

# Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap?

M.F. van Amstel, M.G.J. van den Brand, Z. Protić, and T. Verhoeff

Department of Mathematics and Computer Science  
Technische Universiteit Eindhoven  
Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven, The Netherlands  
{M.F.v.Amstel,M.G.J.v.d.Brand,Z.Protic,T.Verhoeff}@tue.nl

**Abstract.** There exist many formalisms for modeling the behavior of (software) systems. These formalisms serve different purposes. Process algebras are used for algebraic and axiomatic reasoning about the behavior of distributed systems. UML state machines are suitable for automatic software generation. We have developed a transformation from the process algebra ACP into UML state machines to enable automatic software generation from process algebra models. This transformation needs to preserve both behavioral and structural properties. The combination of these preservation requirements gives rise to a *semantic gap*. It implies that we cannot transform ACP models into UML state machines on a syntactic level only.

We address this semantic gap and propose a way of bridging it. To validate our proposal, we have implemented a tool for automatic transformation of ACP process algebra models into UML state machines.

## 1 Introduction

In this paper we address the semantic gap that arises when transforming models specified in one formalism into models in another formalism. A transformation between models in different formalisms needs to bridge a syntactic gap. This is a well-known problem. However, in many applications one also needs to preserve semantic properties. This is not trivial since the semantic domains of the source and target formalism may differ, or a formal semantics may be lacking. Moreover, additional requirements on semantical properties can affect a transformation.

The goal within the FALCON project [1] is to model embedded systems in a warehousing environment and use these models initially for simulation, but later for, amongst others, automatic software generation. These systems are being modeled using a process algebra [2]. Process algebra is a formalism used for algebraic and axiomatic reasoning about the behavior of systems, in particular those involving concurrency [3]. However, little is known about automatic code generation from process algebra models. We use UML state machines as an intermediate step because multiple techniques are available for automatic code

generation from them. Therefore we propose a transformation from process algebra models to UML state machines [4]. We start with plain process algebra in order to understand the basics of software generation from, and model transformations based on process algebras. In this paper we use the well-known process algebra ACP (Algebra of Communicating Processes) [5,6] without encapsulation. In the transformation of this small process algebra we already encounter a semantic gap. The obtained results will be used when translating process algebra based formalisms like timed  $\chi$  [7] or mCRL2 [8].

Processes in distributed systems are allocated to different machines that run in parallel. Therefore we have to ensure that the automatically generated code can also be deployed on different machines. This requires that the ACP models and the obtained state machines are structurally equivalent with respect to parallel behavior. The ACP models and the state machines obtained from this transformation obviously need to exhibit the same behavior. It is this combination of requirements, i.e., preserving structural and behavioral properties, that confronted us with the problems of bridging a semantic gap. In ACP, constructs are available for modeling synchronous communication between parallel processes. UML state machines are inherently asynchronous, hence no primitives exist for modeling synchronous communication. This means that the transformation from ACP to UML state machines encompasses more than translating syntax. Special care is needed to ensure that the semantic gap is bridged in order to preserve both behavioral and structural properties.

The remainder of this paper is structured as follows. Section 2 describes related work. In Section 3 our approach to transform ACP models into UML state machines is explained. In this section also the semantic gap is explained in depth and we propose and evaluate some solutions for bridging this gap. Section 4 describes the implementation of our transformation. An illustration using our implementation can be found in Section 5. Section 6 contains the conclusions of our work and gives some directions for further research.

## 2 Related Work

Many papers have been published on the subject of transforming process algebras into various formalisms. In one of the first papers in this area a transformation from the Algebra of Timed Processes (ATP) into a variant of timed graphs is presented [9]. The authors aim at unifying behavioral description formalisms for timed systems. In [10], a transformation of a timed process algebra based on LOTOS operators to Dynamic State Graphs is presented. The main purpose of that mapping is to visualize and simulate process algebra models. In [11] three different process algebras are analyzed and compared. The results of that analysis are used to propose a framework for visualizing process algebras. In a recent technical report [12], a transformation from timed  $\chi$  into UPPAAL timed automata is presented. The main purpose of that mapping is to enable model checking and verification of process algebra models. Our approach, however, focuses on automatic software generation.

Research has also been performed in the field of software generation from UML state machines. In [13] an overview is given of different approaches for generating code from UML state machines. The authors identify the weaknesses in these approaches and also propose their own technique. The approach described in [14] generates object-oriented code from UML state machines. In our work we use the tool Telelogic Rhapsody to generate simulation code from state machines. The semantics of Rhapsody state machines differs slightly from UML state machines [15], but this does not affect our approach.

