Viewing CSP specifications with UML-RT diagrams*

Patrícia Ferreira, Augusto Sampaio and Alexandre Mota

[1]Centro de Informática
Universidade Federal de Pernambuco
P.O.Box 7851 50740-540 Recife-PE, Brazil

{pmf,acas,acm}@cin.ufpe.br

Abstract. *To precisely specify and reason about the properties of a system requires using formal methods like, for instance, process algebras. Complementary, semi-formal notations like UML are extensively used in practice to describe several architectural views of a system with the aid of modeling diagrams. In this paper we present an automated approach for translating specifications in the CSP process algebra into UML-RT models, in which we can describe both static and dynamic views of the system. The strategy is based on compositional rules that preserve the CSP semantics. We illustrate the systematic translation through an example.*

1. Introduction

Formal methods have demonstrated to be effectively applicable in the industrial development of critical systems. Nevertheless, formal methods gather less specialized developers than the industrial necessities. The consequence is that formal methods are generally considered too difficult or too expensive to be used in "ordinary" software development. Moreover, it is common that the formal specifications be misinterpreted or neglected in the subsequent phases of the system development, because usually there are conceptual gaps between the models used on these phases, and these models are often only informally related.

CSP [Roscoe et al. 1997, Hoare 1985], for instance, is a very attractive formalism to describe concurrent and dynamic aspects of computer systems. One of the fundamental features of CSP is that it can serve as a notation for describing concurrent and communicating processes at different levels of abstraction. Furthermore, it is possible to prove refinements and classical properties, as deadlock and determinism, as well as domain specific properties of CSP specifications using the FDR [Goldsmith 2001] refinement checker. However, CSP lacks graphical visualization; therefore it can be difficult to understand and to be used by non-specialists. Hence it can be costly and error-prone to informally associate the dynamic behaviour of CSP constructions with structural elements of the design phase such as components and independent processes.

On the other hand, graphical modeling notations are tremendously used to structure and visualize systems, but usually do not embody a consolidated formal foundation to allow reasoning about classic and domain specific properties. Even semi-formal graphical notations such as UML [OMG 2003] and ROOM [Selic 1993] do not offer a reasoning framework to prove classical or domain specific properties. Some initiatives have been proposed to give formal semantics to UML and to

*This project is funded by Motorola Inc.

some of its profiles [Fischer et al. 2001, Ramos et al. 2005], through translations of diagrams and elements of UML into specifications in formal notations, such as CSP, Z [Spivey 1992] and *Circus* [Sampaio et al. 2002]. However, these initiatives address only a small subset of UML.

The reverse process, translating CSP specifications into UML graphical models preserving the formal semantics, permits that the design of an application be driven and constrained both by the modeling features available in UML, as its architectural and behavioural style rules, and the properties imposed by the source CSP specification [Medvidovic et al. 2002]. Although these UML models cannot be used to reason about complex properties, the formal CSP specifications that give rise to these models carry the desired properties.

This paper presents compositional rules to systematically map CSP specifications into UML-RT models. Although formal proofs are suggested as future work, the rules are intended to preserve semantics of the source model. UML-RT [Selic and Rumbaugh 1998, Lyons 1998] is a UML profile that is suitable for modeling complex event-driven systems, such as mobile phone applications. This profile has all possible elements and diagrams from the UML standard [OMG 2003], in addition to some specific elements from ROOM [Lyons 1998, Selic 1993], which allow modeling complex dynamic structures and dynamic relationships between them. As a result, UML-RT allows representing the main behavioural and structural concepts from CSP through its diagrams. Furthermore, the formal semantics inherited from ROOM allows generating code, making it possible also to animate and test CSP models through translation. The CSP notation under consideration here is the one described in [Roscoe et al. 1997].

This translation makes it possible to bridge the gap between formal modeling and system analysis. A major advantage is the possibility to associate the system functionalities with structural elements, such as components and independent processes, and to present their interactions through a visual model, with preservation of the formal semantics. This abstract visual model can then be formally refined using sound transformation laws for UML-RT [Ramos et al. 2005]. The design becomes incrementally more concrete, with the advantage of having a formal basis in its origin. However, not everything should be translated into graphical notation. Certain parts fit better as textual form, such as constraints representing invariants and pre or post conditions, for instance.

This work is being developed in the context of a cooperation project between the Federal University of Pernambuco and Motorola. Within this project, the generated UML-RT models are used both to automate test cases generation [Cartaxo et al. 2006] and as an analysis model for mobile feature implementations.

The next two sections give a brief overview of CSP and UML-RT. The transformation rules are presented in Section 4, where we also briefly discuss tool support and present an example to illustrate the translation strategy. Finally, Section 5 draws conclusions and discusses related and future works.

2. CSP Overview

The process algebra CSP (Communicating Sequential Process) [Roscoe et al. 1997] is a formal language primarily designed to model the behaviour of concurrent and distributed systems. CSP has three main elements: events, processes and operators. Events are abstractions of real world actions. For example, the event

turn.On.Button

can be used to model the real action of turning on the button of a radio. Besides events, CSP provides channels that are used as a collection of events. The main difference between events and channels in CSP resides in their declarations. The declaration

channel a

introduces a single event, while

channel e : *Int*

introduces the channel e that can communicate any event that carry an integer data value. In particular, the event $e.2$ is one of the elements provided by the declaration of channel e. The occurrence of an event characterizes a communication, where at least two participants are involved. In general, a participant is a process but when there is no explicit process, the participant is the external environment that interacts with the processes. Processes are behavioural description units, which can be combined using operators and events to produce complex behaviours. CSP uses a synchronous communication model which means that all participants must be ready for the communication to occur. Here we use the term *alphabet* to denote the set of events that appear in a process description (body). The entire alphabet in a specification is represented by Σ. The order and availability with which events occur are determined by the CSP operators.

