   receiving:bool := false;
   pc:int[];
   bound:obj[];
   /* Attributes declaration */
   attr_1;
   attr_2;
   ...
Transitions:
   /* Initial movement of the system */
   init -> SYS {- /
     for i in 0..pc.length-1 {
        self.allowsend(i);
   }}
   /* Transitions of all components */
   S[[⟨D, S⟩]]
end AbCSystem
```

**Fig. 2.** The template of the output UMC class

The basic idea is to translate each AbC action into each UMC transition with respect to meaning of actions and process structure. Process terms are first parsed into tree structures, which will then be visited recursively while translating. For each component, UMC transitions are generated in a way that they share the same source and target names while transitions them selves are distinguished by several parameters: the index `k` of the component, the index `p` of the executing process and a value `val` to define a guard on transition. Such transitions have a generic form:

```
Ck.s0 -> Ck.s0 {Trigger[pc[k][p]=val & ...]/
   /* transition body */
   pc[k][p]:=newval;
}
```




where `Trigger` can be one of the two signal names (defined in Fig. 2) and the specific UMC code inside this transition body depends on the type of actions. This transition has an intuitive representation shown next to the code snippets. It exposes an *entry point* which is a guard on the value of the `pc[k][p]` with respect to `val`. In the very end of a transition, `pc[k][p]` is assigned a new value referred as *exit point* which is used to guard following up transitions. The `newval` is obtained by looking up a global program counter *cnt* whose value is calculated by the translation.

**Structural Mapping** Let us explain first the structural mapping for a component with index $k$, ignoring for a moment the meaning of actions (send and receive). We let $\mathcal{S}[\![P]\!]_{\rho}^{k,p,v}$ the function that maps a process term $P$ into a set of UMC transitions. $\mathcal{S}$ is parameterized with component index $k$, process index $p$, entry value $v$. It also caries an environment $\rho$ which is a set of mappings in the form $\$x \mapsto \_$. $\rho$ stores necessary information while traversing the process structure which will be eventually used by another mapping function $\mathcal{B}$ to generate the actual UMC code. The function $\mathcal{S}$ works in depth-first manner and we use ; to denote the completion of a left translation before start a new one.

*Inaction.* An inaction process is translated into nothing.

$\mathcal{S}[\![nil]\!]_{\rho}^{k,p,v} = \emptyset$.

*Update.* The translation of $[a := E]P$ stores the update $[a := E]$ into $\rho$ and returns the translation of $P$ under new environment.

$\mathcal{S}[\![[a := E]P]\!]_{\rho}^{k,p,v} = \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v}$ `where` $\rho' = \rho\{\$update \mapsto [a := E]\}$

*Awareness.* The translation of $\langle \Pi \rangle P$ stores the predicate $\Pi$ into $\rho$ and returns the translation of $P$ under new environment.

$\mathcal{S}[\![\langle \Pi \rangle P]\!]_{\rho}^{k,p,v} = \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v}$ `where` $\rho' = \rho\{\$aware \mapsto \Pi\}$

*Nondeterministic choice.* Translation of $P_1 + P_2$ is a sequence of two translations of sub-processes with the same set of parameters.

$\mathcal{S}[\![P_1 + P_2]\!]_{\rho}^{k,p,v} = \mathcal{S}[\![P_1]\!]_{\rho}^{k,p,v}; \mathcal{S}[\![P_2]\!]_{\rho'}^{k,p,v}$

`where` $\rho' = \rho\{\$cnt \mapsto \rho(\$cnt), \$procs \mapsto \rho(\$procs)\}$.
That is, the updated environment $\rho'$ inherits new values of global program counter and new number of processes after finishing the transation for $P_1$.