### 3 Transforming ACP Models into UML State Machines

In this section first a short introduction is given to the relevant parts of ACP. Next, our transformation from ACP to UML state machines is described. In Section 3.3 the semantic gap is discussed in more detail. Section 3.4 presents our solution to bridge this gap.

#### 3.1 Algebra of Communicating Processes

In this paper, we consider basic ACP without the encapsulation operator ( $\partial$ ) to illustrate one of the main issues in bridging a semantic gap. It suffices to leave out the encapsulation operator because a semantic gap already emerges without it. An *ACP model* consists of a *process term* ( $P$ ) and a *communication function* ( $\gamma$ ). A process term is built from *atoms* and *operators*.

In ACP there are three types of atoms. First, there is the deadlock constant ( $\delta$ ) that denotes inaction. When this constant is encountered in a process it deadlocks. Second, there is the empty process constant ( $\epsilon$ ) that denotes doing nothing. Third, there are the actions that can be performed by a process term.

In ACP there are six operators. First, there is the sequential composition ( $\cdot$ ). The sequential composition of  $n$  process terms,  $P_1 \cdot P_2 \cdot \dots \cdot P_n$ , denotes that the execution of  $P_1$  precedes the execution of  $P_2$  and so on. Second there is the action prefix operator ( $\cdot$ ). This operator is similar to the sequential composition, therefore we consider it as such. Third, there is the alternative composition ( $+$ ). The alternative composition of  $n$  process terms,  $P_1 + P_2 + \dots + P_n$ , denotes that only one of these process terms is executed. This choice is made non-deterministically. Fourth, there is the parallel composition ( $\parallel$ ). The parallel composition of  $n$  process terms,  $P_1 \parallel P_2 \parallel \dots \parallel P_n$ , denotes that these process terms are executed quasi-parallel. This means that the process terms are arbitrarily interleaved whilst maintaining their internal ordering. Consider for example the parallel composition  $(a \cdot b) \parallel (c \cdot d)$ . The arbitrary interleaving may not result in a situation where the execution of  $b$  precedes the execution of  $a$ , or where the execution of  $d$  precedes the execution of  $c$ . There is, however, more to the parallel composition which will be explained in Section 3.3. Fifth, there is the left merge operator ( $\parallel$ ) which is closely related to the parallel composition. It denotes that the first action to the left of the operator is executed first whereafter the remaining process term continues as a parallel composition. Consider

for example the process term  $(a \cdot x) \parallel y$ . This means that first action  $a$  is executed whereafter the process term behaves as  $x \parallel y$ . This operator occurs for technical reasons [16] in the reduction of ACP process terms and is seldomly used in modeling directly. Last, there is the communication merge operator ( $|$ ). This operator is used together with the communication function ( $\gamma$ ) to express communication (or interaction) between two actions. The communication function expresses which actions can communicate and what the result of this communication is. For example  $\gamma(a, b) = c$  expresses that in the process term  $a|b$  actions  $a$  and  $b$  communicate, resulting in action  $c$ . If this communication function does not exist, actions  $a$  and  $b$  cannot communicate. This means that the process term  $a|b$  results in a deadlock ( $\delta$ ). The communication merge operator also occurs for technical reasons [16] in the reduction of ACP process terms and is seldomly used in modeling directly.

### 3.2 Transformation

Our transformation  $f$  from ACP models to UML state machines takes an ACP model, which consists of a process term and a communication function, as an argument and returns a UML state machine.

$$f : \text{ACP model} \rightarrow \text{UML state machine}$$

Every non-atomic process term is built from smaller process terms, resulting in an implicit tree structure. Our transformation traverses this tree and transforms every subtree into a partial state machine that is structurally equivalent to the process term in the node. ACP has axiomatic rewrite rules. This means that an ACP process term can be rewritten (transformed) into a different-but-equivalent process term using these rules. An example of this rewriting process is given in Figure 1. In order to ensure the required structural equivalence with respect to parallel behavior, the original ACP model should not be rewritten such as to remove parallel composition operators. In general it is not required to maintain structure. However, we will maximize structure preservation for the other ACP constructs as well, since we want to preserve designer's choices as closely as possible. Therefore the ACP axioms for rewriting a process term are used as little as possible by our transformation.