Here, we consider the following simplified CSP process grammar:

$$
\begin{array}{rcl}
P & ::= & \textbf{\textit{STOP}} \\
 & | & \textbf{\textit{SKIP}} \\
 & | & P \\
 & | & a \rightarrow P \\
 & | & P \backslash \mathrm{C} \\
 & | & P[\mathrm{R}] \\
 & | & P; P \\
 & | & P \sqcap P \\
 & | & P \Box P \\
 & | & P \ || \ P \\
 & & C
\end{array}
$$

Figure 1. Some CSP process definitions

where P is a process name, a is an event of the process alphabet, C is a set of events, and R is mapping relation with the form $(a \leftarrow b)$, where a and $b \in \Sigma$. There are other constructions, but these are the most relevant for this paper.

The processes ***STOP*** and ***SKIP*** are unit processes: they alone determine a useful behaviour. The process ***STOP*** models a broken situation (a deadlock), whereas ***SKIP*** captures the notion of a successful termination.

The prefix process ($a \rightarrow P$) waits indefinitely for event a be allowed by the environment and when it occurs, the process behaves like P. The $\rightarrow$ operator always takes a single event on the left-hand side and a process on the right-hand side.

The hiding operator takes a set of events and a process as arguments, and makes the events invisible in the process. These events continue happening inside the process, but other processes and the environment cannot see them. The renaming operator is useful to change the name of events (or to create copies of a process with different alphabets). In what follows, the process P executes the event a continuously. The process Q, although defined in terms of the process P, renames all occurrences of a with c.

$$P = a \rightarrow P$$
$$Q = P[a \leftarrow c]$$

The other CSP operators are used to combine processes. The deterministic (or external) choice operator $\Box$ allows the evolution of a process to be defined as a choice between two component processes. The non-deterministic (or internal) choice operator $\sqcap$ allows the evolution of a process to be defined as a choice between two component processes, but does not give the environment any control over which of the component processes will be selected. The sequential composition ($P;\ Q$) builds a process that behaves like P until a successful termination occurs. In this case, the process Q is allowed to occur.

Finally, processes can be combined to describe the architecture of systems through parallel composition. The parallel composition, denoted by $\underset{C}{||}$, is used to put two processes in parallel, in which case they should synchronize in all communication events in the set C. For instance, the process $Q \underset{\{ch\}}{||} R$ describes de parallel composition of processes Q and R, where they should execute all events from channel ch simultaneously. Events outside C should be executed independently on each process. In particular, when A is empty we have pure interleaving, that is, $P \underset{\{\}}{||} Q \equiv P \,|||\, Q$.

3. UML-RT Overview

UML-RT [Selic and Rumbaugh 1998, Lyons 1998] is a conservative extension of UML. It contains specific conceptual elements of ROOM (Real-Time Object-Oriented Modeling language) [Lyons 1998] that make it possible to model architectures and dynamic relationships of real-time event-driven systems. A capsule, for instance, is a stereotype of UML active class adjusted to the ROOM actor concept. Capsules, like processes, are behavioural description units, with specialized semantics to represent components or independent processes, and can have multiple interfaces, named ports. Capsules communicate among themselves exclusively through messages, which should flow between connected ports of capsules. Ports have output signals for sending messages, and input signals for receiving messages. In order for two ports to be connected, the ports must be compatible; that is, every output signal in a port must be

an input signal in the other port. An event represents the reception of a message by a capsule.

Ports realize protocols, which define the input and output signals. Protocols can play two or more roles, in accordance with the ROOM standard. However, the UML-RT specification commonly uses binary protocols, involving just two roles. Only one role, named *Base* role, needs to be specified. The other one, *Conjugate* role, can be derived from the *Base* role simply by inverting the incoming and outgoing signal sets. In this way, ports are run-time entities that provide full two-way interfaces to capsules. Furthermore, a protocol fixes the data types and the order of messages flowing between connected ports. This order is useful to show the potential interactions of a capsule instance with the external environment. In a sense, a protocol captures the contractual obligations that exist between capsules [Selic and Rumbaugh 1998].

UML-RT offers capsule structure diagrams to represent the composite structure of capsules (see Figure 2). It shows both the ports, which are the communication points of capsule, and implicit containment relationships between capsules and capsule roles (contained capsules). Ports can be public or protected. Public ports are located on the boundary of the structure diagram, and these ports may be visible both from outside and inside the capsule. Protected ports are not visible from the outside of a capsule since they are not part of the capsule interface. Only public ports are shown on capsule roles. Furthermore, ports can be *end* or *relay*. Messages sent to an *end* port can be processed directly by the capsule behavior (represented by a state diagram, described later). *Relay* ports are used to forward messages to other ports, but these messages cannot be processed by the capsule behavior. If a *relay* port is not connected to another port, all messages arriving on that port are lost. Outside the capsule there is no distinction between *relay* and *end* ports. Figure 2 shows a structure diagram of a capsule P with capsule role Q, public *end* port a, public *relay* port b, and protected *end* port c. The container capsule Q has two public ports, d and e, but it is not possible to know whether these ports are *relay* or *end*.

