University of Twente

department of Electrical Engineering

Control Software Design and Safeguarding with the support of UML and CT

P.M. Visser

M.Sc.Thesis

Supervisors	Prof.dr.ir. J. Van Amerongen
	Dr.ir. J.F. Broenink
	Ir. G.H. Hilderink
	Ir. D. Jovanovic

September 2002
Report Number 19CE2002
Control Engineering
Electrical Engineering Department
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

This thesis will address the use of the Unified Modeling Language (UML) and Communicating Threads (CT) as a tool for designing control software. Since control systems are concurrent systems, the main focus is to explore the suitability of the concurrency model of UML. The CT-paradigm, developed by Gerald Hilderink, comprehends a special concurrency model for developing real-time software, which allows reasoning about concurrency issues and real-time behavior in an elegant way. Therefore it is interesting to research the integration of the CT-paradigm in UML. As a practical use-case, a safeguard is designed for a mechatronic setup.

UML is a third-generation object-oriented modeling language that rigorously defines the semantics of the object metamodel and provides a notation for capturing and communicating of object-structure and behavior. The UML provides tools to model various aspects of software design at a high level of abstraction. This coincides with the CT-paradigm that provides reasoning at a high level of abstraction about real-time and parallel software design. The UML tool Rhapsody is used for developing UML models, including the safeguard design.

CT is an object-oriented software package. This package provides a clear way of concurrent programming based on the theory of *Communicating Sequential Processes* (CSP). With CSP a concurrent system can be described by concurrently running processes that communicate with each other via channels. CTC++ is the implementation of the package in C++.

A comparison between UML and CT in terms of data transfer, synchronization and concurrent behavior is made. The comparison shows that in UML concurrency is defined on objects and on their operations. UML uses *active* –and *passive* objects to express concurrency and has no native semantical means to express priorities. Scheduling is thread based and the behavior of applications depends on underlying threading mechanism. CT provides channels to let processes communicate with each other. Channels encapsulate synchronization, concurrent behavior and data transfer. CT has extensive means to express concurrent behavior at a high level of abstraction. CTC++ uses 'fair' scheduling and its behavior is independent of a threading mechanism.

CTC++ can be successfully used as a package in Rhapsody and the approach to create the CTC++ package is given. Object model diagrams are analyzed for the use of communication diagrams and composition diagrams. The analysis shows that object model diagrams are suited to be used as communication diagrams but are not suited to be used as composition diagrams. A recommendation is given to create a tool, which is basically UML compliant but extended with symbols to express concurrent behavior. Statechart –and activity diagrams and the implementation mechanism of Rhapsody to create these diagrams are analyzed. An approach about the usage of channels, parallel constructs, and alternative constructs in statecharts is given or suggested. A different implementation mechanism is recommended in order to better integrate CTC++ in statechart –and activity diagrams.

A safeguard based on both UML and CT concepts is designed for a mechatronic setup called JIWY. JIWY is a little tabletop robot with 2 rotational degrees of freedom mounted with a camera. A detailed description of JIWY is given and specifications and requirements for the safeguard are extracted. The safeguard consists out various safeguard monitors, which monitor the mechatronic setup for specific hazards. The behavior of the safeguard monitors is specified in statechart –and activity diagrams. The object model diagrams depict the communication diagram for the design. The main advantage is that the use of channels in combination with the object model diagram makes the design scalable and testable. The disadvantage is that the object model diagrams become large, a lot of objects, and are difficult to read. A recommendation is given to implement missing features on JIWY so that the safeguard can be tested.

Samenvatting

Dit onderzoek richt zich op het gebruik van de Unified Modeling Language (UML) en Communicating Threads (CT) als een gereedschap voor het ontwerpen van regel software. Aangezien regelsystemen concurrent zijn ligt het aandachtspunt in het verkennen van het concurrency model van UML. Het CT-paradigma ontworpen door Gerald Hilderink bevat een speciaal concurrency model voor het ontwerpen van real-time software, welke het redeneren over concurrency zaken en real-time gedrag in een elegante wijze mogelijk maakt. Dit maakt het interessant om de integratie van het CT-paradigma in UML te onderzoeken. Als praktisch voorbeeld wordt een safeguard voor een mechatronische opstelling ontwerpen.

UML is een derde generatie object-georiënteerde taal die de semantiek van het object metamodel strict definieërd en voorziet in een notatie voor communicerend en object-gestructureerd gedrag. De UML voorziet in gereedschappen om verschillende aspecten van software ontwerp op een hoog niveau van abstractie te modelleren. Dit valt samen met het CT-paradigma, welke voorziet in het redeneren op hoog abstractie niveau van real-time en parallel software ontwerp. Het UML programma Rhapsody wordt gebruikt voor het ontwerpen van UML modelen, inclusief het ontwerp van de safeguard.

CT is een object-georiënteerd software pakket. Dit pakket voorziet in een duidelijke manier van concurrent programmeren op basis van de *Communicating Sequential Processes* (CSP) theorie. Met CSP kan een concurrent systeem worden beschreven door concurrent executerende processen die met elkaar communniceren via kanalen. CTC++ is de implementie van het pakket in C++.

Een vergelijk tussen UML en CT in termen van data overdracht, synchronizatie en concurrent gedrag wordt gemaakt. Dit vergelijk toont aan dat UML concurrency definieërd op objecten en hun operaties. UML maakt gebruik van actieve –en passieve objecten om concurrency uit te drukken en heeft van nature niet de semantiek om prioriteiten uit te drukken. Scheduling is gebasseerd op threads en het gedrag van applicaties is afhankelijk van het onderliggende threading mechanisme. CT verschaft kanalen om processen met elkaar te laten communiceren. Kanalen bevatten synchronizatie, concurrent gedrag en data overdracht. CT heeft uitgebreide middelen om concurrent gedrag op een hoog niveau van abstractie te beschrijven. CTC++ gebruikt een ‘eerlijke’ scheduling and het gedrag is onafhanklijk van een threading mechanisme.

CTC++ kan succesvol worden gebruikt als package in Rhapsody en de methode om de CTC++ package te maken wordt gegeven. Object model diagrammen worden geanalyseerd voor het gebruik van communicatie diagrammen en compositie diagrammen. De analyse toont aan dat object model diagrammen geschikt zijn voor het gebruik als communicatie diagram maar niet geschikt zijn voor het gebruik als compositie diagram. Een aanbeveling wordt gemaakt voor het ontwikkelen van een gereedschap, welke overeenkomt met UML maar uitgebreid is met additionele symbolen om concurrent gedrag uit te drukken. Toestands- en activiteitsdiagrammen en het implementatie mechanisme van Rhapsody om deze diagrammen te maken worden geanalyseerd. Een methode voor het gebruik van kanalen, parallele constructies, en keuze constructies in toestandsdiagrammen is gegeven of aangeraden. Een aanbeveling wordt gegeven voor het gebruik van een ander implementatie mechanisme voor een betere integratie van CTC++ in Toestands- en activiteitsdiagrammen.

Een safeguard voor een mechatronische opstelling, genaamd JIWY, gebasseerd op UML en CT concepten is ontworpen. JIWY is een kleine tabletop robot met twee graden van vrijheid voorzien van een camera. Een gedetailleerde beschrijving van JIWY is gegeven en hiervan zijn de specificaties en voorwaarden afgeleid. De safeguard bestaat uit verscheidene safeguard monitoren, welke de mechatronische opstelling voor specifieke gevaren monitoren. Het gedrag van de safeguard monitoren is gespecificieerd in toestands- en activiteitsdiagrammen. De object model diagrammen laten het communicatie diagram voor het ontwerp zien. Het voordeel is dat het gebruik van kanalen in combinatie met het object model diagram het ontwerp schaalbaar en testbaar maken. Het nadeel is dat object model diagrammen groot van omvang worden, met veel objecten, en moeilijk leesbaar zijn. Aanbevolen wordt om de nog niet geimplementeerde features van JIWY te implementeren.

Table of contents

1 Introduction

Control systems operate in a world that is parallel by nature. Control systems typically consist of multiple sensors and actuators that operate simultaneously. At each sampling the sensor data must be analyzed and responses must be calculated. Control systems have well defined, fixed-time constraints that must be met. In addition to this, user-interface tasks, safety tasks, supervisory tasks and possibly tasks for additional control loops should be performed concurrently. When the computational resources are limited, these tasks should possibly be performed with different priorities, e.g. to make sure that the deadlines will be met. To create efficient and reliable software, the software should reflect this natural concurrency (Welch, Peter H., 1998). Concurrent software can result in a higher throughput, higher responsiveness or reactiveness, increase in fault tolerance and, perhaps the most important factor, reduced complexity of the design.

The independent data streams from the sensors and other inputs should be manipulated such that the proper response can by generated within the defined deadlines or otherwise the system will fail. This tends to complicate the software. In addition, implementing concurrency on a single-processor system introduces difficulties. The system is only able to execute a single instruction at a time. Accordingly, it is not possible to run multiple processes (tasks) at the same time. Only a virtual form of parallelism is possible by switching between parallel processes. This way of implementing parallelism requires the use of a scheduling mechanism. This implies problems of when and how scheduling takes place.

One can program concurrency by using the traditional approach of directly controlling the threads that are related to the various tasks. This approach is however hard and error-prone (Drunen, 2000). Threads are low-level entities that are difficult to control and to understand. Directly controlling threads involves a good understanding of the flow of control of concurrent processes. This extensively increases the complexity and development time for those kinds of programs. In addition, there are problems inherent to concurrency. Examples are starvation, livelock, deadlock, race conditions, non-determinism and shared resources management (Welch, P. and Justo, 1993).

1.1 Unified Modeling Language

The Unified Modeling Language (UML) is a language for expressing the constructs and relationships of complex systems (Douglass, 1998). It is a third-generation modeling language that rigorously defines the semantics of the object-metamodel and provides a notation for capturing and communicating object structure and behavior. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is very expressive and effectively supports a variety of software development processes and business applications.

UML is a graphical language for specifying the analysis and design of object-oriented software (Booch *et al.*, 1999). The UML defines a set of diagrams, also called models, with well-defined graphical symbols so developers can easily understand each other's diagrams. The graphical symbols are augmented by textual descriptions that also aid in model comprehension. Booch states that UML is adequate for real-time systems (Douglass, 1998), which will be tested. Using UML for real-time embedded systems is about applying the UML to meet the specialized concerns of the real-time and embedded domains.

UML defines eight diagram types that document the system software from different viewpoints and at different points in the software process (Booch *et al.*, 1999). Some diagrams are very abstract showing a high-level view of the system without much detail. Others show a lot of detail about of a small part of the system. UML diagrams show a higher-level view of the system that is hard to grasp when looking at just the code. The eight diagrams that are used are:

- *Use Case Diagram*, displays relationships between use cases and actors. Describes context of the system. May be used to formalize and develop behavioral requirements.
- *Class Diagram*, shows classes, packages and their associations, displays relationships such as containment, inheritance, associations and others. Describes the static data used by the system. Shows the separation of the system into packages. Also shows objects and their relationships at a particular time during the execution of a system. Useful for documenting complex data structures and object topology.
- Interaction diagrams:
  - *Sequence Diagram*, shows order of events exchanged by objects during the execution of a scenario. Shows how objects collaborate to implement the behavior of the system. This consists of the vertical dimension (time) and horizontal dimension (different objects).
  - *Collaboration Diagram*, shows the same information as the Sequence Diagram in a different format. Lines labeled with events connect object nodes. Displays an interaction organized around the objects and their links to one another.
  - *Statechart Diagram*, shows the lifecycle of an object as a state machine. Displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions
  - *Activity Diagram*, displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.
- Physical diagrams:
  - *Component Diagram*, displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.
  - *Deployment Diagram*, shows topology of the hardware components and software components executing the system. Software component instances represent run-time manifestations of code units.

The UML is a graphical notation only. You must choose a software development process that defines which diagrams to create and the order in which to create them. Using a software development process enables many UML analysis constructs to be automatically mapped to an implementation language such as C, C++, or Java. Automatically mapping models to an implementation language can increase developer productivity, improve the quality of the system software, and enable a team to react more quickly to changes in requirements. Automatic mapping has the added advantage that the models and code are always in sync because the code is generated from the models. The effort to create models in the analysis phase will continue throughout the entire lifecycle. Two main software development processes are Rational Rose® and I-Logix Rhapsody®. Based on the results of the course Real-Time Software Development the software development process I-Logix Rhapsody® will be used in this thesis.

Appendix A contains a historical overview of the UML.

1.2 CSP and CT

In 1985 Hoare introduced the concept of *CSP* (Communicating Sequential Processes), a notation and theory for describing and analyzing concurrent systems (Hoare, 1985). The CSP theory was more recently brought up to date by Roscoe (Roscoe, 1997) and Schneider (Schneider, 2000). It provides a mathematical notation for describing patterns of communication using algebraic expressions and contains formal proofs for analyzing, verifying and eliminating undesirable conditions, such as race hazards, deadlocks, livelock, and starvation. With CSP the behavior of concurrent software can be described in terms of *processes*, *events* and *composition operators*. Channels are used to let processes communicate with each other. When processes communicate they engage in a communication event. The rules that are derived in CSP can be applied to verify the behavior of the software to guarantee a satisfied behavior. The CSP concept is well thought-out and has been proven to be successful for realizing concurrent software for real-time embedded systems (Grebe, 1993).

CT, *Communicating Threads*, is an object oriented software package currently being developed by Gerald Hilderink (Hilderink, G.H. *et al.*, 2000). CT is an implementation of a subset of CSP principles. CTC++ is the C++ implementation, which will be used in this thesis project.

1.3 Outline of the report

This thesis is divided in six chapters. Chapter 2 describes the concepts and definitions used throughout the thesis and is the reference frame for the thesis. Chapter 3 describes the differences in data transfer, synchronizarion, and concurrent behavior between UML and CSP, specific for Rhapsody and CT. Chapter 4 describes how to use the CTC++ package within Rhapsody. Chapter 5 describes the design of a safeguard for a mechatronic setup based on UML in conjunction with CT. Finally, in Chapter 6, the conclusions and recommendations are given.

2 Concepts and Definitions

This chapter describes the key concepts and definitions of UML and CSP that are applicable in the context of this thesis assignment. After defining the concept a short comparison is made between UML and CSP and some of the main differences are mentioned.

2.1 UML Concepts and Definitions

This section summarizes UML concepts and definitions as found in the OMG UML specification version 1.4 (OMG-UML, 2001).

2.1.1 Objects and Classes

Classes and objects are two fundamental concepts in UML. A class is an abstraction of a set of real-world entities that have the same data and behavior. A class may have many instances called objects. Each object conforms to the rules defined by its class. Attributes define the data encapsulated in the class. When modeling a class, only include the attributes, services, and states required by the application. Classes cooperate with one another to satisfy the requirements of the system. Associations link classes together. Classes communicate with one another by passing events and invoking services.

In UML two types of objects are distinguished:

- An *active* object is one that owns a thread of control and may initiate control activity and runs concurrently with other active objects.
- A *passive* object is one that holds data, but does not initiate control. However, a passive object may send Stimuli in the process of processing a request that it has received. It shares common thread with other passive objects.

2.1.2 Message

A message is a specification of a communication. It defines the roles of the sender and the receiver instances, as well as which association role specifies the communication link. The message is connected to an action, which specifies the statement that, when executed, causes the communication specified by the message to take place. If the action is a call action or a send action, the signal to be sent or the operation to be invoked in the communication is stated by the action. The action also contains the argument expressions that, when executed, will determine the actual arguments being transmitted in the communication. Moreover, any conditions or iterations of the communication are also specified by the action. Apart from send action and call action, the action connected to a message can also be of other kinds, like create action and destroy action. In these cases, the communication will not raise a signal or invoke an operation, but cause a new instance to be created or an already existing instance to be destroyed. In the case of a create action, the receiver specified by the message is the role to be played by the instance, which is created when the action is performed.

2.1.3 Event

An event is a type of message. It is a specification of a type of observable occurrence. The occurrence that generates an event instance is assumed to take place at an instant in time with no duration. Strictly speaking, the term “event” is used to refer to the type and not to an instance of the type. However, on occasion, where the meaning is clear from the context, the term is also used to refer to an event instance.

Event instance

Event instances are generated as a result of some action either within the system or in the environment surrounding the system. An event is then conveyed to one or more targets. The means by which event instances are transported to their destination depend on the type of action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is practically instantaneous and completely reliable while in others it may involve variable transmission delays, loss of events, reordering, or duplication. No specific assumptions are made in this regard. This provides full flexibility for modeling different types of communication facilities. An event is received when it is placed on the event queue of its target. An event is dispatched when it is dequeued from the event queue and delivered to the state machine for processing. At this point, it is referred to as the current event. Finally, it is consumed when event processing is completed. A consumed event is no longer available for processing. No assumptions are made about the time intervals between event reception, event dispatching, and consumption. This leaves open the possibility of different semantic models such as zero-time semantics.

2.1.4 Concurrency

The occurrence of two or more activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads.

Concurrency in respect to operations of passive and active objects.

The `concurrency` of operations specifies the semantics of concurrent calls to the same passive object. Active objects control access to their own operations so this property is usually (although not required in UML) set to sequential. The concurrency can be specified as:

- `Sequential`: Callers must coordinate so that only one call to an instance (on any sequential operation) may be outstanding at once. If simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed.
- `Guarded:` Multiple calls from concurrent threads may occur simultaneously to one instance (on any guarded operation), but only one is allowed to commence. The others are blocked until the performance of the first operation is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks. Guarded Operations must perform correctly (or block themselves) in the case of a simultaneous sequential operation or guarded semantics cannot be claimed.
- `Concurrent`: Multiple calls from concurrent threads may occur simultaneously to one instance (on any concurrent operation). All of them may proceed concurrently with correct semantics. Concurrent operations must perform correctly in the case of a simultaneous sequential or guarded Operation or concurrent semantics cannot be claimed.

2.2 CSP Concepts and Definitions

This section summarizes CSP concepts and definitions (Roscoe, 1997). The main reason that the concepts and definitions defer from the UML concepts is that CSP was designed for describing and analyzing systems whose primary interest are the ways in which different components interact at the level of communication at a higher level of abstraction.

2.2.1 Process

A CSP process is completely described by the way it can communicate with its external environment. The external environment of a process is usually another process.

2.2.2 Event

The event, or communicating event, is the most fundamental object in CSP. Events are assumed to be drawn from a set, which contains all possible communications for processes in the universe under consideration. Communication describes a transaction or synchronization between two or more processes rather than the transmission of data one way. An event can only happen when all its participants are prepared to execute it. The fundamental assumptions about event communications in CSP are:

- Events are instantaneous.
- Events only occur when both the process and its environment allow them.

2.2.3 Concurrency

Concurrency is an abstraction of behavior where the system is viewed as a set of parallel, sequential, and alternative processes that communicate with each other. It ignores the speed of execution due to the absence of any absolute notion of real time.

The notions of real time, priorities, and scheduling are refinements that underlie the abstraction of concurrency.

2.3 Comparison of concepts

In UML there are multiple types of communication, events are one of those types. Events can be both asynchronous and synchronous. In CSP there is one type of communication, the communication event. This event is synchronous.