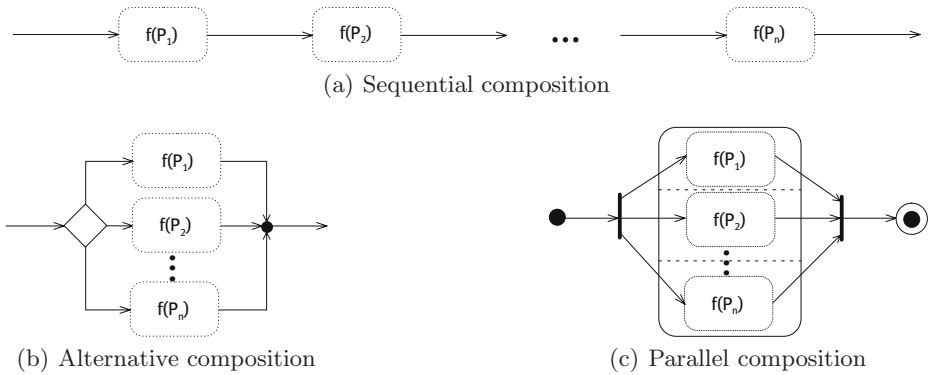
We use the formal semantics of ACP described in [17] and the semantics description of UML presented in [4] to explain informally the behavioral equivalence of ACP constructs and the resulting UML state machines. With behavioral equivalence we mean that the state machines need to define exactly the same traces as the original ACP models.

The state machine constructs for the sequential, alternative, and parallel composition are straightforward, i.e., the semantics is clear from the syntax. The sequential composition maps to the state machine depicted in Figure 2(a). This construct enforces that the execution of  $P_1$  precedes the execution of  $P_2$  and so on. The dotted state labeled  $f(P_i)$  represent the partial state machine acquired after applying  $f$  to process term  $P_i$ . The state machine to which the alternative composition maps is depicted in Figure 2(b). The choice state ensures that

$$\begin{aligned}
 & a.x + (b.y + a.x) \\
 = & \{\text{Axiom A1 : } x + y = y + x\} \\
 & a.x + (a.x + b.y) \\
 = & \{\text{Axiom A2 : } (x + y) + z = x + (y + z)\} \\
 & (a.x + a.x) + b.y \\
 = & \{\text{Axiom A3 : } (x + x) = x\} \\
 & a.x + b.y
 \end{aligned}$$

**Fig. 1.** Example of rewriting using the ACP axioms

only one of the paths will (non-deterministically) be selected for execution. Figure 2(c) depicts the state machine for the parallel composition. The fork and join states are used to ensure that all parallel branches start and end simultaneously. The transformation of the atoms is explained in Section 3.4.



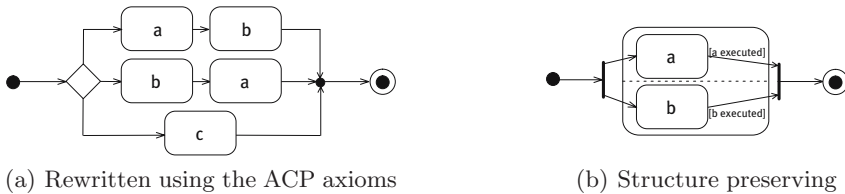
**Fig. 2.** Transformation

The left merge operator cannot be expressed in a natural way in a state machine. It is impossible to express that a specific action in one branch of a parallel composition should be performed first. Therefore, the left merge operator is eliminated by rewriting according to the axioms of ACP. Also the communication merge operator cannot be expressed in a natural way in a state machine. Therefore, when communication of two actions is encountered the communication function ( $\gamma$ ) is consulted whether this communication should be rewritten into an action or the deadlock constant. These are the only two cases in which structure is not preserved, but since these two constructs are seldomly used in modeling this is acceptable.

### 3.3 Semantic Gap

In ACP the parallel composition of two or more process terms represents not just the interleaving of these terms. It also involves communication of the actions inside them. Consider for example the ACP process term  $a||b$ . This will rewrite using the ACP axioms to  $a \cdot b + b \cdot a + a|b$ . Suppose now the communication function  $\gamma(a, b) = c$  exists for some action  $c$ . In this case actions  $a$  and  $b$  can be executed simultaneously ( $a|b$ ) and communicate. The result of this communication is action  $c$ . So the traces allowed by this parallel composition are  $a \cdot b$ ,  $b \cdot a$ , and  $c$ . In UML state machines the parallel composition, created by transitions that fork into orthogonal regions, represents interleaving or concurrent execution of the traces in the orthogonal regions. There is no communication between the actions in these traces like in ACP. This gap between the semantics of ACP and UML state machines needs to be bridged.