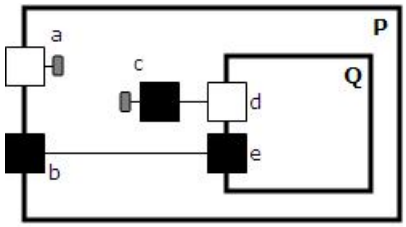


Figure 2. Structure diagram

UML-RT offers state machine diagrams to model the internal behaviour of a capsule when messages arrive on its ports. A state machine is a directed graph of states that are connected by transitions. Except for the initial transition, which is automatic, the other transitions in a state machine are triggered by the arrival of messages on a capsule port. As mentioned previously, only the events incoming through *end* ports can be processed by the state machine. Each state can have its own internal state machine. This allows constructing complex behaviours. There is no final state in capsule state machines, because capsules are active classes that never terminate.

In general, a transition has the form $p.e[g]/a$, where e is an input signal, p is

the port through which the message arrives, g is a boolean expression (named guard condition), and a is an action. If the event occurs on port p, and the guard evaluates to *True*, then the action is executed, possibly changing the current state. If a transition has the form $p.e/a$ we consider that the guard is *True*. Here, for notation compatibility with CSP, we use the notation $p.e?x/a$ to denote a transition that accepts the message x through signal e of the port p, and subsequently executes the action a; and the notation $p.e!F$ to denote the action of sending a message F through the signal e of the port p.

Figure 3 shows the state machine diagram of the capsule P, named S_P. The black circle at the top left is the initial state S_0, and its outgoing transition is automatic; therefore, no incoming event is necessary for this transition, but such transition can execute an action, if necessary. After the initial transition, the state S_1 becomes the current state. The outgoing transition from the current state accepts a message x through signal *in* of the port a, and executes the action that communicates a new message through signal *out* of the port c. Following, the state S_2 becomes the current state.

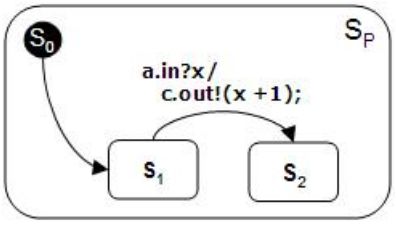


Figure 3. State machine diagram

The communication between capsules can be either synchronous or asynchronous. Asynchronous operation calls are stored in an event-queue of the receiver, and the sender remains free to execute its next actions. The receiver always checks the first element in its event-queue. If it is ready to execute a trigger involving this first call, it should perform the associated transition and continue its execution; otherwise, it suspends. On the other hand, synchronous operation calls involve a *rendezvous* between the sender and the receiver: when the sender executes the operation call, it is suspended until the receiver synchronizes with it (that means executing a corresponding trigger). If the synchronization proceeds, a return value is sent back to the sender, after which both the sender and the receiver resume their own executions. Otherwise, an internal timeout sets free the sender, but the message will be lost. The uses of synchronous and asynchronous messages are design decisions, and do not depend on port or protocol configurations.

Finally, it is possible to use capsule roles dynamically. Capsule roles are strongly owned by the container capsule, and cannot exist independently of the container capsule. By default, capsule roles are fixed, meaning that they are created automatically when their containing capsule is created, and are destroyed when the container is destroyed. However, some capsule roles in the structure cannot be created at the same time as their containing capsule. Instead, they may be created subsequently, when and if necessary, by the state machine of the container, and they can be destroyed before the container is destroyed. These capsule roles are named *optional*. In order to use an optional capsule role it is necessary to define a slot capsule role (pointer) in the container structure diagram. This slot is not active, and it should have de same set of

public ports (that is, the same interface) as the intended optional instance.

4. Transformation Rules

This section presents a systematic strategy for translating CSP specifications into UML-RT models. We propose rules that intend to be independent and compositional, in which case it is possible systematically obtaining capsules from processes. The exhaustive application of these rules translates a specification into a compound UML-RT model.

The approach presented here translates each CSP process into a UML-RT capsule with the same name, taking advantage of the concepts of reuse and modularity in the context of CSP processes. Each channel usage in the process alphabet is mapped into a port with the same name in the capsule structure diagram, and the events occurring in the process should determine the transitions in the capsule state machine.

Actually, the translation of CSP processes involves capsules and protocols, because the communication events occurring among capsules should be transmitted through ports, which realize protocols. The protocol signals should carry objects that correspond to values of CSP events. A possible mapping would be to create a protocol for each channel defined in the specification, or for each data type, in which case the occurrence of a channel in a process implies in the creation of a port that realizes the protocol of the channel data type. Instead, for simplicity, we use a unique protocol to transmit all messages between capsules, as long as its signals accept any type of object. This decision is merely structural, and does not affect the communication between capsules because CSP events can be represented by synchronous messages in UML-RT, and the communication mode of these messages is not influenced by the representation of protocols. In this way, our rules consider only the construction of capsules, since the protocol is fixed. Here, this protocol is named *CSPMessageProtocol*.

The mapping of data type declarations is not included here because we assume that these are mapped into simple UML classes, since we consider that each data type represents a class of messages used by the system. Compound data types must also be translated as a unique class, which accepts any possible value of each type involved in the composition.