The UML makes no assumption between the transmission links of object; it allows freedom to fit an applicable communication mechanism. This can lead to loss of events, reordering or duplication. In CSP events there will not be loss of events, reordering or duplication between processes.

In UML concurrency is defined on objects and on their operations. In CSP concurrency refers to the way processes behave. Processes are abstracted from real-live entities and will reflect their natural behavior.

The next chapter will give a detailed description about the differences between UML and CT in terms of data transfer, synchronization and concurrent behavior.

3 Data Transfer, Synchronization and Concurrent behavior

This chapter will elaborate on the differences between data transfer, synchronization and scheduling in UML and CSP, more specifically the difference in scheduling between Rhapsody and CTC++. In order to shed some light in the scheduling schemes both Rhapsody and CTC++ will be analyzed. This chapter first describes the differences between UML and CT and is concluded by a demonstrating example. The goal of this chapter is to demonstrate the necessity to use CSP/CT for real-time control software design in UML/Rhapsody.

CSP provides channels to let processes communicate with each other. Channels provide communication between processes and encapsulate synchronization, concurrent behavior and data transfer. The UML uses events to communicate between objects. Events do not capture synchronization, concurrent behavior and data transfer in the same fashion as a CSP channel does. The differences between data transfer, synchronization and scheduling will be elaborated. Examples created in Rhapsody and CTC++ will be used to show specific cases.

3.1 Data transfer

3.1.1 Data transfer in CSP/CT

Figure 1 depicts two communicating processes. `Process A` can send or receive data to or from `Process B` by use of a channel.

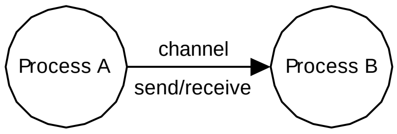


Figure 1 : Communicating Processes

In CT channels perform data transfer on communication. The type of channel specifies the type of data that is transferred. For instance, a channel of integer will transfer integers from one process to another. Figure 2 depicts two communicating processes. `Process A` will write value `x` and `Process B` will read value `x`.

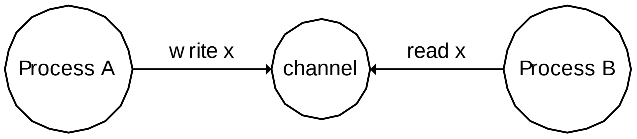


Figure 2 : CT and data transfer

Figure 2 shows that `Process A` sends data to `Process B`. Since it is important to know the direction of communication CTC++ provides two special types of channel interfaces to specify the direction of data transfer. These channel interfaces are `ChannelInput` and `ChannelOutput`. The `ChannelInput` only allows a read on the channel (`read` method). The `ChannelOutput` only allows a write on the channel (`write` method).

The CTC++ implementation of the example in Figure 2 looks like this:

Process A interface:	`Process_A(ChannelOutput_of_Integer *channelout);`
Process A body:	`channelout→write(&x);`
Process B interface:	`Process_B(ChannelInput_of_Integer *channelin);`
Process B body:	`channelin→read(&x);`

3.1.2 Data transfer in UML

In the UML objects need to be associated with other objects in order to enable messaging. Messaging is an abstraction of data and/ or control information passed between objects and doesn't refer to the actual data transfer. In order to send messages between objects an association must exist between the objects. The object model diagram in Figure 3 shows two objects that are connected with an association.

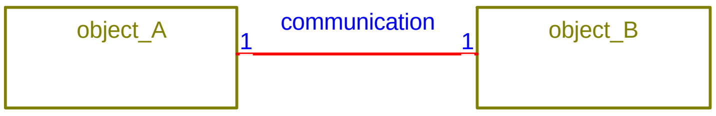


Figure 3 : OMD of two associated objects in UML

An association is a relationship that permits the exchange of messages among objects. (Douglass, 1999) UML associations are by default bi-directional and support messaging in either direction. Associations can be chosen to be uni-directional and in that case an open arrowhead points to the receiving object.

The implementation of an association is a pointer. Thus the association relations in Figure 3 results in:

- A property of `object_A` that contains a pointer to `object_B`.
- A property of `object_B` that contains a pointer to `object_A`.

Thus in other words, objects will have to know each other in order to communicate. This is a fundamental difference when compared to CT. In CT processes don't know each other, a process has an interface consisting out of channels to enable communication to its environment. The way channels connect processes determines the data transfer. The processes don't determine the communication flow themselves as object in UML do.

Now to the actual data transfer between `object_A` and `object_B`. Different messaging implementations are possible, for instance: direct method calls or events. The latter is discussed since it is also a mechanism for synchronization and scheduling. In order to transfer data from `object_A` to `object_B` a message event needs to be sent from `object_A` to `object_B`. The UML event instance is an object itself and will contain the data that is to be transferred. The type of data can be specified by parameterization of the event. A parameter in the context of an UML event is an unbound variable that can be changed, passed, or returned. The sequence diagram in Figure 4 depicts the message event that is send.

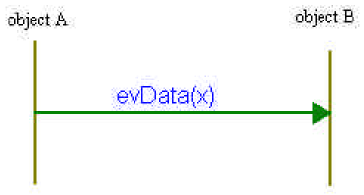


Figure 4 : Sequence diagram UML data transfer

The sequence diagram shows that `object A` sends an event `evData` with parameter `x` to `object B`. Although in this example `object A` is the sender of the event and `object B` is the recipient of the event this does not determine the direction of the data transfer. The direction of data transfer depends on the specification of parameter `x`. If parameter x is defined as type `in` data can be transferred from `object A` to `object B`. If parameter `x` is defined as type `out` date can be transferred from `object B` to `object A`. If parameter `x` is defined as `in/out` data can be transferred in both directions. Rhapsody allows this parameter specification but all types are implemented as type `in`. Since Rhapsody is in development this is probably a bug.

Summarizing, objects need an association in order to allow exchange events. The direction of data transfer depends on the specification of the parameters of the event. Processes don’t need to be associated with other processes. The interface of a process consists of channel inputs or outputs, which determines how a process communicates with its environment.

3.2 Synchronization

3.2.1 Synchronization in CT

In CT, channels encapsulate synchronization. Communication only takes place when both sender and receiver are ready. In the example of Figure 2, `process A` only sends data to `process B` when both processes are ready. This is called rendezvous communication. Channels are basically unbuffered, however, buffers may be added to compensate for communication latencies. (Jovanovic *et al.*, 2002)

3.2.2 Synchronization in UML and Rhapsody

In UML both synchronous as asynchronous communication is possible. Synchronous communication in UML is referred to as invoking an operation. Asynchronous communication in UML is referred to as sending a signal (OMG-UML, 2001). The UML does not specify an implementation mechanism for asynchronous or synchronous communication. The actual implementation mechanism is left to the UML tool, which can vary among different tools.

The UML tool used in this thesis project is Rhapsody. The rest of this section will describe the mechanism Rhapsody uses. In Rhapsody, the mechanism used for synchronous communication is called a *triggered operation*. The mechanism for asynchronous communication is called an *event*. In communication between objects there is a sender and a recipient. The sender will create the triggered operation or event. The recipient will consume the event. A recipient can consume an event if a statechart diagram or an activity diagram describes its behavior. In other words, an object can consume an event if it is reactive. Reactive behavior is described by a statechart –or activity diagram.

If a sending object sends an event to a receiving object, it will continue immediately and not wait for the recipient. The event is placed in the event queue of the receiving object. Events in the queue are handled in the sequence they are received.

If a sending object sends a triggered operation to a receiving object, it will wait until the receiving object has executed the triggered operation. If there are other events pending in the recipient’s queue these will be handled first.

In context of the recipient there is no difference between triggered operations and events. Both are received and consumed in the order they arrive.

Rhapsody discourages the use of both triggered operations and events at the same time. The behavior can be unpredicted and can lead to possible unwarranted results.(UserGuide, 2002)

Summarizing, CT embeds synchronization in its channels where Rhapsody gives the possibility for both synchronous and asynchronous communication. The UML tool used predominantly determines the exact implementation mechanism for synchronous and asynchronous communication.

3.3 Concurrent behavior

3.3.1 Concurrent behavior in CT

In CT channels perform scheduling. All the scheduling is performed on the read -and write actions of channels. The actual scheduling is determined by the manner in which the processes are composed. CT basically offers the following compositional constructers:

- Sequential construct, this construct combines processes into a construction in which one process follows another. The processes are performed sequentially.
- Parallel construct, combines a number of processes, which are performed in parallel.
- Alternative construct, combines a number of processes guarded by inputs. The alternation performs the process associated with a guard that is ready.
- Priority Parallel construct, the same as a Parallel construct but with priority.
- Priority Alternative construct, the same as a Alternative construct but with priority.

The manner in which these compositional constructs operate will be demonstrated with an example.

Sequential composition

The scheduling of the sequential composition will be demonstrated with the following example. There are two processes, `process A` and `process B`. `Process A` will receive a value from its environment. `Process B` will send a value to its environment. Both processes will run sequentially. The semantics of this figure are based on the work of Gerald Hilderink (Hilderink, Gerald H., 2002). Figure 5 depicts the sequential example.

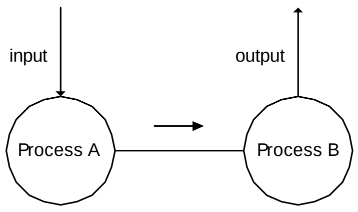


Figure 5 : Sequential Composition

The implementation of this sequential composition has the following form in CTC++.

```
Sequential seq1 = new Sequential();
seq1→add(process_A);
seq1→add(process_B);
```

This composition can be run by calling the `run` method of the sequential process. The behavior is straightforward, `process A` will read from its channel and then sequentially `process B` will write to its channel. If both processes are looped this sequence will repeat itself.

Parallel composition

The scheduling of the parallel composition will be demonstrated with the following example. There are two processes, `Process A` and `Process B`. `Process A` will receive a value from its environment. `Process B` will send a value to its environment. Both processes will run in parallel. The situation is depicted in Figure 6.

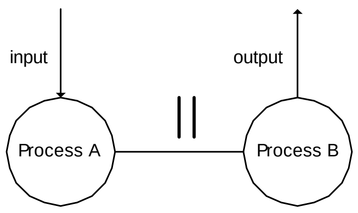


Figure 6 : Parallel Composition

The implementation of this parallel composition has the following form in CTC++.

```
Parallel par1 = new Parallel();
par1→add(process_A);
par1→add(process_B);
```

This composition can be run by calling the `run` method of the parallel constructor. The behavior in this case is in parallel. Either `process A` will read from the channel and then `process B` will write to the channel or vice versa. This interleaving is non-deterministic. Remember that the processes can only read or write if the process they are connected to is ready to write or read. Now consider the case that both processes will continuously perform read –and write operations. By means of this parallel construction both processes will perform one read and write action before the next read or write action.

Alternative composition

The scheduling of the alternative composition will be demonstrated with the same example as is used by the parallel construct. The difference is now that both processes will read or write the moment they can actually read or write. `Process A` will be able to read its value if the channel from which it reads is ready. `Process B` will be able to write its value if the channel from which it writes is ready. If both channels are ready at the same time, only one of the channels is read or written. Figure 7 depicts the alternative composition of two processes.

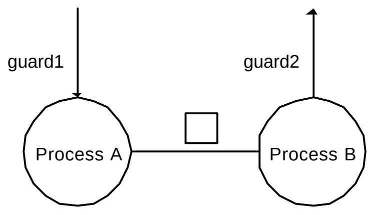


Figure 7 : Alternative Composition

The implementation of this Alternative composition has the following form in CTC++.

```
Guard *guard_A = new Guard(channelinput_A);
Guard *guard_B = new Guard(channelouput_B);
Guard *guardlist = {guard_A, guard_B};
Alternative *alt1 = new Alternative(guardlist, 2);
```

This composition can be run by calling the `run` method of the alternative constructor. The alternative `alt1` can be used as a switch statement. The alternative will return value `0` if `process A` can read from its channel. It will return a value `1` if `process B` can write to its channel. When a guard is ready the associated process will read or write its channel. Thus the scheduling behavior in this case is that both processes will read respectively write the instant their channel becomes ready.

Priority Parallel composition

The piority parallel composition adds priority to the parallel composition. Priority is expressed by an arrow on the compositional symbol. Figure 8 depicts the priority parallel composition of two processes.

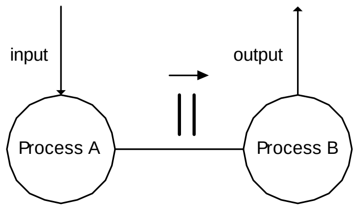


Figure 8 : Priority Parallel Composition

Process A will have a higher priority than Process B. Lower priority processes may only continue when all higher priority processes are unable to.

Priority Alternative composition

The priority alternative composition adds priority to the alternative composition. Figure 9 depicts the priority alternative composition of two processes.

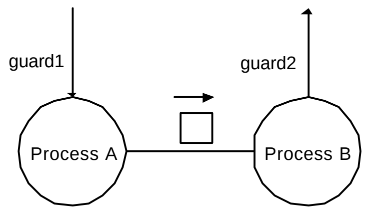


Figure 9 : Priority Alternative Composition

Process A will have a higher priority than Process B. If the guards of both processes become ready at the same time the process with the highest priority will be selected and run.

3.3.2 Concurrent behavior in UML and Rhapsody

The UML doesn't specify actual scheduling. The scheduling mechanism is determined by the UML tool used. If necessary the UML allows the designer to specify one's own specific scheduling mechanism. UML distinguishes two types of objects in terms of concurrency. (For details about the exact definition see chapter 2.)

- Active objects; an object that owns a thread of control and may initiate control activity.
- Passive objects; a passive object is one that holds data, but does not initiate control.

The two types can be used in Rhapsody. Active objects can be used by specifying `active` as the type of concurrency for the object. Passive objects can be used by specifying `sequential` as the type of concurrency of the object. Note that Rhapsody uses a different name for passive objects. The definition the objects are also slightly different when compared to the exact UML definitions.

- Active objects are application objects with active concurrency. An active object runs on the thread of its own task.
- Sequential objects have sequential concurrency, which means that the system runs on a single thread and all operations are executed in sequential order.

Rhapsody provides support for one's own specific scheduling mechanism by offering various concurrency -and synchronization objects. Types that are provided are message queue objects, mutex objects, resource objects, semaphore objects and timer objects. Some of these types are predefined in the Rhapsody framework, the remaining types have to be defined by the designer.

The actual scheduling is thus thread based and depends on the thread library used. The Rhapsody object execution framework (OXF) represents threads by the class `OMOSThread`. This class wraps an operating system thread and adds the behavior, which is necessary for the framework. (For more details see (OXF-Guide, 2002)) This means that the scheduling of Rhapsody depends on the operating system it runs. Thus the same model can behave differently on different platforms.

In the context of real-time controllers, this is unthinkable. The real-time behavior needs to be modeled in a straightforward fashion and not depend on the system it runs. UML does not provide native modeling for this kind of system modeling, by allowing freedom of scheduling to the tool or designer. UML does not provide native semantics for expressing priorities. CT offers clear support for elegant scheduling, synchronization and data transfer at the model level. The scheduling is captured in channels and composition constructs. Furthermore, the scheduling does not depend on the operating system used.

3.4 Simple Processing Example

The goal of this section is to show the differences between Rhapsody and CTC++ applied to an actual example. This section shows the difference in design of the example and the working of the example.

The example models a system that has two inputs and two outputs. The choice for multiple inputs and multiple outputs has been made to collaborate with a controller system which usually has multiple inputs and outputs. The input signals are independent and need to be read and computed. The signals can be read and written instantaneous, there is no blocking. Both signals should be treated as equally important. The computation on the signals is the same. Figure 10 displays the data flow diagram of this example.

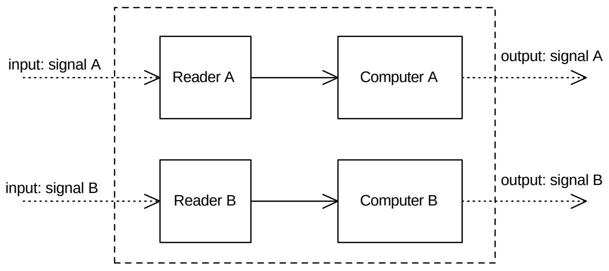


Figure 10 : Data flow diagram of simple processing example

The aim for this system is to be reactive; both signals need to be read and computed in time with equal priority.

3.4.1 Simple processing example in CTC++

The readers and computers are implemented as processes. `Reader A` and `Computer A` communicate with a channel. `Reader B` and `Computer B` communicate with a channel. Both pairs run in parallel. Figure 11 depicts the composition of the processes in this example.

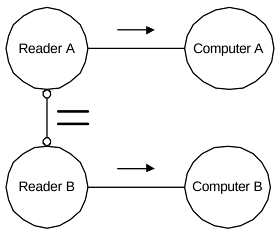


Figure 11 : Composition diagram of processing example

Figure 11 graphically demonstrates the processing example. Currently, there is no tool that can translate such a diagram in code. This is one of the reasons for the use of UML, which allows graphical modeling. The actual composition has to be made manually. The composition can be made as described in the previous sections. The composition is as follows:

```
Par
  Seq
    Reader A
    Computer A
  Seq
    Reader B
    Computer B
```

The bodies of the processes also have to be written manually.

The readers are simple processes that send a value to the computers through a channel. The input signal is omitted since it can be read instantaneous; hence there is no input channel for the reader. The behavior of the reader is specified in the run method.

```
void Reader::run{
  while (TRUE) {
    channeloutput_signal->write(&position);
  }
}
```

The computers are simple processes that read from their input channel and print a message. The behavior of the computer is specified in the run method.

```
void Computer::run{
  while (TRUE) {
    channelinput_signal->write(&position);
    printf("Computed X\n");
  }
}
```

Now that the behavior of the processes is defined and the composition has been made, the composition is run. The output of this example in CTC++ is as follows:

```
Computed A
Computed B
Computed A
Computed B
Computed B
Computed A
Computed A
Computed B
Computed A
Computed B
Computed B
Computed A
```

The output is straightforward and shows that every cycle both signals are read and computed. The output shows that the system treats both signals with equal priority and hence the system is reactive on both inputs. The actual sequence of which signal is read and computed is non-deterministic. The CT++ example will give the same output independent on which operating system it runs. CTC++ treats both signals equally since it implements 'fair' scheduling.

Furthermore, the example can be easily extended in CTC++. Assume that the reading of the signals take some time, either because there is a hardware latency or some signal preprocessing that has to be performed. Assume that the computation also needs some time. Instead of executing the read and computation sequentially they can be executed in parallel for a better reactivity. The reactivity will be better because the computation can be performed while the reader has to wait for the hardware. If the reader has read a signal before the computer is finished it will wait for the computer. If the computer has finished computing it will wait for the reader. This extension is easy in CTC++. The processes do not need to be changed, the only change is in the composition. Figure 12 reflects the changes.

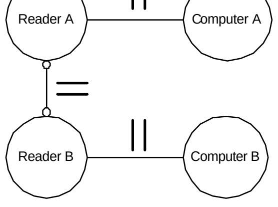


Figure 12 : Extended processing example in a CT diagram

3.4.2 Simple processing example in Rhapsody

The readers and writers are implemented as active objects. In UML relations between objects are depicted in object model diagrams. Figure 13 shows the object model diagram of the processing example.