One possibility to bridge this gap is to use the ACP axioms to rewrite an ACP model such that all parallel composition operators are removed. In this way all communication is made explicit. The state machine acquired after rewriting is sketched in Figure 3(a). This is not a valid solution since one of the requirements is that the UML state machines need to preserve the structure of the ACP models, at least with respect to the parallel composition. In fact, the combination of the requirements of preserving both behavioral and structural properties gives rise to the semantic gap.



**Fig. 3.** State machine representations of  $a||b$

Another possible solution is to exploit the semantic openness of UML state machines. Therefore we propose an action dispatcher that takes care of executing all actions. Actions are not executed in the state machine itself. Instead, an action is announced to the action dispatcher and the branch of the state machine that contains the action is blocked. After the action dispatcher executes the action, it enables the appropriate branch again such that the state machine can continue. If multiple actions that can communicate have been announced, the action dispatcher ensures that communication can occur in accordance with the communication function. Suppose for example that actions  $a$  and  $b$  are announced and that  $\gamma(a, b) = c$ , i.e., actions  $a$  and  $b$  can communicate resulting in action  $c$ . The action dispatcher now also allows action  $c$  to be executed. Using the action dispatcher, we succeed in preserving most of the structure of the ACP model. Figure 3(b) sketches the resulting state machine.

It can be argued whether having a global action dispatcher that exploits the semantic openness of UML state machines is a proper solution. The disadvantage of having this global action dispatcher is that communication behavior is invisible in the state machine. In the case of ACP this is not a problem. Communication in ACP models is also invisible because it is expressed by a global communication function ( $\gamma$ ) and not in a process term itself. The effect on the semantics is also limited since the action dispatcher can be modeled using the UML as well. We generate a state machine from an ACP model and add an implementation of the action dispatcher in the form of a UML class and state machine to it.

### 3.4 Action Dispatcher

Figure 4 depicts the class diagram representing the (single) action dispatcher. This action dispatcher object has an *action pool* of zero or more action objects. The action pool consists of all action objects ready for execution. The  $\gamma$  attribute of the class is the communication function  $\gamma$ . It is, like in ACP, used to determine whether a pair of actions can communicate. The methods of the class handle adding actions to, executing actions in, and removing actions from the action pool. The functionality of the methods is explained below. The action dispatcher is generic. This means that the same action dispatcher is generated for all ACP models. Only the  $\gamma$  attribute is generated from the model.

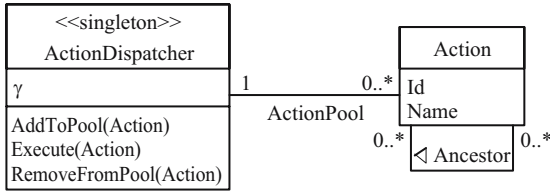


Fig. 4. Action dispatcher class diagram

An action object has two attributes: an identifier to uniquely identify the syntactic occurrence of the action and the name of the action. This name is the same name as the one occurring in the ACP process term. An action can be related to other actions, its *ancestors*. If an action  $x$  is the result of communication, e.g.  $\gamma(a, b) = x$ , actions  $a$  and  $b$  are considered to be its *parents*. The set of ancestors of  $x$  can be found by taking the transitive closure of the ‘*is parent of*’  $x$  relation. Note that an action that is the result of communication can communicate with other actions, e.g.  $\gamma(a, b) = c$  and  $\gamma(c, d) = e$ . Because an action cannot communicate with its ancestors, the ancestors of an action need to be known to correctly handle communication. If an action is not the result of a communication it does not have any ancestors.

The life cycle of an action object is such that it will first be added to the action pool. After some time, it may be executed whereafter it is removed from the action pool. In case of communication, action objects can also be removed from the action pool without having been executed themselves. Action objects can even stay in the action pool forever in case of a deadlock.



**Fig. 5.** Transformation of atoms

*Transformation of Atoms.* The transformation of all constructs except for the atoms has already been explained in Section 3.2. The atom  $a$  maps to the state machine depicted in Figure 5. The entry activity on the simple state creates an action object from atom  $a$  and invokes the *AddToPool* method of the action dispatcher. This puts the newly created action object in the action pool. In order to ensure that the state machine does not continue until the action has been executed, a guard is present on the outgoing transition. This guard is *true* when the action object is not in the action pool. This is the case when the action has been executed or has communicated.

*Addition.* The method *AddToPool*( $x$ ) is invoked by the entry activity on the simple state an atom is mapped to (cf. Figure 5). Its purpose is to extend the action pool with  $x$  and to maintain closure of the action pool under  $\gamma$ . If an action object  $x$  is added to the action pool, and it can communicate with another action object  $a$  already in the pool that is not one of its ancestors, then a new action object for the communication result given by  $\gamma$  is recursively added to the action pool (lines 3–6 in Figure 6). On line 5 a new action object is created with a new identifier and as name the result of the communication function  $\gamma$ . Also, the set of all ancestors of  $x$  is assigned to this action object.