Recall that, in UML-RT, capsules can have *Base* and *Conjugate* ports, for sending and receiving messages concerning the signals orientation. However, the capsules generated by this translation strategy have been simplified so that they contain only *Conjugate* ports to represent the channels on processes, except in especial cases described later. In our translation strategy, the external environment, which is implicit in CSP specifications, is made explicit in the generated UML-RT models. Furthermore, only the external environment, which is connected to capsules through *Base* ports, can send messages to capsules, using output signals defined with a *Base* role. This decision was taken in order to simplify the UML-RT model, but it does not change the semantics of the model, since it is possible to duplicate signals or represent the CSP specification without event orientation.

The translation strategy can be thought of as a term rewriting system that exhaustively applies the rules to progressively replace CSP expressions with UML-RT

capsules and protocols. The first set of rules is concerned with simplifying CSP equations so that all equations have the simple form:

$$P = N_P$$
$$P = a \rightarrow N_P$$
$$P = N_P \; uop \; args$$
$$P = N_{P1} \; bop \; N_{P2}$$

where *uop* is a CSP unary operator (*hiding* or *renaming*), and *bop* is a CSP binary operator (*external choice*, *internal choice*, *sequential* or *parallel*). In such a form, the right-hand side of an equation can be a process name (N_P); a prefix process involving a single event and a process name; a unary process operator with a process name and a set of elements as arguments; or, finally, a binary process operator with two process names as arguments. The following is an example of a rule in this category.

Rule 1 *Prefix Expression Simplification*

$$P = a \rightarrow Exp \quad \Longrightarrow \quad \begin{array}{l} P = a \rightarrow N_P \\ N_P = Exp \end{array}$$

Consider that *Exp* is a CSP process expression, excluding the simple expression formed of a process name. This rule replaces *Exp* with a name of a new process (N_P) and introduces a new equation, as expected. □

In order to make the translation rules more readable, we consider in this paper processes without arguments.

Concerning the translation of CSP equations, when a process P behaves as a process Q (like $P = a \rightarrow Q$, for example), the corresponding capsule P must behave like a capsule Q. As a first intuition, the idea would be that capsule P contain capsule Q, in which case P replicates ports of Q. In this way, P would assimilate the alphabet of process Q, and forwards all messages involving the capsule Q and the external environment, thus simulating the behaviour of process Q. Figure 4 shows the structure and state machine diagrams of capsule P for this translation alternative. In this case consider that the transmitted messages x are empty values, since the involved channels do not carry values. After capsule P receives the event $a.in?x$, it simulates the behavior of the process Q, just forwarding all events arriving from the replicated ports to capsule Q, since the capsule Q is the result of the translation of process Q. However, this alternative would not be compositional if there were mutual references between the original processes (like $P = a \rightarrow Q$ and $Q = a \rightarrow P$, for example), because UML-RT does not allow mutual containments of capsules. Actually, not even self recursions could be resolved by this strategy. Consider the process

$$P = a \rightarrow a \rightarrow P$$

applying Rule 1, this equation is rewritten to

$$P = a \rightarrow N_P$$
$$N_P = a \rightarrow P$$

that also results on mutual references between the processes.

The absence of mutual containments on capsules is a limitation of UML-RT to avoid infinite recursion when instantiating capsules. As a solution, we use optional

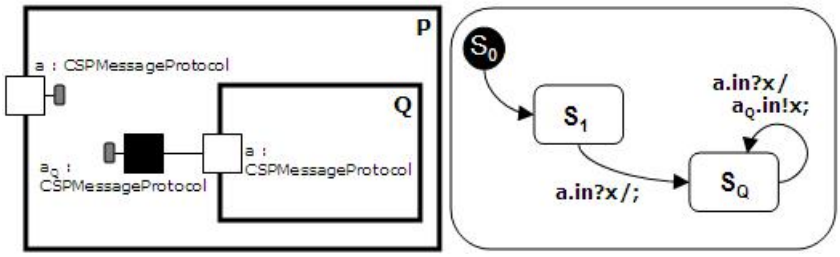


Figure 4. An intuitive, but limited, approach

capsules in our translation strategy. These capsules can be created and used until a new configuration becomes necessary, at which case they are removed, and new optional capsules simulating the new behavioural structure are created. In this way, the system can have several capsules executing simultaneously according to different configurations, such as parallel or sequential composition. Furthermore, this approach improves the allocation of system resources and makes it possible to control complex dynamic systems, thus avoiding infinite recursions. Additionally, it is possible to instantiate optional capsules with arguments.

The strategy presented here makes all generated capsules to be used as optional capsule roles of a unique capsule that controls the others. This controller capsule, named *SystemController*, replicates all public ports of its capsule roles, regardless of these capsules being active or not. It is similar to providing the entire alphabet of the specification. The capsule *SystemController* is connected directly to the external environment, simulating the interface of the entire system.

Additionally, every generated capsule has a specific *Base* port to inform the *SystemController* about new (behavioural) configurations, in which case the *SystemController* removes the instance of this capsule and creates new instances of other capsules according to the intended behaviour. This special port should transmit values that represent the relevant CSP process being translated. The port uses the protocol *CSPBehaviorProtocol*, whose signal *term* carries the process representation. In the implementation of these rules we use a Java class to represent the processes. For improving readability we use the CSP expressions themselves as arguments to the signal *term*. This port is not replicated by *SystemController* because it is an internal control element; the external environment does not need to know about this internal replacement.