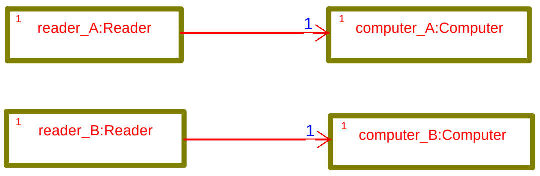


Figure 13 : OMD of processing example

Rhapsody supports automatic code generation. The OMD in Figure 13 will be translated into the appropriate classes, objects and their relations. The behavior of the reader and the computer will be graphically modeled in an activity diagram and statechart diagram. An activity diagram is used to specify the behavior of the reader because it consist out of a single read action. Figure 14 shows the activity diagram of the reader.

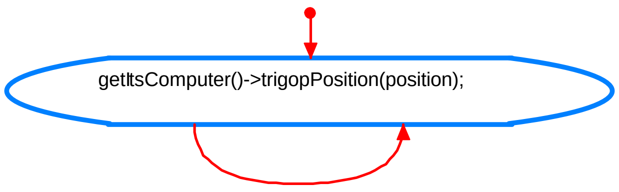


Figure 14 : Activity diagram of Reader

The activity diagram translates to an endless loop of triggered operations. The position will be passed as a parameter.

A statechart diagram is used to specify the behavior of the computer because it ‘waits’ on the signal and will compute the signal on reception. The statechart diagram of the computer is depicted in Figure 15.

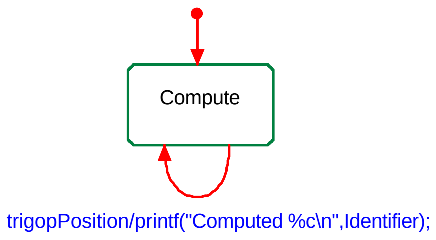


Figure 15 : Statechart diagram of Computer

The statechart diagram shows that on every reception of the triggered operation `trigopPosition()` some output is printed to the screen.

The behavior of the reader and computer is simple in this example and can be easily written by hand. However, in more complex examples, statechart diagrams and activity diagrams will facilitate the design and testing of complex behaviors. Furthermore these diagrams are WYSIWIG and can be understood by looking at them.

Now to the run-time behavior of this example. Will it show equal competition between both readers and computers?

The output of this example on different operating systems.

```
Windows 98   Linux
Computer A   Computer A
Computer B   Computer B
Computer A   Computer A
Computer A   Computer A
Computer A   Computer A
Computer A   Computer A
Computer B   Computer A
Computer B   Computer A
Computer B   Computer A
Computer A   Computer A
Computer A   Computer A
Computer B   Computer B
Computer B   Computer B
Computer B   Computer B
Computer B   Computer B
Computer B   Computer B
Computer A   Computer B
```

The output shows that there is a difference in behavior between both the operating systems. The behavior is different because UML uses threads for its concurrent behavior. Since the operating systems use different threading mechanisms the output differs. As previously mentioned, this is not wanted behavior for controller systems. The same model may not behave fundamentally different just because it runs on another (operating) system.

Furthermore, the responsiveness of the Rhapsody model will be lower than the CTC++ model. A deadlines could be missed because sometimes one signal is read and computed five times before the other signal is read and computed. The overall performance can be higher, since the amount of context switches is limited, but its importance subordinate to the reactivity.

3.5 Outline

The basics about data transfer, synchronization and concurrent behavior of both CSP/CT and UML/Rhapsody have been discussed and compared. This chapter showed that CSP has extensive means to express concurrent behavior at a high level of abstraction. UML is thread based and has ‘only’ *active* –and *passive* objects to express concurrency and no native semantical means to express priorities.

Furthermore, this chapter showed that a system designed and implemented in CTC++ uses ‘fair’ scheduling and its behavior is independent of the operating system on which it runs. A system designed and implemented in Rhapsody is thread based and its actual behavior depends on the threading mechanism used, which is operating system depended.

The next chapter describes how to use CTC++ within Rhapsody.

4 How to use CTC++ within Rhapsody

This chapter describes how to use the CTC++ package within Rhapsody. In more general terms speaking; using the CTC++ package in Rhapsody is using the CSP concepts in conjunction with UML. Wherever possible generalizations will be made and restrictions of Rhapsody or the CTC++ package will be indicated. This chapter starts by specifying a process-oriented view in UML, followed by an approach on how to make communication diagrams and composition diagrams in UML. Then an analysis is made on how to use statechart –and activity diagrams with CTC++. Appendix B.1 contains the approach to make CTC++ available in Rhapsody, this is a prerequisite before using CTC++ within Rhapsody.

4.1 Specifying objects as processes

The CT paradigm reasons about processes, communication between processes, and concurrency between processes. This section describes how to specify processes using objects.

When using CTC++ without Rhapsody, thus writing C++ code by hand, a process can be made by specifying the process interface to the class by inheritance. The process interface enforces the subclass to implement the run method. The example class `Writer` demonstrates this specific inheritance.

```
class Writer : public Process {
        ...............
}
```

When using Rhapsody, the same inheritance needs to be made. There are three slightly different methods to achieve this inheritance:

- Stereotyping.
- Inheritance relation.
- Superclass selection.

All these methods will yield the same result but differ in the easiness to use, a suggestion towards a specific method will be made.

Stereotyping

A class can be stereotyped as a `Process` by adding the `Process` type to the stereotype field in the feature dialog of the class. The result can be viewed in the object model diagram; the stereotype of a class is visible between brackets. Figure 16 depicts this for an example class called `Writer`.

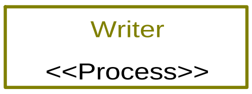


Figure 16 : Object model diagram of class Writer

The base class property of the stereotype `<<Process>>` needs to be set to `public Process` in order to generate the correct code. The advantage of stereotyping is the clear visibility in the object model diagram. It is elegant and WYSIWYG. The disadvantage is that stereotypes cannot be stored in a package for reuse in other models.

Inheritance relation

An inheritance relation can be drawn in the object model diagram. The object model diagram in Figure 17 shows this relation for an example class called `Writer`.

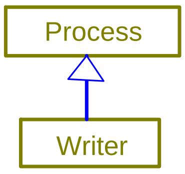


Figure 17 : Object model diagram of inheritance relation

The advantage of this method is its good readability. The disadvantage is that for every process class to be used this relation has to be drawn resulting in either an object model diagram with many classes or many object model diagrams.

Superclass selection

A superclass can be selected by selecting the “add new superclass” method in the browser tree. A drop down box appears and the class `Process` can be selected. Figure 18 shows the result of this approach.

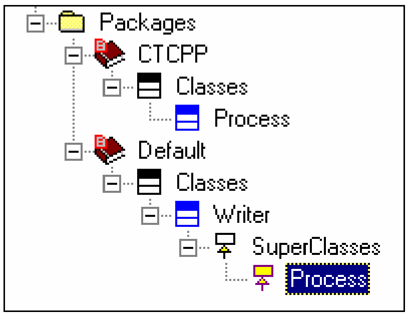


Figure 18 : Superclass in browser tree

The advantage of this method is its easiness to use. The disadvantage is that the superclass is not visible in an object model diagram, such that there is no graphical distinction between an object and process.

The recommended approach for to specify processes is the superclass selection. The approach is easy and reusable.

4.2 CT and the object model diagram

The object model diagram (OMD) displays and sets relationships among objects and classes. Classes an objects are not distinguished in the OMD other than in the respect that an object is an instance of a class. An OMD shows that objects can communicate with each other, classes that inherit from each other, classes that own each other and so on. The common types of relationships that exist between classes in an object model diagram are association, aggregation, composition, generalization, and dependency.

In CT, process are related to each other in the following two ways:

1. Processes communicate with each other. These can be displayed in a communication diagram.
2. Processes relate to each other in terms of execution order; which processes are executed in parallel, in sequence, or by choice. These relations can be displayed in a composition diagram.

This section describes the possibilities of the object model diagram to express these relations. The reason to use a object model diagram is that it provides automatic code generation, which is beneficiary in the following ways:

- It removes the tedious and error-prone work of translating a diagram in code by hand.

- It means that there is a direct mapping between the diagram and the code, between the specification and the implementation.

4.2.1 Object model diagram as communication diagram

The best way to explore if an object diagram is suited as a communication diagram is by constructing a test case. The test case consists of two processes and one channel:

- `Process Producer`, this process will produce integer values to a channel of integer.
- `Process Consumer`, this process will consume integer values from a channel of integer.
- `Channel_of_integer`, this channel is the medium between the Producer and the Consumer.

Figure 19 shows the communication diagram for this example.

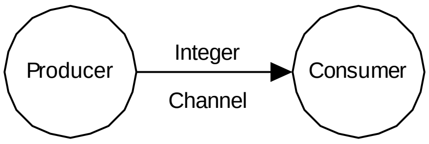


Figure 19 : Communication diagram of Producer Consumer example

The communication diagram is mapped to the object model diagram depicted in Figure 20.

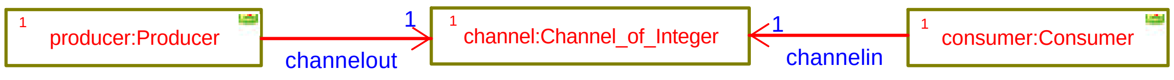


Figure 20 : Object model diagram of Producer-Consumer example

The main difference between the communication diagram and the object model diagram is that the channel is explicitly visible in the latter. This is necessary for automatic code generation due to the following two reasons:

- A channel can be used only if it is instantiated; a channel is an object. The object model diagram of Rhapsody allows it classes to be specified as objects in the instance dialog.
- The relation between the producer, consumer and the channel needs to be specified in order to be set automatically.

In order to translate the object model to a working implementation the producer and consumer need to be instantiated also. The object model diagram shows the names of the instances and the names of the classes separated by a colon.

The associations in the object model diagram of Figure 20 are uni-directional towards the channel. This has to be the case since a process *uses* a channel. The channel does not use a process and hence does not *use* a relation towards that process. Setting an association from a process to a channel results that within the process the channel is known and can be used for a read or write action on the channel. In this specific case the class `Producer` will have a protected property of type `Channel_of_Integer*` named `channelout`. Section 3.1.1 showed that CTC++ normally uses the interface class `ChannelOutput` that only allows write actions. It is not possible to use this interface class in a object model diagram because the result of the automatic code generation cannot provide the right syntax for use of the class. Thus naming heuristics have to be used in the object model diagram. In this case, the name `channelout` is chosen to reflect that the channel should be used as an output channel only.

Another difference between the communication diagram and the object model diagram is that the direction of the data transfer cannot be depicted with an arrow. The direction of the data transfer can be seen by looking at the names of the association.

The object model diagram is translated into code by Rhapsody in the following manner. When instances and relations are specified, Rhapsody generates a function called `initRelations()`. This function is build of two separate functions: `OMCreateInstances()` and `OMConnectRelations()`

The function `OMCreateInstances()` contains the code to create the instances. For this example the code is:

```
static void OMCreateInstances() {
    channel = new Channel_of_Integer;
    consumer = new Consumer;
    par = new Parallel;
    producer = new Producer;
}
```

The function `OMConnectRelations()` contains the code to set the relations. For this example the code is:

```
static void OMConnectRelations() {
    producer->setChannelout(channel);
    consumer->setChannelin(channel);
    producer->setItsParallel(par);
}
```

When writing CTC++ code by hand, the relations between processes and channels are specified in the interface of the process. In the generated code the relations between processes and channels are specified in the properties of the process. However, this will yield the same behavior.

This section showed that object model diagrams of UML are suited to depict communication diagrams. Furthermore, Rhapsody provides direct code mapping which means that drawing the communication diagram in an object model diagram maps directly to the implementation. A useful feature in the object model diagram is multiplicity as will be explained in the next section.

Multiplicity and the communication diagram

In an object model diagram, multiplicity refers to the number of instances of one class that may relate to a single instance of an associated class. Thus multiplicity is an easy mechanism to create many instances from a single class. Figure 21 shows the producer-consumer example in case there are ten producers and ten consumers.

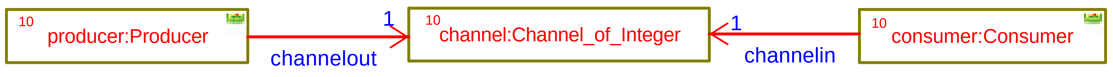


Figure 21 : OMD Producer-Consumer with multiplicity

The diagram shows that the processes and the channel have a multiplicity of ten while the multiplicity of the associations remains one. This might seem strange but consider that each producer has a single relation with each channel and it will make sense.

Rhapsody uses a special container class `OMlist` to keep a list of pointers of the class that will have multiplicity. The definition can be found in the header file of the package.

```
extern OMList<Producer*> producer;
```

The creation of the correct number of instances of the class will be done by a for-loop, containing the right number of iterations. This instantiation can be found in the function `OMCreateInstances()`.

```
for (int i = 0; i < 10; i++)  producer.add(new Producer);
```

The variable `i` is used for the iteration. Knowing the way the instances are created is useful. For instance for uniquely naming each producer and consumer in the example of Figure 21. The class Producer can be extended with an attribute called Identity. This attribute will contain a unique identity for each instance of the class Producer. In order to set this Identity a constructor containing this attribute is added to the class Producer. This actual call of an instantiation can be specified in the instance dialog box of the class Producer provided by Rhapsody. The constructor containing the identity argument can be selected and as actual parameter the value `i` is given. This will result in the creation of ten producers each having a unique identity ranging from 1 to 10. The actual instantiation will look like this:

```
for (int i = 0; i < 10; i++) producer.add(new Producer(i));
```

The setting of the relations will be done in the function `OMConnectRelations()`. The implementation of Rhapsody shows another special class in order to properly setup the relations. This special class is named `OMIterator` and is used to iterate through the `OMList` of pointers of the object. In order to set the relations between two objects with multiplicity, two definitions of `OMIterator` are necessary. The next piece of code shows how the ten instances of the class Consumer are connected to the ten instances of the class of `Channel_of_Integer`.

```
OMIterator<Consumer*> frIter(consumer);frIter.reset();
OMIterator<Channel_of_Integer*> toIter(channel);toIter.reset();
while (*frIter){
  (*frIter)->setChannelin(*toIter);
  toIter++;
  if(!(*toIter))
  {
    toIter.reset();
  }
  frIter++;
}
```

The loop that sets the relations is straightforward, except the `if(!(*toIter))` expression. This expression will test if the `OMList` on the arrow side of the relation has not yet reached it end. If it has reached its end, the Iterator starts again from the beginning of the list. This means that there is no check that the multiplicity on both ends of a relation is the same. This is not a bug but the Rhapsody tool will not check if the multiplicity between two objects matches. In this case; if there would be ten consumers but only 5 channels the relations will be connected like this:

```
Consumer 1,6  → Channel 1
Consumer 2,7  → Channel 2
Consumer 3,8  → Channel 3
Consumer 4,9  → Channel 4
Consumer 5,10 → Channel 5
```

Furthermore there is a bug in Rhapsody, namely that symmetric relations will only be set in one direction. If both directions are appreciated then the designer had to draw two separate uni-directional associations.

4.2.2 Object model diagram as composition diagram

The composition diagram shows how the processes relate to each other in terms of execution order. Figure 5, Figure 6 and Figure 7 in chapter 3 depict respectively the sequential, parallel and alternative compositions. This section explores the possibility to use the object model diagram as a composition diagram.

In order to use the object model diagram as a composition diagram it needs to express the various constructs. The construct witch will be analyzed first is the `parallel` construct.

In the object model diagram five common types of object relationships exist. On a first glance the composition relation sounds appealing. Unfortunately composition in this respect refers to a strong form of aggregation. The UML concept of aggregation applies to objects that physically or conceptual contain other objects. This concept is not useful for the composition, which needs to be made. The association relationship type is the only type that has the potential to be used in a composition diagram. An association between two objects means that two objects are connected and that messaging between the two objects is possible.

The object model diagram in Figure 22 shows a parallel construct with two associations to its supposed member processes.

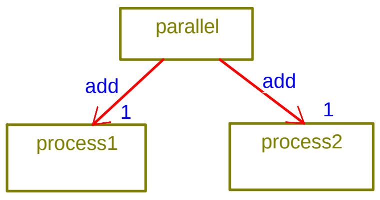


Figure 22 : Possible OMD Composition of Parallel construct

The association is directed from the parallel construct to the processes because processes themselves have no notion about the construct they run under. Graphically this OMD displays a parallel construct with a *use* relation to its member processes. However, this OMD cannot be mapped to the correct implementation. The class of the parallel construct is given (The Parallel construct is found in the CTC++ library) and may not be subject to changes. Furthermore, it is illogical to adapt the construct because a process needs to be added. This means that the parallel composition cannot be successfully made in the object model diagram.

A sequential composition cannot be made either. With a sequential composition it is obligatory to add the processes in a certain order. This order cannot be displayed in an object model diagram.

Unfortunately the object model diagram seems not be suited to draw a composition diagram. This means that the composition has to be made manually. The actual composition of processes entails adding these processes to a CT construct. Afterward the composition needs to be run. The processes and the CT constructs have to be instantiated first before the composition can be made. The code to instantiate the processes will be auto generated if the communication diagram is drawn in an object model diagram. If the CT constructs are added to this object diagram, or drawn in a separate object diagram, these will be automatically instantiated as well. Figure 23 shows the situation for two processes that communicate through a single channel and should be run in parallel.

The "Doc Note" is used to visually group the processes and indicate they belong to the Parallel construct.

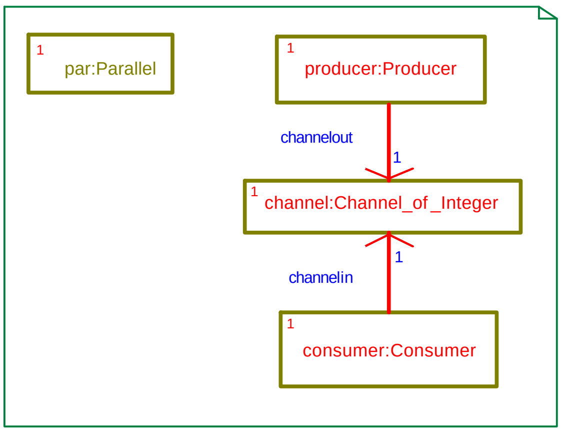


Figure 23 : OMD Composition example

To make an actual composition, a function named `composition()` will be added to the functions of the package. Every composition is a process and therefore has a `run()` function. The `run()` function will be added to the functions of the package. Figure 24 shows the browser view of these functions.

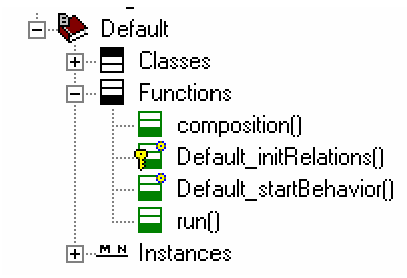


Figure 24 : Browser view of composition() and run() function

The functions are called when the generated application of the package is started if added to the initialization functions. The functions can be added in the feature dialog of the current package under the property `AdditionalInitialization`.