*Execution.* An action object  $x$  in the action pool that does not represent a deadlock constant will non-deterministically be selected at an arbitrary moment for execution. The purpose of method *Execute*( $x$ ) is to find all actions that execute along with  $x$ , i.e., all actions that, directly or indirectly, gave rise to  $x$  through communication, and to clean up the action pool. If the state machine cannot proceed and there are only action objects in the action pool that represent a deadlock constant, it is in a deadlock state.

*Removal.* The purpose of method *RemoveFromPool*( $x$ ) is to remove action object  $x$  from the action pool. To maintain closure of the action pool under  $\gamma$ , also all action objects that are the result, directly or indirectly, of communication involving  $x$  are removed. Note that these resulting action objects do not occur in the conditions of outgoing transitions (cf. Figure 5), because they are the result of communication.

*Correctness Considerations.* Interference between methods of the action dispatcher can be avoided by executing them under mutual exclusion. The action dispatcher controls the state machine through conditions of the form  $a \notin \text{ActionPool}$  only. Note that  $a$  is added to the action pool, falsifying the condition, upon entry into the immediately preceding simple state, and is removed upon its execution, making the condition true. During execution of each method, the action pool changes monotonically to avoid glitches (undesired condition changes).

1. *AddToPool*( $x$ : Action):
2.   ActionPool := ActionPool  $\cup$   $\{x\}$ ;
3.    $\forall_a : \gamma(a.Name, x.Name) = y$
4.      $\rightarrow$  if  $a \in$  ActionPool  $\wedge$   $a.Id \notin x.Ancestor$
5.      $\rightarrow$  NewAction := Action(NewId(),  $y$ ,  $a.Ancestor \cup x.Ancestor \cup \{a, x\}$ );
6.     *AddToPool*(NewAction)
  
7. *Execute*( $x$ : Action):
8.    $\forall_a : a.Id \in x.Ancestor$
9.    $\rightarrow$  *Execute*( $a$ )
10. *RemoveFromPool*( $x$ )
  
11. *RemoveFromPool*( $x$ : Action):
12.   ActionPool := ActionPool  $-$   $\{x\}$ ;
13.    $\forall_a : a \in$  ActionPool
14.    $\rightarrow$  if  $x.Id \in a.Ancestor$
15.    $\rightarrow$  *RemoveFromPool*( $a$ )

**Fig. 6.** Pseudo code for the action dispatcher methods

## 4 Implementation

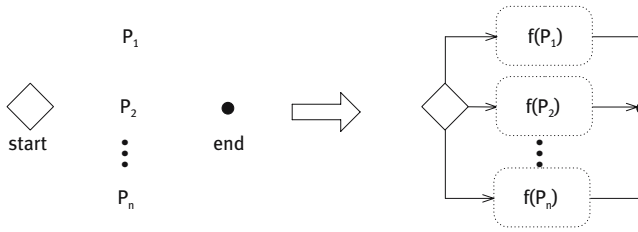
The transformation from ACP models into UML state machines expressed in the XMI format is too complex to implement in a single step. Therefore we split the transformation into four independent steps. This modular approach makes the transformation more transparent, which benefits extensibility, maintainability, and testability. Moreover, every step is (re)usable in isolation.

In the first step of the transformation the ACP model is rewritten using the ACP axioms to remove all instances of the left merge and communication merge operator. Consider for example the ACP process term  $a \parallel (b|c)$  and suppose  $\gamma(b, c) = d$ . This rewrites to  $a.d$ . This step has as in- and output an ACP model expressed in the ACP metamodel<sup>1</sup> we have defined. After this step the ACP model will only consist of constructs that have a state machine equivalent. With a few extensions the transformation used in this step can also be used for rewriting ACP models to their normal form.

In the second step the implicit tree structure of an ACP model is made explicit. For the representation of this tree structure we use an intermediate language for which we defined a metamodel. This language uses a prefix format. Consider for example the alternative composition  $P_1 + P_2 + \dots + P_n$ . The transformation function finds all the alternatives and represents them as  $alt(P_1, P_2, \dots, P_n)$ .

These first two steps are mere preparation for the actual transformation. In the third step the tree representation of an ACP model is transformed into a state machine. This state machine is defined in a state machine language for which we have also defined a metamodel. This language closely resembles UML

<sup>1</sup> Our metamodels are in fact context-free grammars.