*Parallel composition.* Translation of $P_1|P_2$ is a sequence of two translations of sub-processes.

$$\mathcal{S}[\![P_1|P_2]\!]^{k,p,v}_{\rho} = \mathcal{S}[\![P_1]\!]^{k,p_1,1}_{\rho_1}; \mathcal{S}[\![P_2]\!]^{k,p_2,1}_{\rho_2}$$
$$\texttt{where } p_1 = \rho(\$procs), p_2 = p_1 + 1,$$
$$\rho_1 = \rho\{\$procs \mapsto p_2 + 1, \$parent \mapsto (p, v)\},$$
$$\rho_2 = \rho_1\{\$procs \mapsto \rho_1(\$procs), \$cnt \mapsto \rho_1(\$cnt)\}.$$

The parallel process creates two new processes with process indexes $p_1$ and $p_2$ which are calculated from the current number of processes. The translation of each process keeps the pair $(p, v)$ under the name *parent* which will be used as an additional guard for the first action.

*Process call.* The translation of a process call $K$ looks up the environment $\rho$ for its process definition $P$ and return a translation of $P$.

$$\mathcal{S}[\![K]\!]^{k,p,v}_{\rho} = \begin{cases} \mathcal{B}[\![\emptyset]\!]^{k,p,exit}_{\rho} & \text{if } \rho(\$K_{\text{visit}}) = \text{true} \\ \mathcal{S}[\![P]\!]^{k,p,v}_{\rho'} & \text{otherwise} \end{cases}$$

$\texttt{where}$ exit $= \rho(\$K_{\text{entry}}), P = \rho(K_{\text{def}}), \rho' = \rho\{\$K_{\text{visit}} \mapsto true, \$K_{\text{entry}} \mapsto v\}$.
If process $K$ is already translated, the function $\mathcal{B}$ generates a dummy transition whose exit point equals to entry point of $K$. Otherwise, it remembers $K$ is visited and stores this fact together with entry point value of $K$ for later recursions.

*Action-prefixing.* The translation of $\alpha.P$ is a behavioural translation of $\alpha$ and a translation of $P$ under new environment, new entry value:

$$\mathcal{S}[\![\alpha.P]\!]^{k,p,v}_{\rho} = \mathcal{B}[\![\alpha]\!]^{k,p,v}_{\rho}; \mathcal{S}[\![P]\!]^{k,p,v'}_{\rho'} \quad \texttt{where } \rho' = \rho(\$cnt \mapsto \rho(\$cnt)\}, v' = \rho'(\$cnt)$$

**Behavioural mapping** The system state is modelled with a UML Parallel state, where each component is in turn modelled by its own region. In UML the behaviors inside parallel region proceed truely in parallel (a trigger event from the object queue is dispatched to all the regions, which evolve in parallel, making a unique system transition. Attribute input and output semantics are modelled with the help of unique events: i) the `bcast(tgt,msg,j)` event which triggers all the receive actions in all components. The broadcasted messages contain three fields: a `tgt` field describing the actual sets of component which are allowed to receive the message, a `msg` field which contains the actual message data, and a `j` field is the index of the sending component, ii) the `allowsend(i)` event, where `i` denotes a component index used to schedule the components through interleaving when sending messages. The mapping of a specific action $\alpha$ is carried out by function $\mathcal{B}$ under an environment $\rho$. After generating UMC code, $\mathcal{B}$ also updates new values for program counter *cnt* and the current number of processes *procs*.

*Output Action.* We model the output action in two steps which are forced to occur in a strict sequence: the sending to self of the `bcast` event that dispatches to all the parallel components and the discarding of this very message. The sequentialization is achieved by using a global variable `receiving` acting as a lock, which is set to true when sending, and reset by the subsequent transition done by the sending component.

$\mathcal{B}[\![(E)@\Pi]\!]_\rho^{k,p,v} =$