The dynamic configuration of capsules should be controlled by the capsule *SystemController*, which encompasses the expanded equation of the system in each stage. *SystemController* has a local variable that stores this expanded equation and is always updated when some internal capsule informs a new process term. All actions of *SystemController*, such as forwarding an incoming message to a capsule role, creating or removing capsule roles, must consider this variable. Although the capsule *SystemController* centralizes the control flow, its construction is also compositional. In each rule below we show how its state machine is progressively constructed to handle the overall control flow.

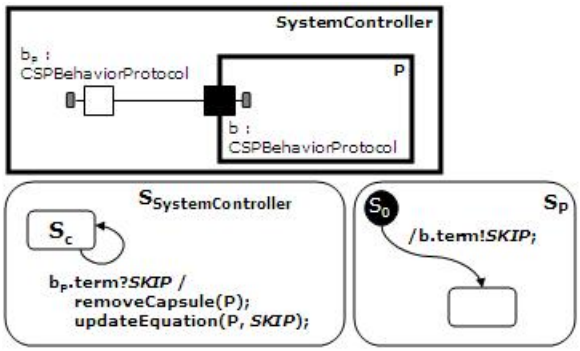
We now present some rules for translating CSP processes into UML-RT capsules. Consider that these processes are in the simplified form discussed previously,

and that the names of ports that realize *CSPBehaviorProtocol* are not in the system alphabet. Furthermore, consider that the state S_C is the current state in the state machine of *SystemController*, and that the capsule roles in structure diagram of *SystemController* are actually slots to optional capsule roles, that is, they are not active. The container capsule uses private *end* ports, which are connected with the ports of the capsule roles, to make it possible to receive and to send messages through ports of capsule roles according to the semantics of the current configuration of capsules.

The following rule deals with the ***SKIP*** process.

Rule 2 *Skip Process Transformation*

$P = \textbf{\textit{SKIP}} \quad \Longrightarrow$



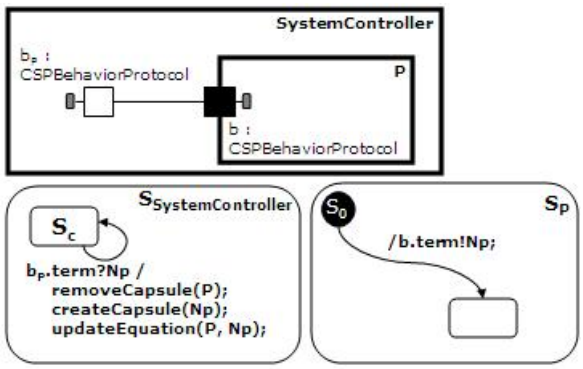
The process that behaves as ***SKIP*** just informs its new state (a successful termination) to the external environment. A new transition is created in the current state of *SystemController* to manage this new behavioural configuration. In this case, the capsule role P is removed and the expanded equation is updated, just replacing its reference to the process P with a reference to ***SKIP***. In this situation, a new capsule role is not created, since the process ***SKIP*** is a successful termination. □

Actually, a new transition should be created in *SystemController* for each possible simplified process equation, and not only for ***SKIP***, since the container capsule cannot previously know the future behaviour of its capsule roles. Here we show only the transitions that will actually execute.

In the following rules, let N_P be an arbitrary process name. The next rule deals with non-guarded process names.

Rule 3 *Named Process Transformation*

$P = N_P \quad \Longrightarrow$



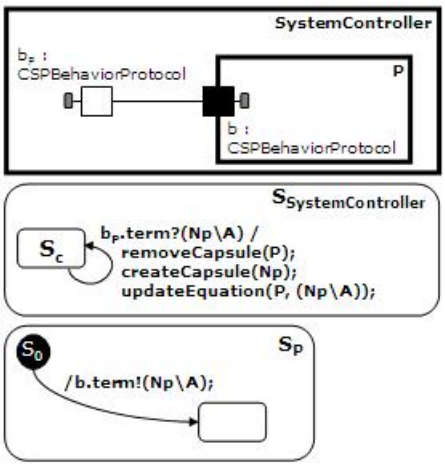
When a process behaves as a simple process (that has no CSP operator involved), the generated capsule informs the new behavior to the external environment. The

transition in *SystemController* that manages this situation removes the capsule role P, creates a new capsule role N_P, and updates the expanded equation. □

The following rule deals with the hiding operator.

Rule 4 *Hiding Transformation*

$$P = N_P \setminus A \quad \Longrightarrow$$

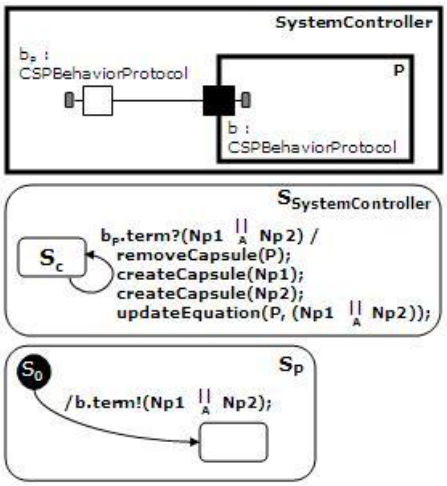


The capsule that represents a hiding process should send a process name and a set of event names (represented by the set A). The transition in *SystemController* that manages this situation removes the capsule role P, creates a new capsule role N_P, and updates the expanded equation. In this case the alphabet A will be considered to hide events on capsule role N_P. □

The following rule deals with alphabetized parallel composition.