The body of `composition()` function for this example is:

```
void Composition() {
  par->add(producer);
  par->add(consumer);
}
```

The body of `run()` function for this example is:

```
void run() {
  par->run();
}
```

Multiplicity and composition

Figure 21 shows multiplicity of the producer-consumer example in case there are ten producers and ten consumers. This means that the composition will consist out of adding the ten producers and consumers to the parallel construct. This composition will be made manually in the `composition()` function. The instances of the producers and consumers are auto generated, the task that remains is adding the multiple instances to parallel construct. The multiplicity can be made by applying the `OMIterator` class by hand as follows:

```
OMIterator<Consumer*> ConsIter(consumer1);ConsIter.reset();
OMIterator<Producer*> ProdIter(producer1);ProdIter.reset();
while (*ConsIter && *ProdIter){
  par->add(*ConsIter);
  par->add(*ProdIter);
  ConsIter++; ProdIter++;
}
```

4.3 Statechart diagrams, activity diagrams and CT

This section first describes statechart diagrams and activity diagrams in detail. It is pointing out possibilities and restrictions. In the second part CT concepts will be used in collaboration with statechart diagrams and activity diagrams.

4.3.1 From statechart diagram to implementation

This section addresses the scheme that Rhapsody implements to translate statecharts into C++ code. The UML leaves room for design tools to have their own symbols for the elements; Appendix B.7 describes all the elements that can be used in Rhapsody statecharts. The most important elements will be discussed to understand the statechart thoroughly, in order to use the CT concepts.

The implementation of a Rhapsody statechart is basically just a plain function consisting out of sequential code. Within the context of a statechart the type and origin of an event are not relevant, an event merely spans the transition between two states. Thus, the notion of an asynchronous or synchronous event is not relevant within the context of a statechart. The "statechart function" is called by the event dispatcher to take corresponding action upon occurring events. (OXF-Guide, 2002)

The elements and their semantics that are discussed, are: states, events, transitions, guards and actions. After these elements have been described, orthogonal states will be discussed.

The implementation scheme is explained based on a few examples of simple statecharts containing the elements to be explained. Figure 25 shows a simple statechart with two states.

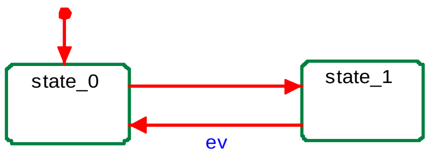


Figure 25 : Simple statechart example

This statechart contains out of the following elements:

- Two empty states: `state_0` and `state_1`. Empty means that the states contain no actions or substates.
- Three transitions of a different type:
  1. A Default transition; the transition to `state_0`.
  2. A Null transition; the transition form `state_0` to `state_1`.
  3. A Triggered transition; the transition triggered with event `ev` from `state_1` to `state_0`

When the `startBehavior` method of the instance of the class containing this statechart is called the statechart will be activated (initialized). The default transition will be taken and `state_0` is entered. The Null transition, a transition without a trigger, will be taken and `state_1` is entered. The transition from `state_1` to `state_0` is taken on event `ev`. Then the Null transition will be taken again and so on.

The implementation of this statechart will look like this:
(Some code is omitted in order to explain the main behavior and not be bothered with small details.)

```
int ActivateStatechart ()
{
  pushNullConfig()
  state=state_0
}
int statechart ()
{
  int res = eventNotConsumed;
  switch (state) {
    case state_0 {
        if (NULL_EVENT) {
          popNullConfig();
          state=state_1;
          res = eventConsumed;
        }
    break;
    }
    case state_1 {
        if (ev) {
           pushNullConfig();
           state=state_0
           res = eventConsumed;
        }
    break;
    }
    default:
      break;
  }
  return res;
}
```

The functions `pushNullconfig` and `popNullconfig` will respectively increment and decrement the counter for 'Null events'. When this counter has a value higher than zero then there are 'Null events'. In this case the event dispatcher will create a 'Null event' if necessary. A 'Null event' will be created when residing in a state without other events and the only way to leave the state is a 'Null transition'. If the state connected to a default transition has a 'Null transition' the function ActivatieStatechart will do a `pushNullConfig`. Furthermore, states with an outgoing 'Null transition' will do a `popNullConfig` and states with an incoming 'Null transition' will do a `pushNullconfig`.

The function `ActivateStatechart` is the implementation of the default transition. This function creates a Null event and sets the statechart to its initial state, in this case to `state_0`.

The function `statechart` represents the implementation of the remainder of the statechart. The code shows that the function is a switch-statement, the switch being the current state. Inside each case statement an if-statement will evaluate the transition that needs to be taken. Since both states have only one outgoing transition both case statements only consist out of one if-statement. The if-statement of `state_0` will evaluate if a 'Null Event' has occurred. The if-statement of `state_1` will evaluate if event `ev1` has occurred. The action upon a `true` evaluation is setting the state to the state the outgoing transitions points to.`state_1`.

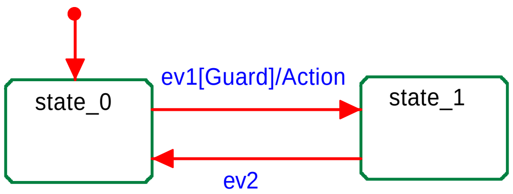


Figure 26 shows the previous statechart appended with a trigger, guard and an action on the transition from `state_0` to `state_1`.

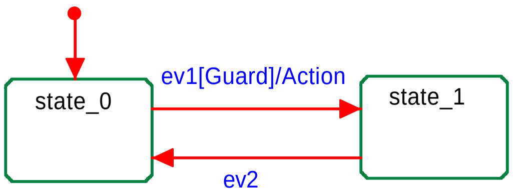


Figure 26 : Statechart diagram with transition, guard and action

The addition of the trigger will make the ‘Null transition’ a triggered transition and there will be no more ‘Null transition’ in the statechart. This will result in the removal of the functions `pushNullConfig` and `popNullConfig`.

By applying a trigger, a guard and an action to the transition from `state_0` to `state_1` the case-statement of `state_0` will look like this:

```
case (state_0)
{
  if (ev1)
    if (Guard)
    {
      Action;
      state=state_1;
      res = eventConsumed;
    }
  break;
}
```

The Guard can be seen in the implementation as another if-statement inside the body of the event if-statement. If the guard will evaluate true the action the action is executed and the state is set to `state_1`. However, if the guard will evaluate false then the event is consumed, no action is executed and `state_0` remains the current state.

Furthermore states can have actions upon entry or upon exit. Figure 27 shows all the mentioned features plus entry and exit actions for both states.

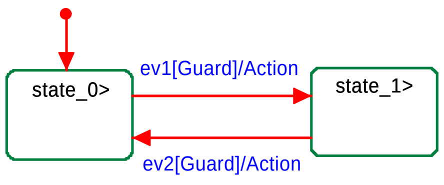


Figure 27 : Statechart diagram example

The implementation of this statechart looks like this:

```
int ActivateStatechart () {
  state=state_0;
  Action on entry state_0;
}

int statechart ()
{
  switch (state)
    case state_0 {
        if (ev1)
          if (Guard)
          {
            Action on exit state_0;
            Action (of the transition);
            state=state_1;
            Action on entry state_1;
            res = eventConsumed;
          }
    break;
    }
    case state_1 {
        if (ev2)
          if (Guard)
          {
            Action on exit state_1;
            Action (transition);
            state=state_0;
            Action on entry state_0;
            res = eventConsumed;
           }
    break;
    }
    default:
      break;
}
```

Inside of the guard clause of the function `statechart()` the sequence of actions can be seen. When a transition is taken the following sequence applies:

- `action on exit` is executed of the current state.
- The action specified on the transition is executed.
- The state is set to the next state.
- `action on entry` of the next state is executed.

Now that the basic workings of the implementation of Rhapsody statecharts are explained, the features to express concurrent behavior within a statechart are analyzed.

4.3.2 Statechart diagrams vs. activity diagrams

This section will describe some of the differences between activity diagrams and state diagrams. The differences described are relevant to the usage of the CT paradigm in state diagrams and activity diagrams.

Activity diagrams are similar in most respects to statecharts. They share a lot of common elements and semantics and the code generation is similar. The main difference is the responsiveness to events. The elements that will be described in this section are states, action states, subactivity states and block states.

States in activity diagrams are just like states in the statechart diagram with one exception: there are no orthogonal states (see appendix B.7 for orthogonal states) in activity diagrams. Furthermore, states as in statecharts are the only constructs in an activity diagram that may have outgoing transitions with triggers. Thus the only way that an activity diagram can respond to events is from states.

Action states describe a sequence of actions to be executed. The actions inside an action state can be considered atomic to rest of the activity diagram (The atomic in this context has no relation to context-switching). All the actions will be executed and only then the action state can be left. The outgoing transitions may only contain guard conditions and actions. Thus an actions state can be considered as a triggerless transition with actions. If the guard conditions are not met on the outgoing transitions the activity diagram will be stuck.

Subactivity states are submachine states that execute nested activity diagrams. When an input transition to the subactivity state is triggered, execution begins with the initial state of the nested activity diagram. The outgoing transitions of a subactivity state are enabled when the final state of the nested activity diagram is reached, or when the trigger events occur on the transitions. Thus subactivity state represents the execution of a non-atomic sequence of steps with some duration.

4.4 CTC++ applied in statechart diagrams

This section describes the use of the CTC++ concepts in statechart diagrams. The concepts are also applicable in activity diagrams; no specific examples for activity diagrams will be given. The section will first describe the use of channels, the Alternative construct and the Parallel construct in statechart diagrams.

4.4.1 Channels in statechart diagrams

This section describes in which way channels can be used in statechart diagrams in the restricted case of one outgoing transition per state. Figure 28 shows two statechart diagrams. The left-hand statechart shows transitions triggered by a Rhapsody event. The right-hand statechart shows transitions triggered by a Rhapsody 'NULL-event' equipped with a channel as action.

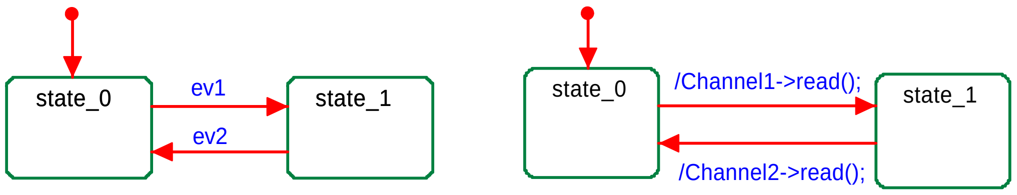


Figure 28 : Statechart UML and Statechart CT

The behavior of both statechart diagrams will be equivalent. The transition from `state_0` to `state_1`, in both diagrams will be taken if a certain event occurs. In the left-hand diagram this transition will be taken if `ev1` occurs. In the right-hand diagram this transition will be taken if event `channel1->read()` occurs. Although specified as an action, it will behave similar to a Rhapsody event in a statechart diagram.

A channel provides communication between processes and encapsulates synchronization, scheduling and data transfer. Communication takes place when both sender and receiver are ready. If one becomes ready before the other, the first will automatically wait for the second to become ready. Thus when a `channel→read()` function is called it will wait until the channel is ready to communicate.

A channel can be used as an action on a transition in a statechart diagram. An action on a transition will be executed if the trigger and guard evaluate true. If there is no trigger or a guard then the action will always be executed. Thus by specifying a channel read function on a transition without a trigger or guard implies that the transition will be taken on the channel event.

A channel cannot be used as trigger on a transition diagram. The previous section showed how triggers are handled in Rhapsody. The statechart function is called with a certain event and reacts to this event. The `channel→read` does not call the statechart function. Furthermore, it cannot be used as a trigger in the statechart function because a CTC++ channel object is not the same as a Rhapsody event object. This sounds a bit cryptic but evaluations like "`if (ev1)`" cannot be applied.

A channel cannot be used as a guard on a transition in a statechart diagram. A guard is implemented as a Boolean evaluation in the statechart function. A channel will not give a result value after a successful read or write and thus cannot be used in a Boolean expression evaluation.

In conclusion; channels have to be specified as actions on transitions in order to use them.

Figure 29 shows a statechart diagram with a transition containing a trigger, a guard and an action.

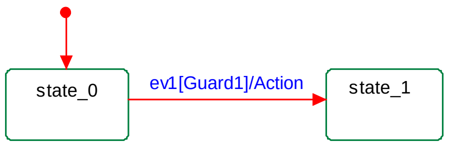


Figure 29 : Statechart diagram with trigger, guard and action

In order to use a channel to trigger a transition, the transition may not contain a trigger or a guard. The behavior of the statechart in Figure 29 can be obtained with a channel as well. The statechart needs to be modified with an additional state, Figure 30 shows the resulting statechart.

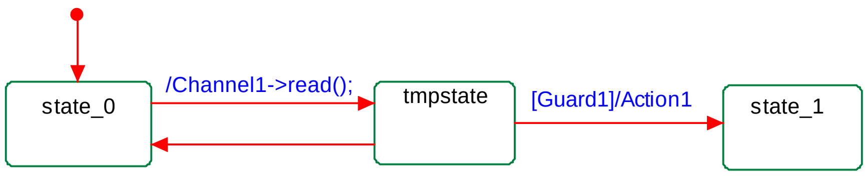


Figure 30 : Statechart with channel, guard and action

The behavior of the statechart diagrams in Figure 29 and Figure 30 will be equivalent. On an incoming event the guard is evaluated.

- If the guard evaluates to true the action is taken in state_1 is entered.
- If the guard evaluates to false. The statechart in Figure 29 will remain in state_0. The statechart in Figure 30 will first take the transition to “tmpstate” and then return to state_0. (The transition back to state_0 is necessary if the guard evaluates to false.)

This section described how to use channels in case of one outgoing transition per state. The section on Alternative constructs describes states with more than one outgoing transition to other states.

4.4.2 Parallel constructs in statechart diagrams

This section will analyze orthogonal states and relate them to a parallel constructs in CTC++. Orthogonal states are two or more independent states that an object can be in at the same time. This is also known as an ‘and’ state. For example, a clock radio can be counting the time and playing music at the same time. This collaborates with a process-oriented view where two independent processes can be composed in a parallel construct. This section first addresses the workings of the orthogonal states in Rhapsody.

Dividing a state by drawing a dotted and-line creates two orthogonal states. Figure 31 depicts these two orthogonal states.

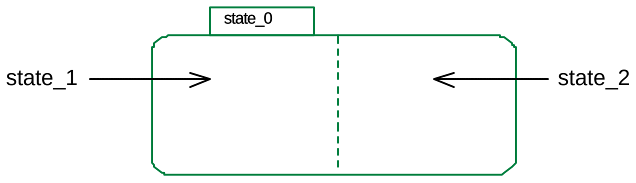


Figure 31 : Orthogonal states

This results not in two separate orthogonal states but in a root state containing two sub states. Thus the net result are three states, a root state containing two orthogonal sub states. The sub states are named, in this case `state_1` and `state_2`, but the names are not visible in the diagram. For each sub state a function describing the behavior of the sub state will be created. In this example the following statechart functions are created, `statechart_0`, `statechart_1` and `statechart_2`.

The orthogonal states are separate statechart diagrams. The statechart functions of the orthogonal states, `state_1` and `state_2`, will be created as described in the previous section. The statechart function of the root state consists out of calling the functions of `state_1` and `state_2` in arbitrary fashion. Figure 32 depicts an example with two orthogonal states both containing the same event to trigger a transition.

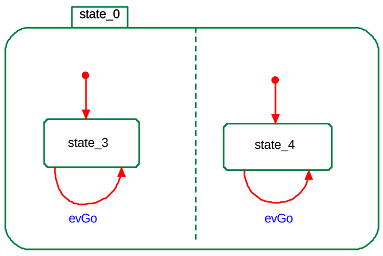


Figure 32 : Statechart diagram example with orthogonal states

What will happen when this statechart receives the event `evGo`? There is only one event `evGo` and no split is visible in the diagram. The result however is that both transitions will be taken. The statechart functions of `state_1` and `state_2` will be called with event `evGo,` which translates in splitting the event. In CT events will never be split implicitly, splitting events is done by using a delta process. Unfortunately the splitting of the event is not WYSIWYG in Rhapsody and allows to create complex and unpredicted constructs as will be explained in the next example.

Figure 33 illustrates the same example but now the event `evGo` will trigger a transition to a state outside the orthogonal states.

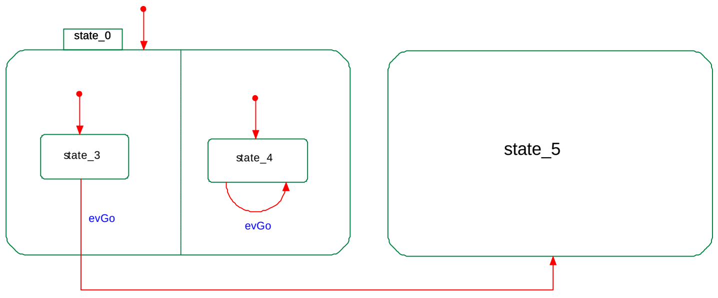


Figure 33 : Statechart example with orthogonal states

What will now happen when the event `evGo` is received, will both orthogonal states, `state_1` and `state_2`, receive the event? As previously mentioned, the function of `state_1` and `state_2` are called sequentially. Since the function of `state_1` is called first the outgoing transitions to `state_5` will be taken. This means that the function of `state_2` cannot be called anymore since `state_5` and `state_2` are not orthogonal. Thus by changing `state_1`, a state orthogonal to `state_2`, the behavior of `state_2` is also changed.

The following will demonstrate another irregularity in the orthogonality. Changing the origin from the outgoing relation from `state_3` to `state_4` will change the behavior of this statechart. In this case `state_1` will receive the event `evGo` `state_2` will receive the event also. Thus both orthogonal states will receive the event. This example demonstrates that special care needs to be taken in account for outgoing transitions in orthogonal states. The example may illustrate bad usage of a statechart diagram since one of the orthogonal states has not got an outgoing transition but UML doesn't prohibit this use. If both orthogonal states would have had an outgoing transition special care needs to be taken to avoid unwarranted behavior of this kind. A join bar can be used to guarantee a WYSIWYG behavior. A join bar joins multiple ingoing transitions into one outgoing transition. This outgoing transition will only be taken if all incoming transitions are taken. Thus if a join bar is used to join all the outgoing transitions from orthogonal states every orthogonal state will have to take its outgoing transition in order to leave the root state containing the orthogonal states. Figure 34 depicts the statechart with a join bar.

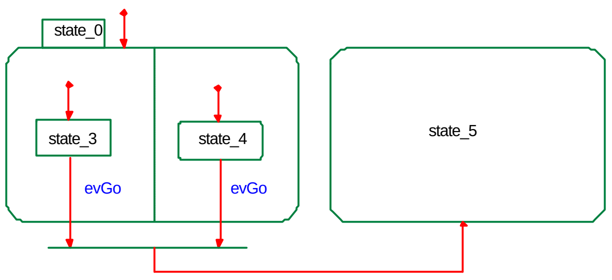


Figure 34 : Statechart with join bar

CT maintains a process-oriented view. In a process oriented view these problems would not have occurred. There can be no implicit splitting of events; the splitting of events always has to be made explicit with a delta process. Two parallel processes can only be completed if both processes will finish their behavior. If the implementation from Rhapsody statechart diagram to code could be altered an parallel processes could be used to reflect orthogonal states. Splitting the events can be made explicit by using a delta process.