**Fig. 7.** Transformation of the alternative composition

state machines. The only difference is that it does not support the history mechanism. We chose to use this intermediate format to avoid having to transform into complex XMI constructs directly. Moreover, it enables the transformation of any UML state machine defined in our state machine language into XMI. This third transformation step is similar to Thompson’s algorithm for transforming regular expressions into non-deterministic finite automata [18]. The transformation function has as arguments an ACP process term represented as a tree and a start and an end state. For the alternative and parallel composition these start state and end state are respectively the choice and junction state, and the fork and join state. In Figure 7 an example is depicted in which the partial state machines for  $n$  alternatives are generated and connected to the choice and junction states. For the sequential composition it is more difficult because the end state of the first partial state machine in the sequence is the start state for the next. These states are not known in advance. To overcome this problem, dummy states are inserted such that the start and end states are known in advance. These dummy states are removed afterwards.

In the last step a state machine is transformed into its XMI [19] representation. This back-end part is isolated, because the XMI standard is actually not so standard. Most UML tools use a different dialect of XMI requiring different back-ends. Currently our implementation is able to generate XMI files for the UML tools ArgoUML [20] and Telelogic Rhapsody [21]. Our state machine language closely resembles UML state machines and there is a one-to-one mapping from UML state machine constructs to XMI. Therefore, this final transformation step is straightforward.

The Telelogic Rhapsody tool allows for execution of state machines. We use this feature to simulate the execution of ACP models. To ensure a correct handling of communication, an implementation of the action dispatcher presented in Section 3.4 is added to the XMI file.

We use the term rewriting system ASF+SDF [22,23] for the development of our metamodels and for the implementation of our transformation. Transformations between languages is one of the main applications of ASF+SDF. These transformations are performed between languages specified in the Syntax Definition Formalism (SDF) using conditional equations specified in the Algebraic Specification Formalism (ASF). Because the concrete syntax of the source and target language of a transformation are formally defined in SDF, syntax-safety

of the input and output of a transformation is guaranteed. This implies that every syntactically correct ACP model is transformed into a syntactically correct XMI document representing a state machine that preserves structural and behavioral properties. Syntax-safety also implies that every ACP model that is syntactically incorrect is not transformed at all.

## 5 Illustration

We have used our implementation on multiple ACP models to verify the correctness of our transformation. This section describes the transformation of an ACP model of a conveyor system into a UML state machine that preserves structural and behavioral properties.

The conveyor system is schematically depicted in Figure 8. Machines  $M_1$  and  $M_2$  put products on a conveyor belt. The products from machine  $M_1$  can go to machines  $M_3$  or  $M_4$  for further processing and the products from machine  $M_2$  can go to machines  $M_4$  or  $M_5$ . When products are sent to machine  $M_4$  by both machines  $M_1$  and  $M_2$  at the same time a collision will occur and an operator should ensure that both products can still enter the machine for processing.

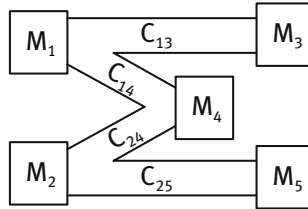


Fig. 8. Conveyor system

The ACP model representing this system is depicted in Figure 9. The process term expresses that a product is produced by machine  $M_1$  which is then sent to machine  $M_3$  or  $M_4$  for further processing and that another product is produced by machine  $M_2$  which is then sent to machine  $M_5$  or  $M_4$  for further processing, possibly at the same time. The communication function ( $\gamma$ ) expresses that an operator rearranges products that collide if two products go from machines  $M_1$  and  $M_2$  to machine  $M_4$  at the same time. Note that only one iteration is modeled.

$$\gamma(C_{14}, C_{24}) = operator$$

$$(M_1 \cdot (C_{13} \cdot M_3 + C_{14} \cdot M_4)) \parallel (M_2 \cdot (C_{25} \cdot M_5 + C_{24} \cdot M_4))$$