Rule 5 *Parallel Transformation*

$$P = N_{P1} \underset{A}{||} N_{P2} \quad \Longrightarrow$$

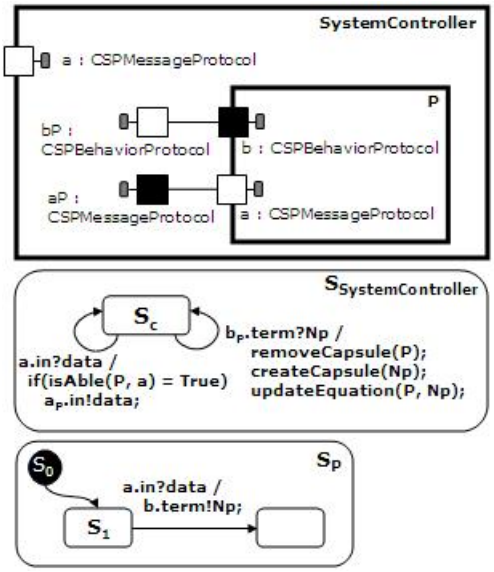


When a process behaves as a parallel composition, the generated capsule should send a process name for each of the two process arguments, and a set of event names (described on set A) to represent the synchronization alphabet. In this way, we expect to represent other variations of parallelism, such as interleaving, which uses an empty set as alphabet. The transition in *SystemController* removes the capsule role P, creates new capsule roles for each process name in the parallel composition, and updates the expanded equation. □

The following rule deals with the prefix operator.

Rule 6 *Prefix Transformation*

$P = a \rightarrow N_P \quad \Longrightarrow$



The occurrence of the channel a in the process P generates a port with the same name in the structure diagram of capsule P. This port implements the protocol *CSPMessageProtocol*, as mentioned previously. The CSP event results in an outgoing transition in the current state. In this case, the state S_1 is the current state of the capsule P, since the initial transition from S_0 to S_1 is automatic. *SystemController* replicates each public port of capsule P, and creates transitions to manage the events arriving from these replicated ports. When *SystemController* receives an event through its port a, it verifies the availability of the capsule P, according with the expanded equation. If the capsule P is allowed to synchronize on this event, the message is forwarded to P through port a_P. When the capsule P receives the event through its port a, it informs its new behaviour, and its internal state changes. As well as the Rule 3, a process name is transmitted by the port b, and a transition in *SystemController* manages the new behaviour. □

Now, we show a progressive simulation of the capsule *SystemController* with the example in Figure 5. Consider the capsules roles in structure diagrams representing the active capsules in that moment. Furthermore, we show only transitions executing during that moment in the state machine diagrams. Consider S_C the current state of the capsule.

$$\begin{gathered} P = Q \underset{\{\}}{||} R \\ Q = a \rightarrow \textbf{\textit{SKIP}} \\ R = \textbf{\textit{SKIP}} \end{gathered}$$

Figure 5. A CSP example

In Figure 6(a), *SystemController* has a unique capsule P; therefore, the expanded equation equals the process P. The container capsule has a port a since it is in system alphabet. Suddenly, *SystemController* receives a new process term through port b_P, in which case it removes the instance of capsule P and creates new instances of capsules Q and R. The expanded equation is updated from $Q \underset{\{\}}{||} R$.