4.4.3 Alternative constructs in statechart diagrams

Figure 35 shows two statechart diagrams. The left-hand statechart diagram is without channel; the right-hand statechart diagram is with channels.

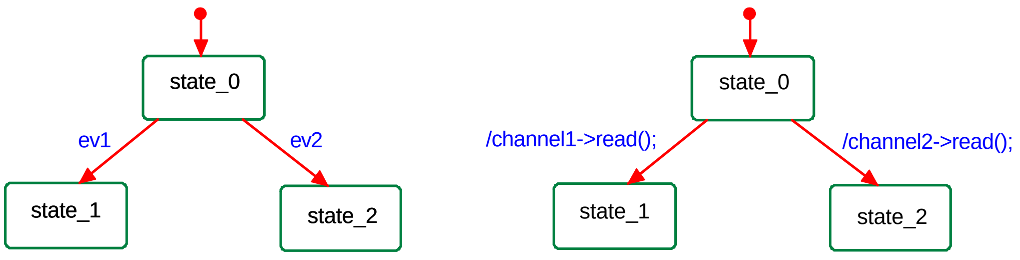


Figure 35 : Statechart UML and Statechart CT

In this case both statechart diagrams will not have the same behavior. The left-hand statechart diagram will take a transition to either state_1 or state_2 depending on which event is received. The right-hand statechart diagram will chose a transition randomly to state_1 or state_2 and will reach the state if the corresponding channel is ready. In order to obtain the same behavior the Alternative construct must be used.

The Alternative construct is a combination of processes guarded by inputs or outputs. The Alternation performs the process associated with a guard, which is ready. The implementation of the CTC++ Alternative is slightly different form the traditional ALT (Hoare, 1988) in OCCAM. The OCCAM ALT will read (or write) the channel if it is used as a guard and then runs the accompanying process. The Alternative in CTC++ uses the `alt→select()` or `alt→run()` function to determine which guard is ready. After the selection the actual read (or write) should be performed.

In order to use an Alternative construct for two channels named channel1 and channel2 the following steps are necessary:

- Creation of a guard for each channel.
```
Guard *guard1 = new Guard(channel1);
Guard *guard2 = new Guard(channel2);
```

- Creation of a guard list containing all the guards.
```
Guard *guardlist[] = {guard1, guard2};
```

- Creation of the Alternative and addition of the guards.
  `alt = new Alternative(guardlist, 2);`

(The alternative construct can also be created first and the guards added afterward with the function `alt→add(*Guard))`

If these steps are taken the alternative construct can be used in the following manner.

```
// Select a ready guard from the channels with a ready status
switch(alt->select())
{
case 0:
   channelinput1->read(&integer);
   /* some action */
   break;
case 1:
   channelinput2->read(&integer);
   /* some action */
   break;
default:
  break;
}
```

It would be great that the right-hand statechart diagram from Figure 35 could be translated automatically into this code. The semantics would allow an automatic generation. Unfortunately the scheme that Rhapsody uses to implement statechart diagrams cannot be altered.

The following approach is a suggestion to use an Alternative construct in a statechart diagram. Although the approach needs some manual actions it is easy to employ and the statechart is still readable. Figure 36 shows the resulting statechart diagram.

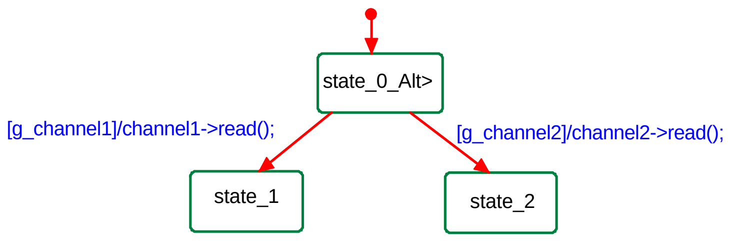

state_0_Alt>
[g_channel1]/channel1->read();
[g_channel2]/channel2->read();
state_1
state_2

Figure 36 : Alternative in statechart diagram

The approach suggests the following naming suggestion for the state diagram.

- The name of the state containing the Alternative construct is appended with an underscore and Alt. ("_Alt")
- The transitions will have a guard with the name of the channel preceded with the letter g and an under score. ("g_")

The guards on the transitions, `g_channel1` and `g_channel2`, have to be attributes of the class containing the statechart diagram. This has to be the case since `g_channel1` and `g_channel2` have to be known in the scope of `state_0` and in the scope of the transition.

The code for the Alternative will be placed in the `action_on_entry` box of `state_0_Alt`, and looks like this:

```
//reset the guards of the outgoing transitions
g_channel1==FALSE;
g_channel2==FALSE;

//creation of the guards
Guard *guard1 = new Guard(channel1);
Guard *guard2 = new Guard(channel2);

//creation of the guard list
```

```
Guard *guardlist[] = {guard1, guard2};

//creation of the Alternative and addition of the guards.
alt = new Alternative(guardlist, 2);

// Select a ready guard from the channels with a ready status
switch(alt->select())
{
case 0:
   g_channel1==TRUE
   break;
case 1:
   g_channel2==TRUE
   break;
default:
  break;
}

//destruction of the alt
delete alt;
```

4.5 Outline

This chapter described how to use CTC++ in conjunction with Rhapsody. It described to use object model diagrams as communication diagrams. It described that object model diagrams are not suited to draw composition diagrams, meaning that compositions have to be made manually. Statechart –and activity diagrams where analyzed and an approach to use channels and constructs in these diagrams was given or suggested. The next chapter is a safeguard case study, which will apply the given approach.

5 Safeguarding with UML and CT

This chapter describes a safeguard design for JIWY (Stephan, 2002), a mechatronic setup, based on both UML and CT concepts. The goal of this chapter is to demonstrate the use of the collaboration between UML and CT in an actual design process. The methods and ideas presented at the design of the safeguard can be used in future designs.

The chapter commences by a basic description and safety requirements of JIWY. The safeguard description and requirements that follow are extracted based on this description. This part is succeeded by the actual design of the safeguard. JIWY is currently still under development, all the features that will be in the final design will be described and the features that are not yet implemented will be indicated.

5.1 System description and safety requirements

JIWY is a little tabletop robot with 2 rotational degrees of freedom mounted with a camera. The setup allows the camera to be rotated horizontally and vertically. In other words the camera can view its surroundings from a fixed point, it can watch up and down, left and right. In the current setup a user can move the camera by using a joystick. This setup will be extended by allowing a remote control through a network connection. Figure 37 shows an overview of the JIWY.

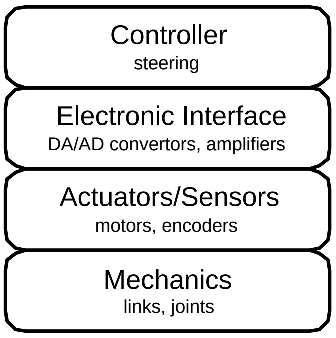


Figure 37 : Setup of JIWY

The safety requirements in this context refer to limits that may not be exceeded in order to prevent unwarranted damage to the system. Four layers and the safety requirements that are depicted in Figure 37 will be described.

5.1.1 Mechanics

The mechanics layer comprises the mechanical construction. The construction contains two joints that allow the camera to rotate on a horizontal axis and a vertical axis.

Safety requirements:
The vertical operating angle is 165 degrees with a full span of 300 degrees, thus full vertical range is from 15 to 315 degrees. The horizontal operating angle is 120 degrees with a full span of 200 degrees, thus full horizontal range is from 20 to 220 degrees. The maximum swing, both vertically and horizontally, are limited by physical end stops.

5.1.2 Actuators/Sensors

This layer includes the actuators and sensors. The actuators will “drive” the mechanics to a desired position. The sensors will measure state of the mechanics.

The actuators:
- Two DC-motors each mounted one a joint.

The sensors:
- Each joint has an incremental decoder in order to measure the actual position.
- Each DC-motor has a current gauge to measure the current supplied to the motor.

- Each joint has two end stop sensors mounted, the end stop sensors will supply a signal when pressed by the mechanics.

The features currently not yet implemented are the current gauge and the end stops sensors. The specification of the actuators and sensors can be found in (Stephan, 2002).

Safety requirements:
- The peak current to the motors may not be exceeded. The average current to the motors has to be bound in order to prevent motor damage.
- Current measured by the current gauge may not exceed a certain level. This level is unknown since there is no current gauge yet. The current gauge will probably the analog input of the electronic interface, in that case the maximum current is the maximum current of the analog input of the electronic interface.
- The impulse that the end stops can endure is limited. (The mechanics may not hit the end stops with a to large force in order to prevent damage.) Since no end stops are implemented this value is unknown.
- The wires between the actuators and sensors and the electronic interface are bundled in one cable. The condition of the cable can be monitored with a watchdog signal. The watchdog signal is a clock signal transmitted for detecting whether the cable is damaged or disconnected. The actuators and sensors of each joint will have a separate cable to the electronic interface and thus a separate watchdog signal. The watchdog signal is not implemented.

5.1.3 Electronic Interface

The electronic interface consists out of hardware and software. The hardware consists out of a PCI I/O card, two amplifiers to drive the DC-motors and a power supply unit. The software consists out of device drivers for the PCI I/O card. The software is CT based and exports a link driver to approach the I/O card.

The specification of the hardware and the specification of the software can be found in (Stephan, 2002).

Safety requirements:
- The inputs and outputs of the PCI I/O controller are limited.
- The input and output of the amplifiers are limited.
- The device drivers should limit the output of the controller to its maximum. This can be achieved since the device drivers are specifically written for the I/O card.

5.1.4 Controller

The purpose of the controller is to position the camera to a certain set point. The controller communicates with channels to obtain the current system state, calculate fitting steering signals and communicate the steering signals back to the system through channels.

5.1.5 Safeguard placement

The safety requirements will be put in a safeguard and not in the controller. This a separation of concerns, the controller can be designed to obtain is set point in the optimal manner, the safeguard will be designed to guarantee the safety in the system. The next section describes the specification and requirements of the safeguard.

5.2 Safeguard specification and requirements

This section describes the specification and requirements of the safeguard for JIWY. The main requirement of the safeguard for JIWY is to protect itself and its environment.

- The controller can damage elements in the underlying layers by giving damaging steering signals. Various scenarios will be mentioned to illustrate this behavior.
- The environment can inflict damage to JIWY in many ways. JIWY cannot be armed to defend itself and cannot prevent direct damage. However, there are scenarios that the environment will put JIWY in a state that damage, or further damage, can be prevented by shutting down quickly.
- The environment needs to be protected for JIWY. This situation is not likely to occur since the mechatronic setup is not really powerful; JIWY is not able to tear off an arm of a human operator but could inflict some scratches. However, it is interesting to include these scenarios because the safeguard design can be used on a bigger, more powerful brother of JIWY.

Two scenarios will be described to illustrate the necessity of a safeguard.

- The movement of the joints can be blocked, for instance by a hand. This can result in an overload of the motors. The motors will have an active voltage and are blocked resulting in a large current through the motors.
- The controller is not aware of the maximum swing. In this case the joints can swing with maximum speed against the physical end stop, which can damage the mechanics, the end stop sensors, the motors and the camera.

Thus the safeguard takes responsibility to prevent or limit damage to the system and/or its environment. The controller can be designed for its main purpose, reaching a certain set point in an optimal way, and not be concerned with safety requirements. To protect the system from the controller the safeguard must be able to interrupt the connection between the controller and the electronic interface. Furthermore the safeguard needs information to make its decisions. Information of the electronic interface is necessary to obtain the state of the system, including steering signals, and output signals of the controller. In order for the safeguard to interrupt and obtain the data, it can be inserted between the controller layer and the electronic interface layer. Figure 38 shows the setup.

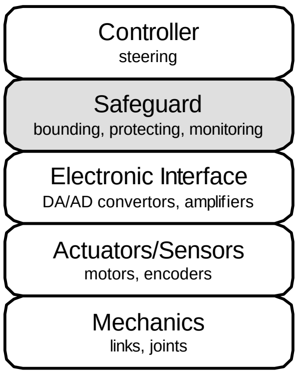


Figure 38 : Setup with safeguard layer

This setup shows that the controller layer is on top of the safeguard layer. In respect of abstraction hierarchy this is correct. Both the controller layer and safeguard layer operate on signals and return signals from the electronic interface and should be drawn next to each other. However the goal of the figure is to depict that the signals of the electronic interface layer will be passed to the controller layer through the safeguard layer.

JIWY is currently under development and various controllers may be designed to control the mechatronic setup. In order to allow design freedom, the controller must not depend on the design of the safeguard. This means that the interface between the safeguard layer and the controller layer has to be same as the original interface between the electronic interface layer and the controller layer. A safeguard provides a fully transparent layer. Therefore, without interdependencies with the controller the safeguard layer can be inserted and removed at will. This is highly recommended because removing the safeguard gives the possibility for controller testing. Controller testing, in this context, means that the behavior of the controller can be tested without intervention of the safeguard. This is only useful in the test phase of the controller and once successfully tested the safeguard should always be inserted. Important is that the current should be limited at the amplifiers to prevent possible damage of the motors.

Thus the following safeguard requirements apply:

- The safeguard layer must be transparent.
- No interdependencies between the safeguard layer and controller layer.

Figure 39 shows the details of the safeguard layer. It also shows the connections to the electronic interface layer and the controller layer.

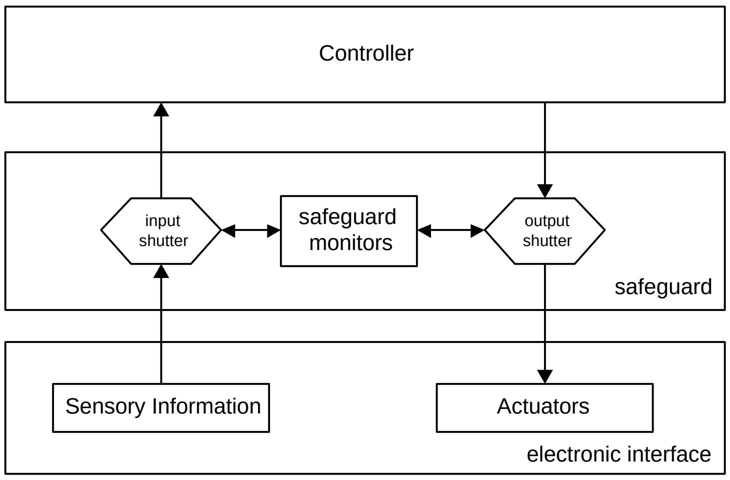


Figure 39: design of safeguard layer

The figure shows that the safeguard layer consists out of three elements. An input shutter, an output shutter and safeguard monitors. The input –and output shutter are bi-directionally connected to the safeguard monitors. The elements will be briefly explained. The complete design will be given in the next section.

The input shutter:

- The first function of the input shutter is to split the sensory information in two. The sensory information goes the controller (unaltered) and to the safeguard monitors.
- The second function of the input shutter is to block the sensory information to the controller if necessary. Blocking the information means that the controller no longer can read its inputs. The controller will be CSP based and its inputs are channels. This means that the controller will stop if there are no more input signals. Blocking can be necessary if one of the safeguard monitors decides the system is in peril.

The output shutter:

- The first function of the output shutter is to pass the steering signals of the controller to the electronic interface.
- The second function of the output shutter is to alter the steering signals to the electronic interface if necessary. The steering signals can be altered in two ways; the steering signals are set to zero or the steering signals are bounded to a certain value specified by the safeguard monitors. Altering can be necessary if one of the safeguard monitors decides the system is in peril.

The safeguard monitors:
The function of the safeguard monitors is to monitor the system for specific scenarios and to react to hazardous situations. The monitors can take two different kinds of emergency measures to guarantee the safety; a *punishing* measure and a *forgiving* measure. Both measures will be described.

- *Punishing measure.* The output shutter will block the steering signal from the controller and send a steering signal of zero to the electronic interface. This means the system will halt in its current state. The input shutter will block the sensory information to the controller. The controller will stop calculating. This measure is irreversible and the whole system has to be restarted manually.
- *Forgiving measure.* The output shutter will overwrite the steering signal from the controller. (It will still read the output of the controller so that the controller will not notice its output is overwritten.) Instead of the controller steering signal the output shutter will give a steering signal supplied by the safeguard monitor that observed the hazardous situation. Once the system may return to a safe state, then the safeguard monitor stops supplying a steering signal and the output shutter will pass on the steering signal from the controller once more.

Whether a punishing measure or a forgiving measure is taken depends on the used safeguard monitors. The specification of the safeguard monitors will show in which scenarios forgiving measures are possible. A punishing measure can always be taken.

A controller may be tested for specific behavior and in that case alteration of the steering signal is not wished, therefore punishing is essential during testing. Protecting the mechatronic system is still necessary so the forgiving measure needs to be easily turned into a punishing measure. The safeguard should have a parameter that can turn all forgiving measures into punishing measures.

Before specifying the safeguard monitors one more consideration needs to be made. JIWY is composed out of a horizontal joint and a vertical joint both equipped with the necessary actuators and sensors. For now the assumption will be made that the horizontal and vertical axes can and will be controlled separately. The electronic interface layer supplies all the signals in two twofold, one signal for the x-axis and one signal for the y-axis. The controller will also supply two sets of steering signals, one set for the motors on the x-axis and one set for the motors on the y-axis. This assumption implicates that two sets of input shutters, output shutters and safeguard monitors work concurrently to ensure the safety of the system. Figure 40 depicts this situation.

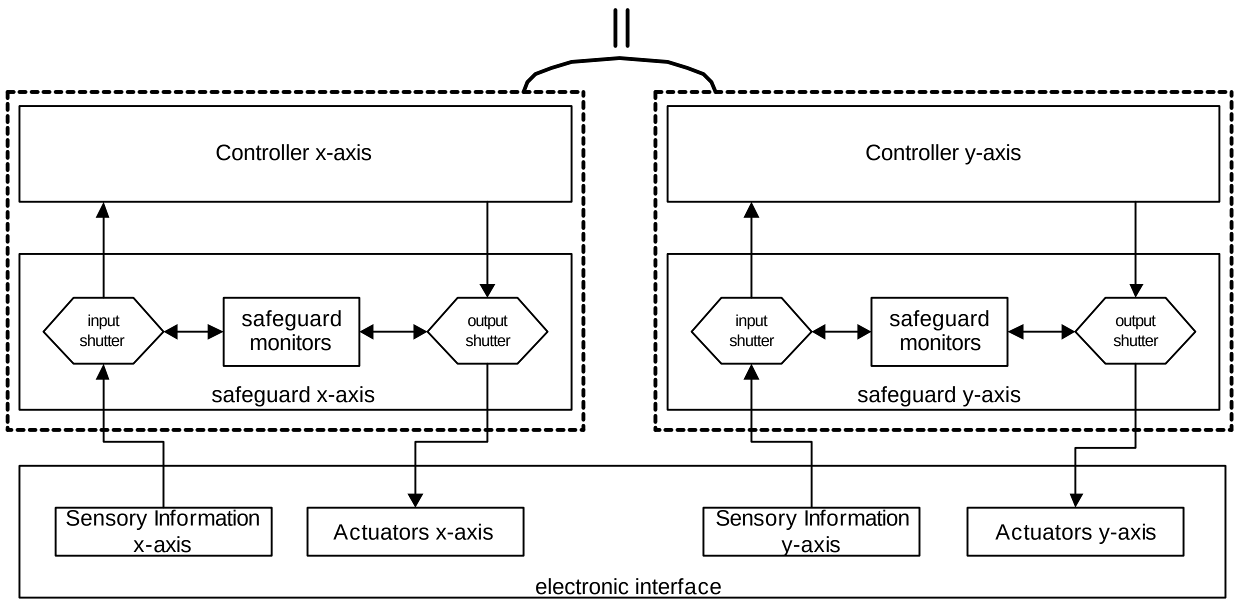


Figure 40 : Safeguard with independent axes

5.2.1 Specification of safeguard monitors

The core of the safeguarding layer consists of safeguarding monitors that run in parallel. Each safeguard will monitor the system for specific situations. The monitors can react based on criteria described in this section. Some of the scenarios in the monitors will have an overlap but that is considered beneficiary since redundancy in safety systems is wishful. Only one issue needs to be considered in the case that multiple monitors reach their criteria; a punishing measure takes precedence over a forgiving measure. This decision is made in the output shutter.