Fig. 9. ACP model of the conveyor system

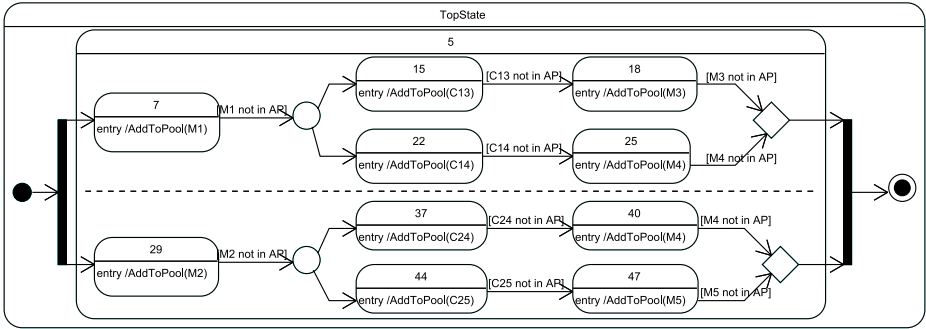


Fig. 10. ArgoUML screen shot depicting the acquired state machine diagram

```
Executable started
Dispatcher Started
Executing : M1
Executing : M2
Executing : C24
Executing : M4
Executing : M3
```

(a) Trace 1

```
Executable started
Dispatcher Started
Executing : M1
Executing : M2
Executing : operator
Executing : M4
Executing : M4
```

(b) Trace 2

```
Executable started
Dispatcher Started
Executing : M2
Executing : M1
Executing : C25
Executing : C14
Executing : M4
Executing : M5
```

(c) Trace 3

Fig. 11. Three Telelogic Rhapsody execution results

The state machine resulting from the transformation should exhibit the same behavior. A screen shot of the state machine acquired from the transformation imported in ArgoUML can be found in Figure 10. Note that ArgoUML uses circles for choice states and diamonds for junction states.

We also transformed this ACP model into an XMI file for Telelogic Rhapsody, enabling simulation of the state machine. Three screen shots showing the results of three different executions of the simulation are depicted in Figure 11. In trace 1 and 3 both products go to different machines. In trace 2 the operator is needed to rearrange collided products.

## 6 Conclusion and Further Research

### 6.1 Conclusions

We have addressed the semantic gap that arises in the transformation from the process algebra ACP without encapsulation into UML state machines. Transforming a model specified in one formalism into a model in another formalism involves more than transforming syntax. Differences in the characteristics of semantics need to be handled meticulously to ensure a correct transformation. In our case a semantic gap emerged as a result of the requirements on the transformation. The transformation should preserve both structural and behavioral properties. Our transformation preserves structure for all operators except for

the seldomly used left merge and communication merge operators. It also preserves behavior by exploiting the semantic openness of UML state machines. We have extended UML state machines with an action dispatcher to ensure that they can generate the same execution traces as the ACP model.

Note that trace equivalence is in general only one aspect of semantic equivalence. Without providing a formal semantics for the UML we cannot guarantee that we have bridged the semantic gap completely. Since there are many formalisms with different (or without) formal semantics, there are probably many model transformations that are not proven to be semantics preserving. Proving that a model transformation preserves semantics requires different expertise.

In general, to bridge a semantic gap when transforming models from one formalism into another, it first has to be identified. Therefore, two steps should be taken. First, the semantics of the source and the target formalism should be well understood. Second, additional requirements on the (static) semantics should be made explicit. When bridging the semantic gap is not a straightforward affair, it is advisable to address a simplified version of the source metamodel first. Another possibility is first to relax the semantic requirements on the transformation. It can also be that the semantic gap is simply too large to be bridged at all.

We have used the term rewriting system ASF+SDF to implement a transformation from ACP without encapsulation to UML state machines. This required us to define metamodels for both ACP and UML state machines. We have created a metamodel for ACP and for UML state machines without history mechanism. The modular implementation of our transformation has proven to be useful for decreasing the complexity. Moreover this benefits reuse, extensibility, maintainability, and testability of the implementation.

Using the CASE tool Telelogic Rhapsody we can generate code to execute the acquired UML state machine and action dispatcher. In this way the execution of an ACP model can be simulated. Since our transformation preserves most structure of ACP models, UML tools can be used for visualizing this structure.

We performed several case studies using our implementation to illustrate our transformation of ACP models into UML state machines.

## 6.2 Directions for Further Research

We have considered ACP without the encapsulation operator ( $\partial$ ). The next thing to consider is the transformation of the encapsulation operator. This makes the semantic gap even larger. The encapsulation operator prevents certain actions from being executed, which cannot be expressed in a state machine. This requires an extension of the action dispatcher such that it forbids the execution of encapsulated actions. Moreover, the alternative composition is no longer non-deterministic. In ACP the alternative composition of an encapsulated and a non-encapsulated action, e.g.  $\partial_{\{a\}}(a) + b$ , rewrites to the non-encapsulated action ( $b$ ) only. In a structure preserving state machine, care has to be taken that the selection of a branch with an encapsulated action is prevented to avoid unwanted deadlocks. This gets even more delicate when an action cannot be executed itself but can communicate with another action.