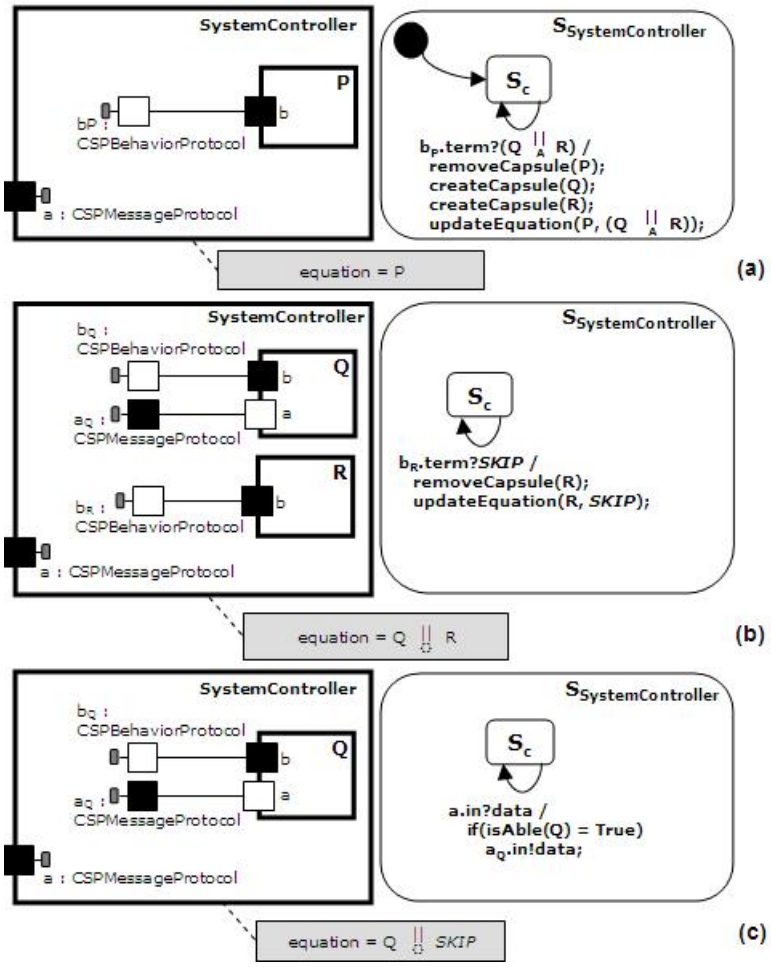


Figure 6. An example of use of capsule *SystemController*

After that, in Figure 6(b), *SystemController* is prepared to evaluate the capsules *Q* and *R*. When the capsule *R* informs a new (behavioural) structure, *SystemController* removes it and updates the expanded equation. Now, only the occurrence of the process *R* (the right-hand side of the actual equation) is replaced with the new term (***SKIP***). In this situation, a new capsule is not created, since the process ***SKIP*** is a successful termination.

Finally, in Figure 6(c), *SystemController* receives an incoming event from the external environment. The method *isAble* verifies the conditions for the capsule roles to receive the incoming messages. In this case, the capsule role *Q* is allowed to receive the message, but it depends on the current configuration of the *SystemController*, which can have more instances of capsule *Q* or other capsules that have a port *a*.

The generation of UML-RT models from CSP specifications was automated by a tool that systematizes the application of the transformation rules. The tool reads CSP specifications, starts the Rational Rose Real Time [Rational/IBM b] and, using the extensibility mechanism of this application, outputs the UML-RT models applying the rules in a compositional way. Figure 7 shows the graphic user interface of the tool, and Figure 8 shows the generated UML-RT model in the Rational Rose Real Time application for the example in Figure 5.

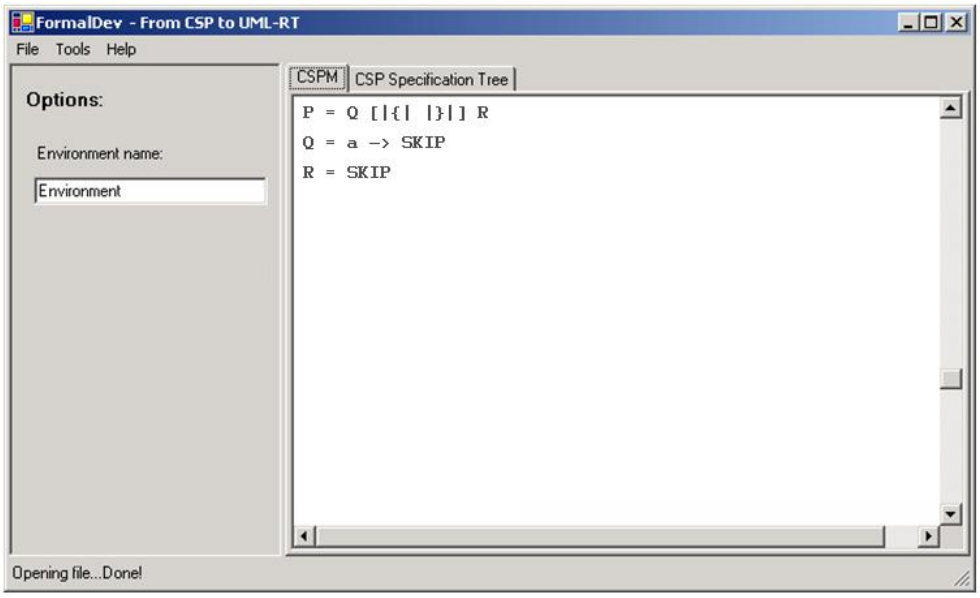


Figure 7. Tool interface

5. Conclusion

We have proposed rules to map CSP processes into UML-RT capsules. This translation benefits from similarities between processes and capsules, because both have behavioural views and can be defined in a compositional way. On the other hand, while CSP has a rich set of operators to combine processes, UML-RT has no operator to build capsules from existing ones. So we had to encode the semantics of each CSP operator during the translation. For simplicity, we have presented a small subset of the rules, and we consider processes without arguments. The complete repertoire of rules can be found in [Ferreira 2006], and also more considerations about the semantics of CSP events, specially which involve multiple synchronizations and events with parameters.

This strategy offers a possible approach to bridge the gap between formal modeling and system analysis. Other initiatives [Fischer et al. 2001, Ramos et al. 2005] propose the inverse process, to give a formal semantics to UML through translations into specifications in formal notations. However, these initiatives address only a small subset of UML. As alternative approaches, there are programming languages offering support by implementing concurrent systems specified in CSP, as CTJ [CTJ] and JCSP [JCS], but due to the size of real concurrent systems their implementation can be problematic and with communication patterns usually very complex. Moreover, the visualization of the system structure is usually as difficult as the CSP specification. Our approach has the advantage of the diagrammatic representation in addition to the code generation, as discussed previously.