The next list summarizes the different safeguard monitors:

- Current monitor
- Position monitor
- Cable condition monitor
- EndStop monitor
- Speed monitor
- Camera monitor

Current monitor specification

The current needs to be monitored in order to protect the motor from a current overload. Two types of current overload are possible.

- Current peak overload, this is the peak value for the current that may never be exceeded.
- Continuous current overload, the motor has an average current that may not be exceeded. This average current is lower than the peak value of the current.

A current overload can occur when the path of the mechatronic setup is blocked and the motor is jammed while still driven by a voltage. The controller will overload the motor by giving a to high steering value.

The measure that the current monitor takes can be forgiving. The maximum current can be limited to the current peak value. Continuous current overload can be forgiven by giving a lower steering signal. Depending on the controller this can cause a different hazard since the controller may be disordered. This hazard is no danger to the system since one of the other safeguards will take this into account.

Position monitor specification

Basically a controller measures the position of the mechatronical setup, calculates the steering signal based on the current position and a set point and will give a steering signal to the electronic interface to reach the new set point.

If for some reason the new position cannot be reached in a given interval the safeguard needs to stop the controller and thus stop the current supplied to the motors. This is not the same as current monitoring since the current level may not be exceeded. The controller may give a low steering signal in order to reach the new set point at a slow pace. Thus the behavior to be monitored is a non-changing position with a non-zero steering signal. The position may vary a little, possible bumping with a soft object, so a range needs to be determined which is considered to be a non-changing position. Also the interval, after which the controller will be stopped, needs to be determined.

The reaction has to be punishing, the system has to be stopped. The steering signal cannot be blocked temporary since there is no knowledge about the controller. If steering signal would block there is no indication when the block should be removed. A time-out is a possibility but may result in repetitive behavior.

Cable condition monitor specification

On both cables a watchdog signal is present. If this signal is not present something is clearly wrong and the system should be stopped. The reaction has to be punishing since the cable problem cannot fix and the sensory information, if still received, is not trustworthy.

EndStop monitor specification

The EndStop monitoring will consider two scenarios:

1. Continuous hitting of the end stop sensor.
2. Repeatedly bouncing against the end stop sensor.

Continuous hitting of the end stop sensor:
The controller can give a set point, which lies outside the bounds of the mechatronical system. The result is that the end stop is pressed with a non-zero steering signal. This is not good for the motor besides the system will be locked in this state. The reaction has to punishing since the set point of the controller cannot be altered. The measure has to be taken when the end stop sensor is hit longer than a given interval. This scenario could already be detected by the position monitor.

Another situation that can occur is that the end stop is pressed and the position keeps varying. A malfunctioning end stop or an external force pressing the end stop can cause this. The external force can be a person pressing the end stop or some piece of material that accidentally fell against the end stop. Since the exact cause cannot be determined in this case the safeguard will have to respond with a punishing measure.

Repeatedly bouncing against the end stop sensor:
A scenario that also could occur is repeatedly bouncing against the end stop sensor. The reaction to this scenario has to be a punishing measure also since the behavior of the controller cannot be altered. The measure has to be taken when the number of bounces in a given interval exceeds a given value.

Speed monitor specification

The speed monitor needs to monitor the system in order to prevent a high-speed bounce against the end stop. This could occur when the controller is not homed properly and thus not aware of the bounds.

The measure can be forgiving in this case. If the speed is too high when approaching the end stop the steering signal to the end stop should be limited to an acceptable value. Thus the steering signal is altered on approach of the end stop to gradually decrease the speed. This will prevent damage to the mechatronical setup. The speed can be measured by taking the derivative of the position.

The speed monitor needs to be aware of the boundaries thus needs a calibrating scheme in order to determine the boundaries. This calibrating scheme has to be executed on startup of the system. This means that the input shutter has to block the signal to the controller until the speed monitor is properly calibrated (homed).

Camera monitoring specification

There is a camera mounted on the mechatronical system. The visual information could be used to detect harmful situation. However, the visual information of the camera is not regarded in the safeguard design. The camera is not considered since obtaining the necessary safeguard information from the camera is complex and lies not within the scope of this safeguard.

The next table summarizes the measures of the described monitors:

Current monitor	Forgiving
Position monitor	Punishing
Cable condition monitor	Punishing
EndStop monitor	Punishing
Speed monitor	Forgiving
Camera monitor	Not applicable

5.3 Safeguard design and implementation

This section describes the design of the safeguard. It also describes the implementation since the model can be translated in code automatically. First the designs of the parts of the safeguard will be given followed by the design to connect all the separate parts. The parts that will be described are:

- Input shutter
- Output shutter
- Current monitor
- Position monitor
- Cable condition monitor
- Endstop monitor
- Speed monitor

The parts are designed to be completely independent, meaning that the parts are autonomous without shared objects between parts. The reason for designing independent parts is that the parts can be tested independently. Thoroughly testing of simpler, less complex, parts means a higher reliability than testing a more complex design.

Every part is designed as a CSP process and the communication between the various parts, processes, is channel based. Using processes and channels has various benefits which joins the primary advantages of an object oriented approach (Douglass, 1999). The benefits aimed at in the safeguard design are:

- Stability in the presence of changes of specification, requirements and design.
- Support for reliability and safety concerns.
- Scalability.
- Support for concurrency.
- Reusability.

At the end of this chapter the benefits as they apply to the safeguard design will be elaborated.

The design consist out of many parts and interconnections, in order to keep the design readable and unambiguous a set of naming conventions will be used. The naming conventions that apply for the safeguard design can be found in Appendix C.1.

Chapter 4 described how to use the CTC++ package in Rhapsody. The following two implementation notes have to be kept in mind for the safeguard:

- Add the superclass `Process` to each class so that its instantiation is a process.
- Each process needs a `run()` method, if the process has a statechart or activity diagram the run method needs to be:

```
OMReactive::startBehavior();
myThread->start(FALSE);
```

5.3.1 Input shutter design

The input shutter is connected to the electronic interface layer, the controller layer and the safeguard monitors. The next list summarizes the connections of the input shutter:

- Reads the inputs of the sensors from the electronic interface layer.
- Reads a control value from the safeguard monitors.
- Writes the inputs of the sensors to the safeguard monitors.
- Writes the input of the sensors to the controller layer depending on the control value of the safeguard monitors.

Figure 41 shows an object model diagram how the input shutter is connected to the electronic interface layer and the controller layer. (There is more than an object diagram since displaying all the connections in one diagram will make the diagram hard to read.)

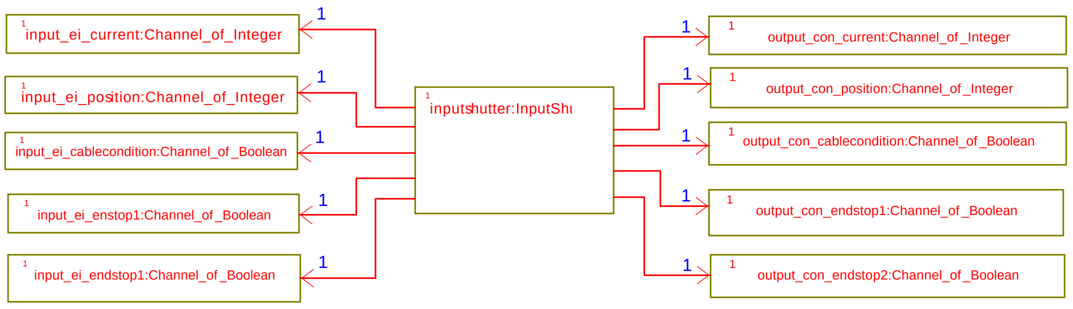

input_ei_current:Channel_of_Integer
input_ei_position:Channel_of_Integer
input_ei_cablecondition:Channel_of_Boolean
input_ei_enstop1:Channel_of_Boolean
input_ei_endstop1:Channel_of_Boolean
inputshutter:InputShu
output_con_current:Channel_of_Integer
output_con_position:Channel_of_Integer
output_con_cablecondition:Channel_of_Boolean
output_con_endstop1:Channel_of_Boolean
output_con_endstop2:Channel_of_Boolean
1

Figure 41 : OMD Input Shutter Environment Connections

The relation names are not visible in order to keep the OMD readable. The relations to the electronic interface are named `chan_input_ei_TYPE`. The relations to the controller are named `chan_output_con_TYPE`.

Figure 42 shows the object model diagram of the input shutter and its connections to the safeguard monitors.

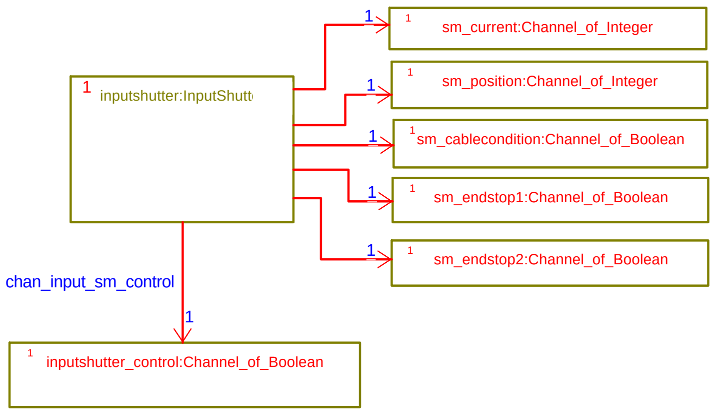

inputshutter:InputShutt
sm_current:Channel_of_Integer
sm_position:Channel_of_Integer
sm_cablecondition:Channel_of_Boolean
sm_endstop1:Channel_of_Boolean
sm_endstop2:Channel_of_Boolean
chan_input_sm_control
inputshutter_control:Channel_of_Boolean
1

Figure 42 : OMD Input Shutter safeguard connections

The relations to the safeguard monitors are named `chan_output_sm_TYPE`.

Now that all the connections are specified, the behavior of the of the input shutter will be specified in activity diagram Figure 43.

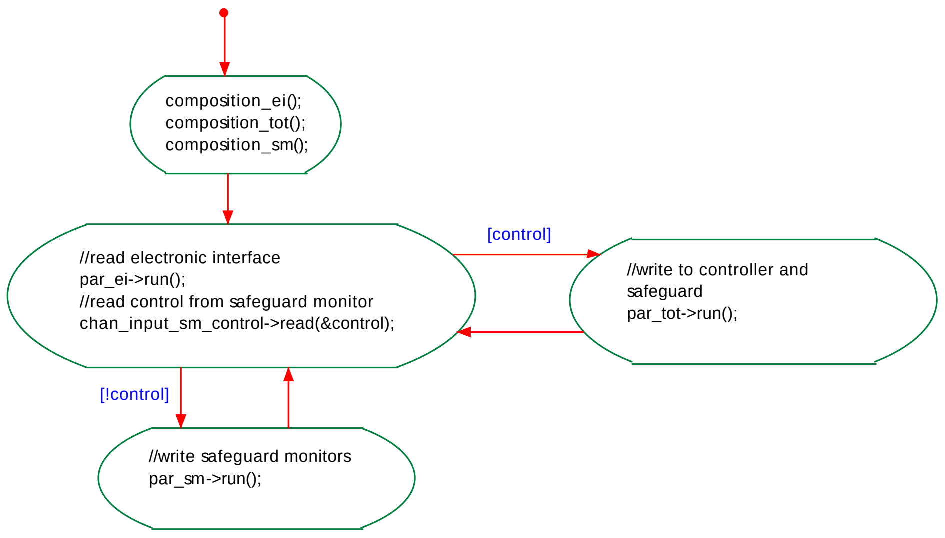

composition_ei();
composition_tot();
composition_sm();
//read electronic interface
par_ei->run();
//read control from safeguard monitor
chan_input_sm_control->read(&control);
[control]
//write to controller and safeguard
par_tot->run();
[!control]
//write safeguard monitors
par_sm->run();

Figure 43 : Activity diagram of the input shutter

The composition functions will create a Parallel construct and adds the channels that will have to be read or written. The composition function of the electronic interface, `composition_ei()`, looks like this:

```
//create the parallel construct
par_ei = new Parallel();
//add the inputs to the parallel process.
par_ei->add(new Reader(chan_input_ei_current,&current));
par_ei->add(new Reader(chan_input_ei_position,&position));
par_ei->add(new Reader(chan_input_ei_cablecondition,&cablecondition));
par_ei->add(new Reader(chan_input_ei_endstop1,&endstop1));
par_ei->add(new Reader(chan_input_ei_endstop2,&endstop2));
```

The function `composition_sm()` adds output channels of the safeguard monitors to a parallel construct.

The function `composition_tot()` adds both the output channels to the safeguard monitors and the output channels to the controller to a parallel construct.

After the creation of these compositions, the next activity state is entered. In this state the run method of the composition for the electronic interface is called; the input channels will be read. Then the control value originating from the safeguard monitors will be read. If the `control` value is true both the safeguard and the controller will receive the sensor signals. If the `control` value is false, a possible safety hazard, only the safeguard will receive the sensor signals.

Note, since the controller and safeguard run in parallel the output to the controller and safeguard will be done in parallel as well. The `parallel_tot->run()` does that.

5.3.2 Output shutter design

The output shutter is connected to the controller layer, the electronic interface layer and the safeguard monitors. The next list summarizes the connections of the output shutter:

- Reads the `steering` value from the controller layer.
- Writes a `steering` value to the electronic interface depending on the control value from the safeguard monitors.
- Reads a control value from the safeguard monitors.
  1. If the value is `0`, the steering signal from the controller is passed on to the electronic interface.
  2. If the value is `1`, the steering signal from the safeguard monitors is passed on the electronic interface. (This is forgiving measure of the safeguard monitors.) In this case the steering signal form the controller will still be read put but in a black hole.
  3. If the value is `2`, the steering signal will be set to zero.

Figure 50 shows the object model diagram of all the connections that the output shutter has.

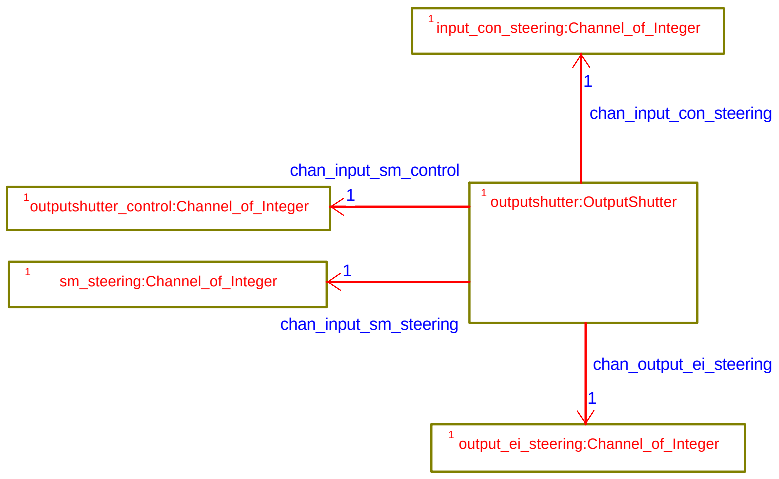

1 input_con_steering:Channel_of_Integer
1
chan_input_con_steering
chan_input_sm_control
1 outputshutter_control:Channel_of_Integer
1
1 outputshutter:OutputShutter
1 sm_steering:Channel_of_Integer
1
chan_input_sm_steering
chan_output_ei_steering
1
1 output_ei_steering:Channel_of_Integer

Figure 44 : OMD Outputs Shutter Connections

Figure 45 shows the activity diagram for the output shutter.

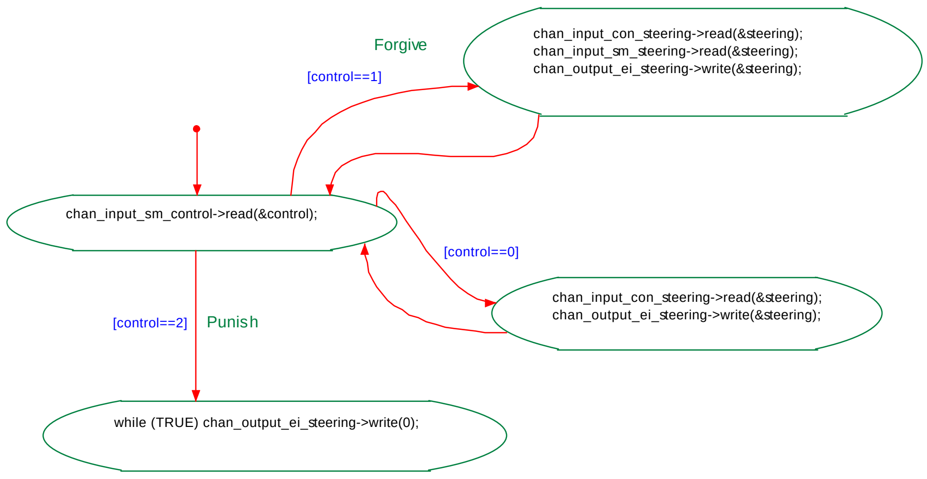

Forgive
[control==1]
chan_input_con_steering->read(&steering);
chan_input_sm_steering->read(&steering);
chan_output_ei_steering->write(&steering);
chan_input_sm_control->read(&control);
[control==0]
chan_input_con_steering->read(&steering);
chan_output_ei_steering->write(&steering);
[control==2]
Punish
while (TRUE) chan_output_ei_steering->write(0);

Figure 45 : Activity diagram of output shutter

5.3.3 Current monitor design

The current monitor will monitor the current supplied to the DC-motor. The current may not exceed the limits specified in the specification. If a limit is exceeded, the current monitor has two options.

- Punish, stop the input to the controller and set the output to the electronic interface to zero.
- Forgive, the steering signal will be bounded to a safe value.

The default choice is forgive. The choice can be altered by calling the constructor of the class with an argument `false`.

The object model diagram in Figure 46 shows the input and output channels of the current monitor.