```
SYS.Ck.s0 -> Ck.s0 {
    allowsend(i)[i=k & receiving=false & pc[k][p]=v & [[ρ($aware)]]]/
    [[ρ($update)]];
    for j in 0..pc.length-1 {
        if ([[Π]]) then {tgt[j]:=1;} else {tgt[j]:=0;}
    };
    receiving:=true;
    self.bcast(tgt,[[E]],k);
    pc[k][p] = [[ρ($cnt) + 1]];
}
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[pc[k][p] = [[ρ($cnt) + 1]]/
    receiving:=false;
    self.allowsend(k);
    pc[k][p] = [[ρ($cnt) + 2]];
}
```

*Input Action.* An input action is translated in the following transition. It is triggered by signal `bcast(tgt,msg,j)` from some senders. It is enabled if the message is for him and the receiving predicate Π and possibly an awareness predicate satisfy. The binding of the input variables names to the received values is modelled by an assignment of actual message `msg` to global data `bound`.

$\mathcal{B}[\![\Pi(x)]\!]_\rho^{k,p,v} =$

```
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[tgt[k]=1 & pc[k][p]=v & [[ρ($aware)]] & [[Π]]]/
    [[ρ($update)]];
    bound[k][p] = msg;
    pc[i][p] = [[ρ($cnt) + 1]];
}
```

*Expressions and Predicates.* In the above generated transition, guards may include the accumulated awareness predicates and the transition body may include accumulated update commands. Those AbC terms, together with the sending, receiving predicates $\Pi$ are translated into UMC code by a family of functions $[\![\cdot]\!]$ whose definitions are given below.

$[\![a]\!]$ = a[j]
$[\![\text{this}.a]\!]$ = a[k]
$[\![v]\!]$ = v
$[\![E_1 \coloneqq E_2]\!]$ = $[\![E_1]\!]$ := $[\![E_2]\!]$

$[\![tt]\!]$ = true
$[\![E_1 \bowtie E_2]\!]$ = $[\![E_1]\!] \bowtie [\![E_2]\!]$
$[\![\Pi_1 \wedge \Pi_2]\!]$ = $[\![\Pi_1]\!]$ & $[\![\Pi_2]\!]$
$[\![\neg \Pi]\!]$ = not $[\![\Pi]\!]$

$$[\![x]\!] = \begin{cases} \texttt{msg} & \text{if } x \text{ appears in receiving predicate} \\ \texttt{bound[k][p]} & \text{otherwise} \end{cases}$$

Notice the case of variable $x$, the translation depends on the context. if $x$ appears in the receiving predicate, its value is contained in the sent message `msg`. Otherwise, we look up its binding from global vector `bound`. Moreover, when $x$ is a sequence of variables, we store the indexes of each variable in a separate mapping for later lookup.

*Global awareness operator* We now describe the translation of the global awareness construct $\Pi?P_1 \_ P_2$. This can be seen as a choice process of $P_1$ and $P_2$ in which each subprocess is equipped with a fixed, additional guard on a global variable $turn[k][c] = 1$, where $c$ is the index of each branch calculated by the translation.

Each branch will be enabled based on the evaluation of $\Pi$. Since $\Pi$ is declared on other components attributes, it is evaluated similarly as the sending predicate.

Let $c1$, $c2$ are branch indexes of $P_1$, $P_2$ respectively, we first translate the global awareness predicate in to a transition (without trigger) whose structures mimic the first transition of the output action.

```
SYS.Ck.s0 -> Ck.s0 {
    -[pc[k][p]=v & [[ρ($aware)]]]/
    for j in 0..pc.length-1 {
        if ([[Π]]) then {tgt[j]:=1;} else {tgt[j]:=0;}
    };
    if tgt.length > 0 then { -- if there any satisfied components
       turn[k][c1] := 1;     -- run P1
       turn[k][c2] := 0;     -- disable P2
   }
   else {
       turn[k][c1] = 0;       -- disable P1
       turn[k][c2] = 1;       -- run P2
   };
   pc[k][p] := newval;  -- continue as normal
}
```

The guard of the first transition of $P_1$ has the form:

```
SYS.Ck.s0 -> Ck.s0 {
    Trigger[... & pc[k][p]=newval & turn[k][c1] = 1 & ...]/
    ...
}
```

and that of $P_2$ has:

```
SYS.Ck.s0 -> Ck.s0 {
    Trigger[... & pc[k][p]=newval & turn[k][c2] = 1 & ...]/
    ...
}
```

### 0.3 Implementation & Optimizations

We have developed an automated tool implementing the above translation in Erlang. The tool contains about 1400 lines of Erlang code, in addition to 50 lines of leex code and 180 lines of yecc code for parsing the AbC specifications. Our tool and translated case studies are available online at [1]. It is worth mentioning that we have done various optimizations from the basic scheme. These include:

- Elimination of recursive transitions: When entering a recursive call, the mapping function creates another transition which resets the program counter to the process entry value. We detect this call early so that this reset can be put in the transition of the previous action.
- Handling empty send: Associated examples come with the AbC calculus [?,?] often use the construct `[a:=E]()@(ff)` in order to model updating attribute `a`. In this case, the translation creates only one transition doing only update operation. Similarly, in cases there are no receivers satisfying a sending predicate, we jumps to the next transition instead of doing a complementary receive.
- Schedule parallel instances of the same process: parallel instances can cause a lot of interleaving, we set additional guards on the first action of parallel instances such that an instance can move only if previous instances have already moved.
- The translation for global awareness operator is also optimized by considering the type of first action $\alpha$ of $P_1$. If $\alpha$ is an attribute output, we merge the check of global awareness operator into the transition of $\alpha$ instead of having a separate transition for it.
- Nice parallelism of UMC: The model checker is tuned with an option which allows the user to specify a "niceparallelism" property of a class, which tells to the model checker that all the $n!$ sequentializations of $n$ parallel transitions occurring when a message is broadcasted to all intended receipts are equivalent, therefore letting UMC to pick just one of the possibilities as the model semantics.

[1] https://github.com/ArBITRAL/AbC2UMC