Although the rules generate an excessive number of capsules, we plan to use refactorings on capsules [Ramos et al. 2005], considering real-time modeling concepts and also preserving the properties of the original model, to make the design incrementally more concrete, with the advantage of having a formal basis in its origin. These refactorings can also reinforce the similarities between UML-RT capsules and CSP processes. For example, a sequence of prefixes in a same process, which

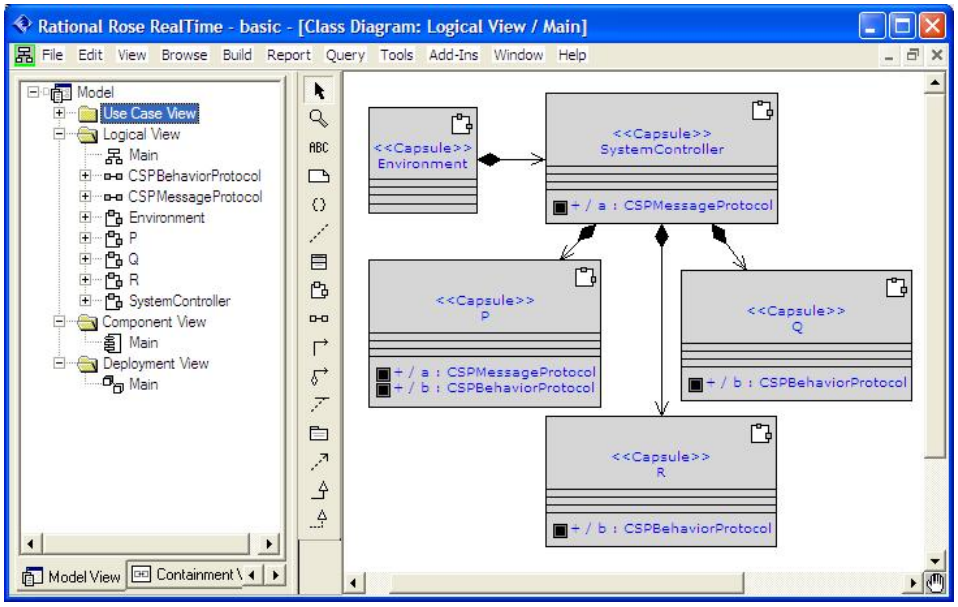


Figure 8. Generated model in the Rational Rose Real Time

originates several capsules, can be reduced as a single capsule.

The exhaustive application of these rules translates a specification into a compound UML-RT model. The translation strategy is systematic, and was automated by an application that takes CSP specifications and outputs models to the Rational Rose Real Time tool [Rational/IBM b]. It is possible to generate code through the generated model, making it possible also to animate and test CSP models thought translation. The interface of the tool that implements our strategy is presented in Figure 7.

These rules can be adapted to UML 2 [OMG 2004], with possible improvements on diagrams. Despite of the fact that UML 2 uses several concepts from UML-RT, its elements and diagrams are still ambiguous and unclear [Rational/IBM a, Eriksson et al. 2003], hence we have chosen to work with UML-RT. Actually, all relevant concepts to represent CSP specifications on this strategy are on UML-RT, which has more consolidated tool support, and whose capsule and protocol concepts are clearer and more intuitive than that of UML 2.

6. Acknowledgment

We would like to thank Joabe Jesus for developing the tool support, and Motorola Inc., and particularly the members of the BTCRD research group, for their support and help during the development of this work.

References

Communicating Sequential Processes for Java (JCSP). `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

Communicating Threads in Java (CTJ). `http://www.ce.utwente.nl/`.

Cartaxo, E., Lima, E., Machado, P., and Helena, L. (2006). Test Case Generation by means of UML Sequence Diagrams and Label Transition System for Mobile Phone Applications.

Eriksson, H.-E., Penker, M., and Fado, D. (2003). *UML 2 Toolkit*. John Wiley & Sons, Inc., New York, NY, USA.

Ferreira, P. (2006). Translating CSP processes into UML-RT diagrams (in Portuguese). Master's thesis, Centro de Informatica, Universidade Federal de Pernambuco.

Fischer, C., Olderog, E.-R., and Wehrheim, H. (2001). A CSP view on UML-RT structure diagrams. In Hussmann, H., editor, *Fundamental Approaches to Software Engineering. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6. 2001 Proceedings*, volume 2029, pages 91–108. Springer.

Goldsmith, M. (2001). *FDR: User Manual and Tutorial, version 2.77*. Formal Systems (Europe) Ltd.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Lyons, A. (1998). UML for Real-Time Overview.

Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., and Robbins, J. E. (2002). Modeling software architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57.

OMG (2003). *OMG Unified Modeling Language Specification, version 1.5, OMG document formal/03-03-01*. Object Management Group.

OMG (2004). *UML 2.0 superstructure specification, version 2.0, documents ptc/03-08-02 and ptc/04-10-02*. Object Management Group.

Ramos, R., Sampaio, A., and Mota, A. (2005). A Semantics for UML-RT Active Classes via Mapping into *Circus*. In *FMOODS*, pages 99–114.

Rational/IBM. IBM Rational Software Modeler.

Rational/IBM. Rose Real-time Development Environment.

Roscoe, A. W., Hoare, C. A. R., and Bird, R. (1997). *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Sampaio, A., Woodcock, J., and Cavalcanti, A. (2002). Refinement in *Circus*. In Eriksson, L. and Lindsay, P., editors, *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag.

Selic, B. (1993). An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 321–330, Ottawa, Canada. Elsevier Science Publishers B.V., Amsterdam, Netherland.

Selic, B. and Rumbaugh, J. (1998). Using UML for Modeling Complex Real-Time Systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK. Springer-Verlag.

Spivey, M. (1992). *The Z Notation: A Reference Manual. second edition.*