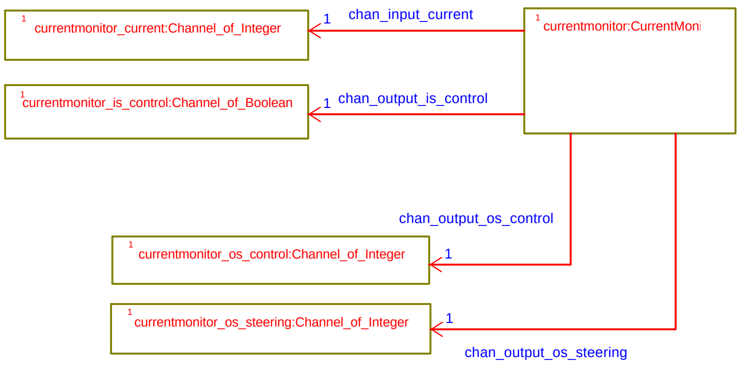

1 currentmonitor_current:Channel_of_Integer
1 chan_input_current
1 currentmonitor:CurrentMoni
1 currentmonitor_is_control:Channel_of_Boolean
1 chan_output_is_control
chan_output_os_control
1 currentmonitor_os_control:Channel_of_Integer
1
1 currentmonitor_os_steering:Channel_of_Integer
1
chan_output_os_steering

Figure 46 : OMD current monitor connections

The behavior of the current monitor is specified in the activity diagram visible in Figure 47.

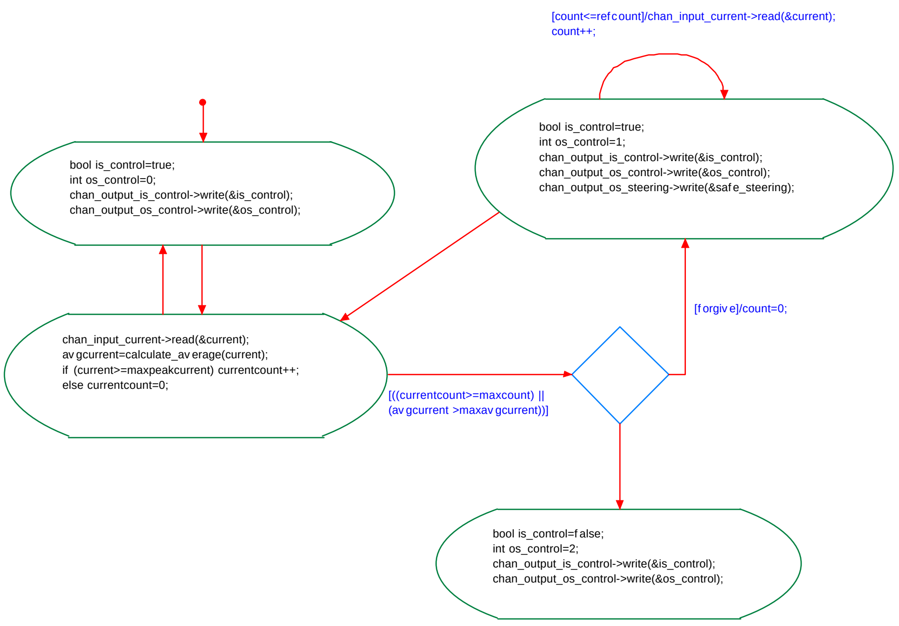

[count<=refcount]/chan_input_current->read(&current);
count++;
bool is_control=true;
int os_control=0;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);
bool is_control=true;
int os_control=1;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);
chan_output_os_steering->write(&safe_steering);
[forgive]/count=0;
chan_input_current->read(&current);
avgcurrent=calculate_average(current);
if (current>=maxpeakcurrent) currentcount++;
else currentcount=0;
[((currentcount>=maxcount) ||
(avgcurrent >maxavgcurrent))]
bool is_control=false;
int os_control=2;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);

Figure 47 : Actvity diagram of current monitor

5.3.4 Position monitor design

The position monitor will monitor the position of the axis. If the pattern of the behavior falls in the scenarios described in the specification, the steering signal to the electronic interface will be set to zero. The output of the sensor signals to the controller layer will be overwritten.

The object model diagram in Figure 48 shows the input and output channels of the position monitor.

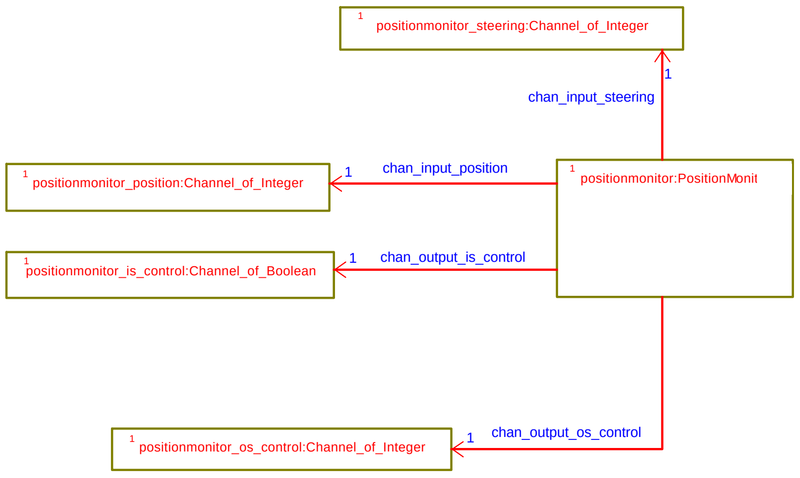

1 positionmonitor_steering:Channel_of_Integer
1
chan_input_steering
1 positionmonitor_position:Channel_of_Integer
1
chan_input_position
1 positionmonitor:PositionMonit
1 positionmonitor_is_control:Channel_of_Boolean
1
chan_output_is_control
1 positionmonitor_os_control:Channel_of_Integer
1
chan_output_os_control

Figure 48 : OMD position monitor connections

The behavior of the position monitor is specified in the activity diagram visible in Figure 49.

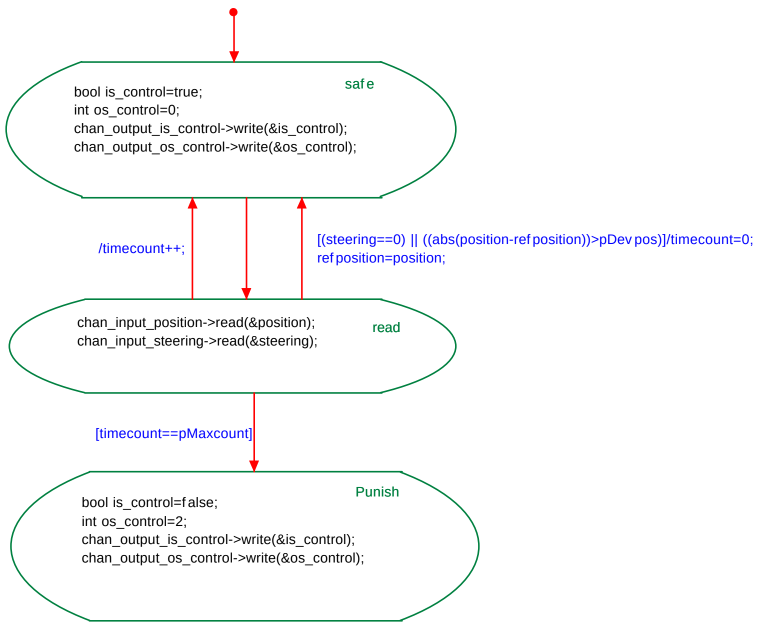

safe
bool is_control=true;
int os_control=0;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);
/timecount++;
[(steering==0) || ((abs(position-refposition))>pDevpos)]/timecount=0;
refposition=position;
chan_input_position->read(&position);
chan_input_steering->read(&steering);
read
[timecount==pMaxcount]
Punish
bool is_control=false;
int os_control=2;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);

Figure 49 : Activity diagram of position monitor

5.3.5 Cable condition monitor design

The cable condition monitor will monitor the condition of the cable. If the condition of the cable is false, it means that there is something wrong with the cable. The cable monitor will then stop the input to the controller and set the steering signal to the electronic interface to zero.

The object model diagram in Figure 50 shows the input and output channels of the cable monitor.

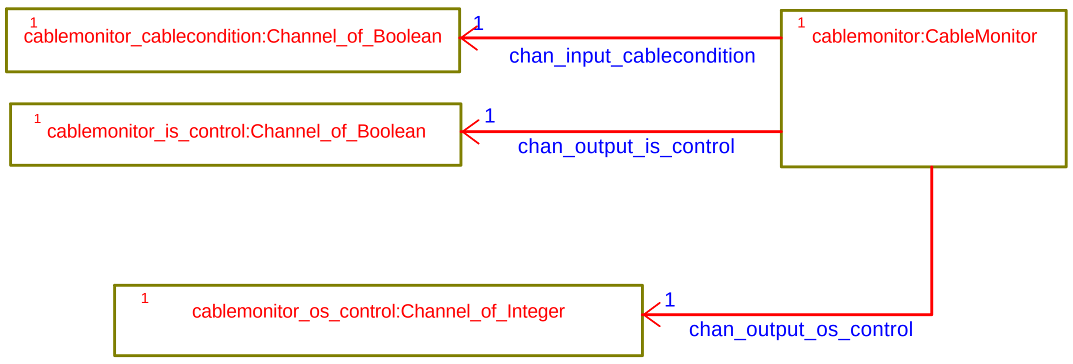

1 cablemonitor_cablecondition:Channel_of_Boolean
1 cablemonitor_is_control:Channel_of_Boolean
1 cablemonitor_os_control:Channel_of_Integer
1 cablemonitor:CableMonitor
1 chan_input_cablecondition
1 chan_output_is_control
1 chan_output_os_control

Figure 50 : OMD cable condition monitor connections

The behavior of the cable monitor is specified in the activity diagram visible in Figure 51.

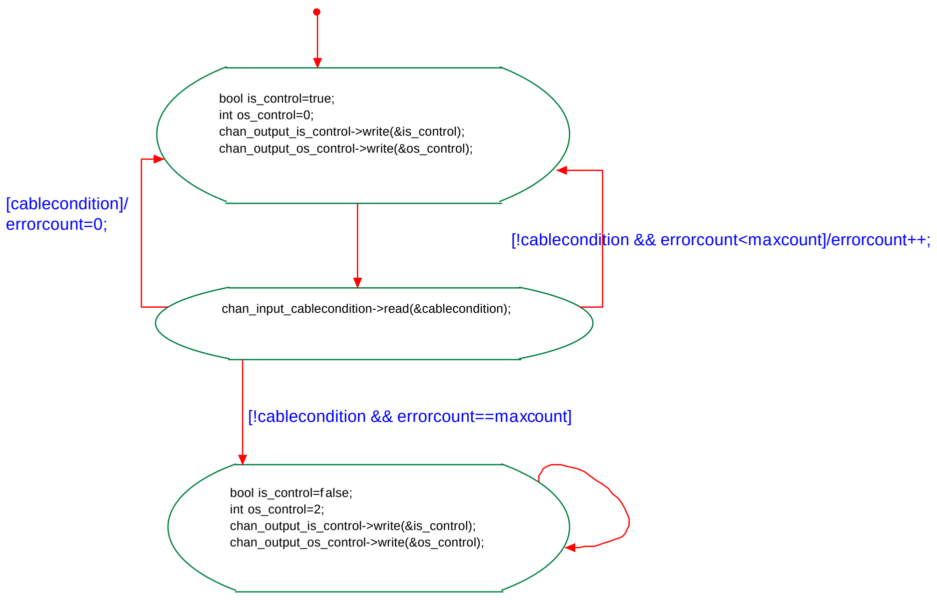

bool is_control=true;
int os_control=0;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);
[cablecondition]/ errorcount=0;
[!cablecondition && errorcount<maxcount]/errorcount++;
chan_input_cablecondition->read(&cablecondition);
[!cablecondition && errorcount==maxcount]
bool is_control=false;
int os_control=2;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);

Figure 51 : Activity diagram of cable condition monitor

The activity diagram shows that there is a punishing measure if the cable condition is false and error count equals maxcount. In this case the parameter maxcount can be used to control the number of samples that the cablecondition signal can be false. Due the noise, or interference of some kind, a single sample could give an incorrect value. It is unwarranted for the safeguard to react in this case, therefore the parameter maxcount that will set the number of subsequent samples that the cablecondition signal may be false. The default value for maxount is three. Setting maxcount to zero means that one wrong sample of the cablecondition can determine the safeguard to react.

5.3.6 EndStop monitor design

The EndStop monitor will monitor the endstops on both side of the axis. If the criteria in the specification are exceeded the resulting action is blocking the sensor signals to the controller and setting the steering signal to the electronic interface to zero.

The object model diagram in Figure 52 shows the input and output channels of the endstop monitor.

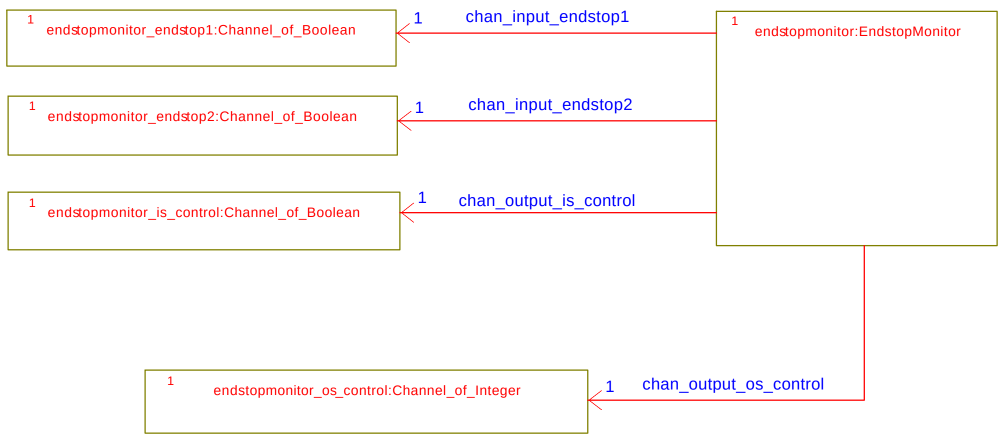

1
endstopmonitor_endstop1:Channel_of_Boolean
1
endstopmonitor_endstop2:Channel_of_Boolean
1
endstopmonitor_is_control:Channel_of_Boolean
1
endstopmonitor_os_control:Channel_of_Integer
1
endstopmonitor:EndstopMonitor
1
chan_input_endstop1
1
chan_input_endstop2
1
chan_output_is_control
1
chan_output_os_control

Figure 52 : OMD endstop monitor connections

The behavior of the cable monitor is specified in the activity diagram visible in Figure 53.

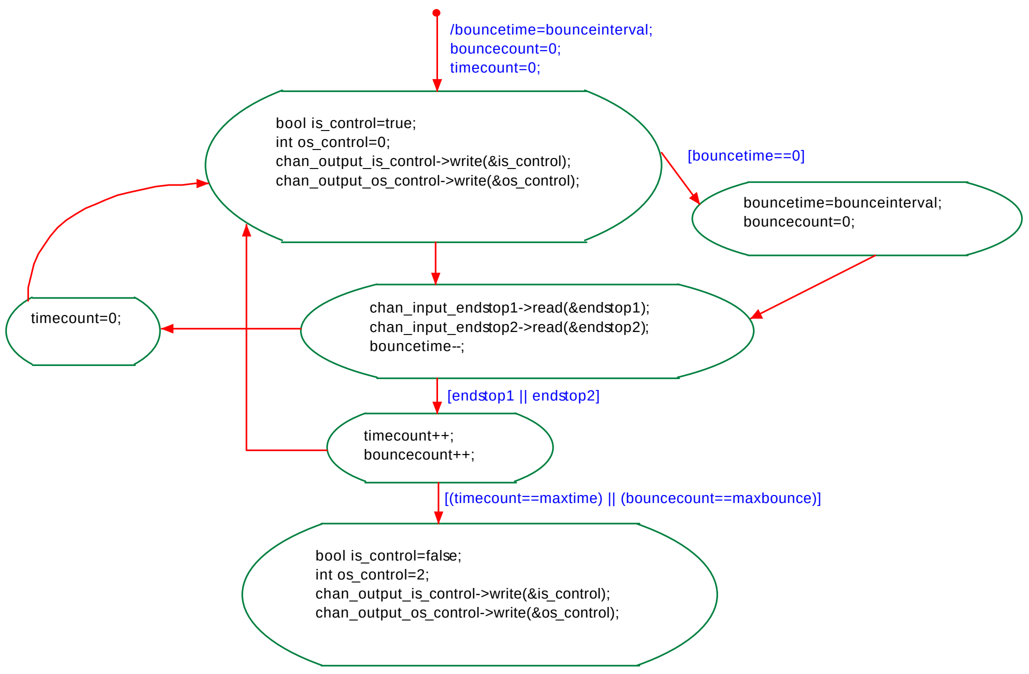

/bouncetime=bounceinterval;
bouncecount=0;
timecount=0;
bool is_control=true;
int os_control=0;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);
[bouncetime==0]
bouncetime=bounceinterval;
bouncecount=0;
timecount=0;
chan_input_endstop1->read(&endstop1);
chan_input_endstop2->read(&endstop2);
bouncetime--;
[endstop1 || endstop2]
timecount++;
bouncecount++;
[(timecount==maxtime) || (bouncecount==maxbounce)]
bool is_control=false;
int os_control=2;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);

Figure 53 : Activity diagram of endstop monitor

The activity diagram shows that there are no separate counters for the endstops. If one of the endstops is pressed the test variables timecount and bouncecount are incremented. Timecount will be set to zero the instant that an endstop is not pressed, thus poses no incorrect behavior. Bouncecount will be set to zero if interval bounceinterval had passed. If bounceinterval is chosen smaller than the time necessary to travel from one endstop to another bouncecount will be zero again before the other endstop will be reached. If the bounceinterval is chosen larger than that time the endstops will not be monitored independently both jointly. Since bouncing between one endstop and the other is not wanted this is not a limitation.

5.3.7 Speed monitor design

The speed monitor needs to monitor the system in order to prevent a high-speed bounce against the end stop. This could occur when the controller is not homed properly and thus not aware of the bounds. The speed monitor needs to be aware of the boundaries thus needs a calibrating scheme in order to determine the boundaries. This calibrating scheme has to be executed on startup of the system. This means that the input shutter has to block the signal to the controller until the speed monitor is properly calibrated (homed). The speed monitor is the only monitor that needs homing, thus the homing algorithm can be placed in safeguard monitor. If more monitors would need homing it can be separated from the speed monitor as a separate process.

The object model diagram in Figure 54 shows the input and output channels of the speed monitor. The endstop signals are necessary to perform the homing. The position signal will be taken as input in order to use the derivative, the speed.