**Acknowledgements.** This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

We would like to thank the anonymous reviewers for their comments which helped us improving an earlier version of this paper.

## References

1. FALCON, <http://www.esi.nl/falcon/>
2. van Amstel, M.F., van de Plassche, E., Hamberg, R., van den Brand, M.G.J., Rooda, J.E.: Performance analysis of a palletizing system. SE Report 2007-09, Department of Mechanical Engineering, Eindhoven University of Technology (2007)
3. Baeten, J.C.M.: A brief history of process algebra. *Theoretical Computer Science* 335(2-3), 131-146 (2005)
4. Object Management Group: Unified Modeling Language: Superstructure specification, version 2.1.1. Document – formal/2007-02-05, OMG (2007)
5. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes. In: de Bakker, J.W., Hazewinkel, M., Lenstra, J.K. (eds.) *Proceedings of the CWI Symposium*. CWI Monographs, vol. 1, pp. 89-138. Centre for Mathematics and Computer Science, North-Holland (1986)
6. Baeten, J.C.M., Weijland, W.P.: *Process Algebra*. Cambridge Tracts in Theoretical Computer Science, vol. (18). Cambridge University Press, Cambridge (1990)
7. van Beek, D.A., Man, K.L., Reniers, M.A., Rooda, J.E., Schiffelers, R.R.H.: Syntax and semantics of timed Chi. CS-Report 05-09, Department of Computer Science, Eindhoven University of Technology (2005)
8. Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., van Weerdenburg, M.: The formal specification language mCRL2. In: Brinksma, E., Harel, D., Mader, A., Stevens, P., Wieringa, R. (eds.) *Methods for Modelling Software Systems (MMOSS)*. Number 06351 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (2007)
9. Nicollin, X., Sifakis, J., Yovine, S.: From ATP to timed graphs and hybrid systems. *Acta Informatica* 30(2), 181-202 (1993)
10. Pardo, J.J., Valero, V., Cuartero, F., Cazorla, D.: Automatic translation of a timed process algebra into dynamic state graphs. In: *Proceedings of the 8th Asia-Pacific Conference on Software Engineering*, pp. 63-70. IEEE Computer Society, Los Alamitos (2001)
11. Cerone, A.: From process algebra to visual language. In: Lakos, C., Esser, R., Kristensen, L.M., Billington, J. (eds.) *Proceedings of the 23rd Conference on Application and Theory of Petri Nets*. *Conferences in Research and Practice in Information Technology*, vol. 12, pp. 27-36. Australian Computer Society (2002)
12. Bortnik, E.M., Mortel-Fronczak, J.M., Rooda, J.E.: Translating  $\chi$  models to UP-PAAL timed automata. SE Report 2007-06, Department of Mechanical Engineering, Eindhoven University of Technology (2007)
13. Pintér, G., Majzik, I.: Program code generation based on UML statechart models. *Periodica Polytechnica* 47(3-4), 187-204 (2003)
14. Niaz, I.A., Tanaka, J.: Code generation from UML statecharts. In: Hamza, M.H. (ed.) *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications*, pp. 315-321. ACTA Press (2003)

15. Crane, M.L., Dingel, J.: UML vs. classical vs. Rhapsody statecharts: Not all models are created equal. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 97–112. Springer, Heidelberg (2005)
16. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. *Information and Control* 60(1–3), 109–137 (1984)
17. Baeten, J.C.M., Basten, T., Reniers, M.A.: Algebra of communicating processes. Lecture notes (DRAFT) (2005)
18. Thompson, K.: Regular expression search algorithm. *Communications of the ACM* 11(6), 419–422 (1968)
19. Object Management Group: Meta Object Facility MOF 2.0/XMI mapping specification, version 2.1. Document – formal/05-09-01, OMG (2005)
20. ArgoUML v0.24 (Viewed January 2008), <http://argouml.tigris.org/>
21. Telelogic Rhapsody 7.1.1 (Viewed January 2008), <http://modeling.telelogic.com/products/rhapsody/index.cfm>
22. van den Brand, M.G.J., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF meta-environment: A component-based language development environment. In: Wilhelm, R. (ed.) CC 2001 and ETAPS 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
23. van Deursen, A.: An overview of ASF+SDF. In: van Deursen, A., Heering, J., Klint, P. (eds.) *Language Prototyping: An Algebraic Specification Approach*. AMAST Series in Computing, vol. 5, pp. 1–29. World Scientific, Singapore (1996)