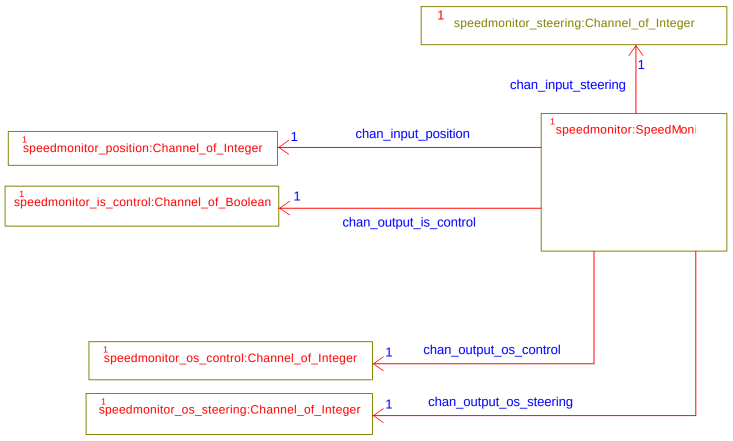

1 speedmonitor_steering:Channel_of_Integer
1
chan_input_steering
1 speedmonitor:SpeedMoni
1 speedmonitor_position:Channel_of_Integer
1
chan_input_position
1 speedmonitor_is_control:Channel_of_Boolean
1
chan_output_is_control
1 speedmonitor_os_control:Channel_of_Integer
1
chan_output_os_control
1 speedmonitor_os_steering:Channel_of_Integer
1
chan_output_os_steering

Figure 54 : OMD speed monitor connections

The behavior of the speed monitor is specified in the activity diagram visible in Figure 55.

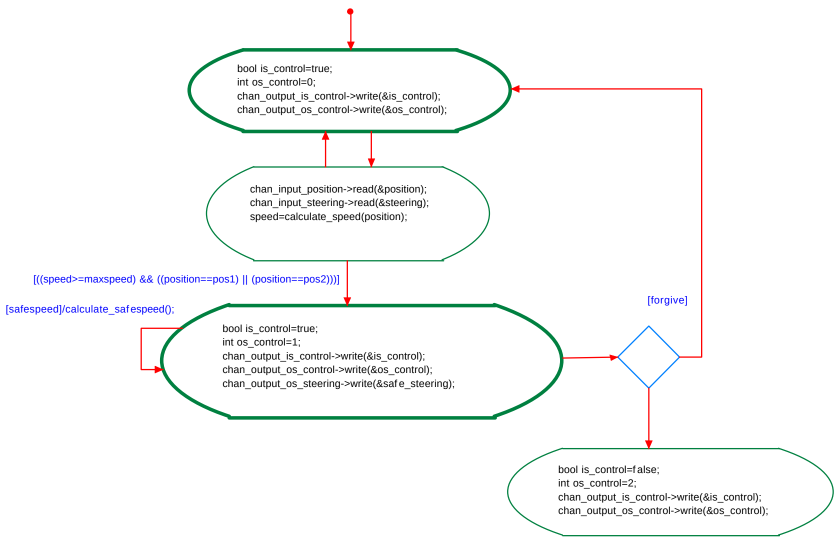

bool is_control=true;
int os_control=0;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);
chan_input_position->read(&position);
chan_input_steering->read(&steering);
speed=calculate_speed(position);
[((speed>=maxspeed) && ((position==pos1) || (position==pos2)))]
[safespeed]/calculate_safespeed();
[forgive]
bool is_control=true;
int os_control=1;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);
chan_output_os_steering->write(&safe_steering);
bool is_control=false;
int os_control=2;
chan_output_is_control->write(&is_control);
chan_output_os_control->write(&os_control);

Figure 55 : Activity diagram of speed monitor

5.4 Outline

This chapter described the specification, design and implementation of a safeguard for a mechatronic setup. The behavior of the various safeguard monitors was specified in statechart –and activity diagrams. The connections between the various safeguard connections are channel based. The object model diagrams depict the communication diagram for the design. The use of channels in combination with the OMD makes the design easy scalable and testable.

This thesis will conclude with the overall conclusions and recommendations in the next chapter.

6 Conclusions and Recommendations

6.1 Conclusions

- UML abstracts away from a specific scheduling mechanism, so therefore the UML tool needs to implement this. The data transfer may not be lossless, which conflicts with controller systems. The UML has no direct support for mutual exclusion between processes for data members. Furthermore, the UML has no semantics at the object level to display concurrency relations between various objects or to express priorities. This makes it necessary to use an additional approach or language to strengthen the UML concurrency model in order to use UML to design real-time applications.
- CTC++ is made available in Rhapsody without conflicting with the features of Rhapsody, for instance animation. The CTC++ elements are stored in a package, which can be easily reused in any model.
- Object model diagrams are suited for drawing communication diagrams, and can be directly and automatically mapped to an implementation. Multiplicity, which allows the instantiation of multiple objects, can be used fully in the context of CTC++ as well.
- Object model diagrams are *not* suited for drawing composition diagrams. The composition has to be made manually by the designer. The designer can use the templates for multiplicity in the manual composition.
- Statechart –and activity diagrams can be used in conjunction with the CT-paradigm to specify the behavior of processes. Communication events can be used to trigger the transitions between states. The Alternative construct can be used to select the appropriate transition in case of multiple outgoing transitions.
- Rhapsody can use the CT-paradigm to develop real-time controller software. Statechart –and activity diagrams facility the design. Animation can be used to test the behavior in an easy and elegant manner. However, since it is not possible to draw composition diagrams in Rhapsody, or in UML for that matter, it is not a complete solution for applying the CT-paradigm in a graphical environment. Furthermore, using channels in object model diagrams create large diagrams with many objects, which are difficult to read.
- The design of the safeguard is elegant and scalable, for instance by adding another axis. Object model diagrams are used to draw communication diagrams that depict the relations of the subparts of the safeguard. Activity –and statechart diagrams express the behavior of the safeguard monitors in a readable and understandable fashion.

6.2 Recommendations

- A minor modification to statechart –and activity diagrams allows channels to be specified as triggers and automatic code generation for alternative compositions. The modified diagrams are still compliant to the UML standard. If a different implementation would be made, orthogonal states can be translated into a composition of parallel processes. The current version of Rhapsody does not allow a different implementation scheme, thus in order to apply a different implementation scheme a different tool should be used or developed.
- Develop a tool, which is basically UML compliant but extended with symbols to express concurrent behavior.
- Use the safeguard on JIWY. In order to do so, the end stops, the current gauge and the watchdog signals have to be implemented.

A History of UML

Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented analysis and design. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, fueling the "method wars." By the mid-1990s, new iterations of these methods began to appear and these methods began to incorporate each other’s techniques, and a few clearly prominent methods emerged.

The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method.

As the primary authors of the Booch, OMT, and OOSE methods, Grady Booch, Jim Rumbaugh, and Ivar Jacobson were motivated to create a unified modeling language for three reasons. First, these methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users. Second, by unifying the semantics and notation, they could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features. Third, they expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. During 1996, the UML authors invited and received feedback from the general community. They incorporated this feedback, but it was clear that additional focused attention was still required.

While Rational was bringing UML together, efforts were being made on achieving the broader goal of an industry standard modeling language. In early 1995, Ivar Jacobson (then Chief Technology Officer of Objectory) and Richard Soley (then Chief Technology Officer of OMG) decided to push harder to achieve standardization in the methods marketplace. In June 1995, an OMG-hosted meeting of all major methodologists (or their representatives) resulted in the first worldwide agreement to seek methodology standards, under the aegis of the OMG process.

During 1996, it became clear that several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included: Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration produced UML 1.0, a modeling language that was well defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response.

In January 1997 IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997.

References (Booch *et al.*, 1999),(OMG-UML, 2001)

B Appendices Chapter 4

B.1 Making CTC++ available in Rhapsody

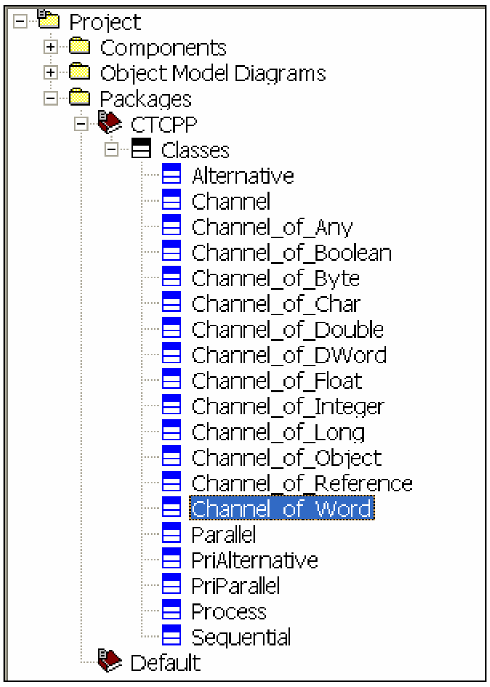
This appendix describes the approach to make CTC++ available in Rhapsody, a main objective of the thesis assignment. The objective is to create a CTC++ package so that CTC++ can be used in conjunction with Rhapsody.

A package is mainly a collection of classes and functions that can be considered and stored as a separate entity. The advantages of a package are that it can be reused in other models and that it can be used and treated as a separate entity. A package is a container that can contain UML diagrams, their classes and functions.

The CTC++ package, and thus all the elements within, may not interfere with the existing functions and features that Rhapsody provides. For instance, the powerful feature of animation should still be applicable to models that use the CTC++ package.

Package with post-processing

In order to use the CTC++ library and all its elements within Rhapsody, CTC++ needs to be available as a model of classes and interfaces. This can be done by creating a package, which contains all the CTC++ constructs. Figure 56 shows the browser view of the package.



- Project
  - Components
  - Object Model Diagrams
  - Packages
    - CTCPP
      - Classes
        - Alternative
        - Channel
        - Channel_of_Any
        - Channel_of_Boolean
        - Channel_of_Byte
        - Channel_of_Char
        - Channel_of_Double
        - Channel_of_DWord
        - Channel_of_Float
        - Channel_of_Integer
        - Channel_of_Long
        - Channel_of_Object
        - Channel_of_Reference
        - Channel_of_Word
        - Parallel
        - PriAlternative
        - PriParallel
        - Process
        - Sequential
    - Default

Figure 56 : Browser view of the CTCPP package

The creation of this package is straightforward. A package named CTC++ is added. The CTC++ elements will be added as empty classes. The empty classes do not contain the behavior of the CTC++ elements but this will be solved later on. In order to fully use CTC++ elements, the interfaces of the constructors, and destructors need to be added. The next example demonstrates this for the `parallel` construct.

The `parallel` construct has the following constructors: (These can be found in Parallel.h of CTC++)

```
Parallel::Parallel(unsigned int stacksize)
Parallel::Parallel(Process* process, unsigned int stacksize)
Parallel::Parallel(Process* processes[], int size, unsigned int stacksize)
```

These can be added to parallel class by selecting the class and choosing the add new constructor method. The result of adding the interfaces of the constructors of the parallel construct is depicted in the Figure 57.

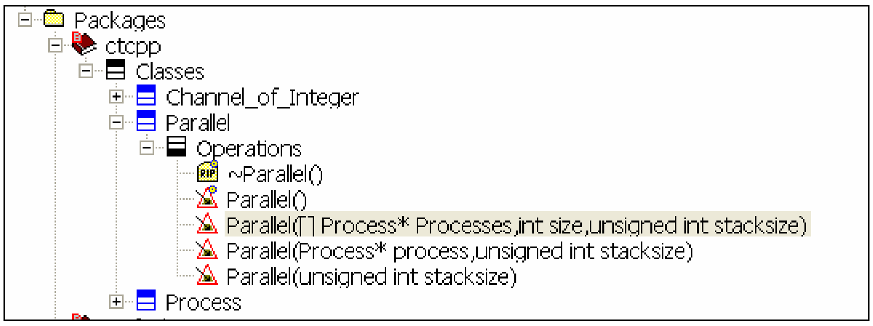

Packages
ctcpp
Classes
Channel_of_Integer
Parallel
Operations
~Parallel()
Parallel()
Parallel([] Process* Processes,int size,unsigned int stacksize)
Parallel(Process* process,unsigned int stacksize)
Parallel(unsigned int stacksize)
Process

Figure 57 : Constructors of class Parallel

Now the `parallel` construct can be instantiated with the different constructors in an object diagram with actual parameters. The same needs to be done for all the other CTC++ elements.

Now CTC++ elements can be used and code can be generated. The generated code will consist of the generated files shown in Table 1.

Project makefile	Projectname.mak
Main project source	MainProjectname.cpp, MainProjectname.h
Ctc++ classes	Parallel.cpp, Parallel.h …..
Package source for the application	e.g. Default.cpp, Default.h
Package classes for the application	e.g. Consumer.cpp, Consumer.h, Producer.cpp …..

Table 1 : Generated files of CTC++ package

In order to succesfully compile the application the following steps need to be taken:

- The generated CTC++ source files need to be replaced by the actual CTC++ source files. This is necessary since the generated CTC++ classes are empty.
- No object files need to be generated for the Rhapsody CTC++ package sources. The CTC++ package depends on the CTC package resulting in a specific compilation process. The CTC++ source files are compiled into a library.
- The CTC++ package library needs to be linked to the Rhapsody application.

The implementation of these steps are described in Appendix B.10.

Use of Animation

Animation is a powerful feature that allows the designer to do animated testing or animated demonstrations. In order to use the animation instrumentation of Rhapsody, the property `CPP_CG→Class→Animate` of the CTC++ package needs to be set to `FALSE`! The animation extends each class with a number of methods. The ‘empty’ CTC++ classes will contain these methods if the mentioned property is not set to `FALSE`. When using the CTC++ classes, dependencies will be made to these methods. Since the generated CTC++ classes will be deleted these dependencies cannot be met anymore resulting in a compile error. Altering the animate property can prevent this. Animation is still possible because the CTC++ classes do not contain parts that can be animated, like statechart diagrams.

The run and startBehavior method.

Rhapsody uses the `startBehavior()` method to start the behavior of the different objects whereas CTC++ uses the `run()` method to start the behavior of a processes. This section describes how to use the methods in collaboration so that the behavior of the processes is correctly and automatically started. The implementation details can be found in Appendix B.9.

Every process contains a `run()` method that contains the implementation of the behavior of the process. This method is called when the processes will be run. Processes will be run when the construct to which they have been added to will be run. The following example demonstrates a parallel construct with two member processes. The run method of these processes will be called when the run method of the parallel construct is called.

```
par
  process A
  process B
```

Process A and Process B will run in parallel and each will keep running until the run method is finished. Thus processes keep running until their `run()` method is finished.

The function `startbehavior()` contains the runtime behavior of the statechart of an object. The `run()` method of a process must call the `startbehavior()` method to start the statechart. The problem is that the `startBehavior()` method will exit immediately after starting the run-time behavior. This means that the `run()` method exits immediately and the process will be descheduled which means that the behavior of the process is stopped prematurely. This means that the `startBehavior()` method needs to be altered not to exit immediately but only when reaching the termination state.

The current implementation of `startBehavior()` consists out of calling the `OXF::startBehavior()` method and then calling the start method, which starts the main event loop of the object. For more information see the Rhapsody Object Execution Framework guide. (OXF-Guide, 2002) The `start` method results in the stated problem, after starting the main event loop it will exit immediately. The start method can be called with an additional parameter `doFork`. The parameter `doFork` is of type Boolean and determines whether the event loop will start and give back control or will start and keep control. When the `start()` method is called with a `doFork` value false, the main event loop will be started and the function will not give back control. The use of the `doFork` value is intended for the application itself. By default the application will not terminate. By changing the value of `doFork` the application can terminate. The reasoning for not terminating the program even after finishing its behavior is that Rhapsody is intended to generate embedded applications that should run forever.

The run method of a process using a statechart will be:

```
ProcessA::run () {
  OMReactive::startBehavior();
  myThread->start(FALSE);
}
```

This doesn't solve the issue completely, by calling the main event loop with `doFork=False` the main event loop will never terminate. This means that the mean event loop needs some additional code in order to exit when the statechart reaches its termination connector. The main event loop can be found in method `execute()` in the Rhapsody framework component `OMThread`. The adaptation of this function can be found in Appendix B.4.

OUTLINE

This appendix described an approach to use CTC++ in Rhapsody by using a package with post processing. The package contains all the CTC++ elements, the post processing consists out of a script, which takes care of the necessary compiling and linking. The package can be saved as a separate entity and can be reused in new projects. The scripts are independent of projects and reusable.

Appendix B.2 describes an approach to transfer the existing CTC++ library into an UML model. This is not the chosen approach since there are difficulties as appendix B.2 describes.

The code generated with CTC++ and UML should be targetable to Linux since the safeguard design will be run on Linux. Appendix B.5 describes the necessary setup to make the Rhapsody framework and the CTC++ library available for Linux. Furthermore, this Appendix describes a method to generate, to build and to run applications directly from Rhapsody to a Linux machine.

Appendix B.3 contains changes that have been made to CTC++ in order to use CTC++ in Rhapsody. All the implications made in this appendix refer to Rhapsody in C++ 4.0.1. There are some dependencies to the current framework of Rhapsody meaning that CTC++ might not be used in the exact same way in a newer version of Rhapsody.

B.2 Importing CTC++ by Reverse Engineering

This appendix describes the approach and the problems of reverse engineering CTC++ into a Rhapsody model. Reverse Engineering allows importing legacy C and C++ code into a Rhapsody model. The Rhapsody model can be saved as a Rhapsody package. Reverse Engineering extracts design information from the code constructs found in the source files and builds a model, insofar as possible. Having CTC++ available as a package is useful in the following respects:

- CTC++ can be used in Rhapsody to use the CSP concepts in conjunction with UML.
- The designer can look into the CTC++ model for a better understanding of CTC++.
- CTC++ can be further developed with the design tools and facilities Rhapsody provides.

Unfortunately reverse engineering is not the exact opposite of generating code from a model. Design information may get lost during the reverse engineering process if Rhapsody cannot represent it internally. Design information can also be approximated if Rhapsody has not got an exact internal representation. Code generated from a reversed engineered model with lost or approximated information can lead to unwarranted behavior. Lost information can cause the model not to be compiled at all and approximated information can cause a different behavior than originally intended. Thus for successful reverse engineering of CTC++, there may be no lost or approximated information. Although CTC++ is written in C++ and depends on CTC, which is written in C, this does not guarantee for successful reverse engineering.

Table 2 shows some of the most important constructs that are lost on import:

Lost C++ Construct:	**Description:**
Anonymous types with members	Enum, class
Qualifiers	Const, volatile
Storage classes	Auto, register, static, extern, mutable
Function specifies	Inline, explicit

Table 2 : Constructs that are lost on import

Unfortunately CTC++ has an abundant number of these constructs. Thus reverse engineering CTC++ cannot be done automatically without losing information. Apart from the previously mentioned constructs another source of information will be lost. The CTC++ library can be build with different compilers and for different targets. This flexibility can be found in the source code by many ‘defines’. The reverse engineering tool of Rhapsody has two possibilities to treat defines, either the define is completely ignored or the define is completely imported in the model. If the define should be ignored or imported, can be specified in the tool. The result however, is that after reverse engineering all the defines disappear. Thus the reversed engineered model cannot be build with different compilers or for different targets. This means that CTC++ cannot be automatically reversed engineered and Rhapsody cannot be used for the development of the CTC++ package.

B.3 Changes made to CTC++

This appendix describes the changes made to CTC++.

A naming conflict needed to be resolved. Both the Rhapsody framework and CTC++ used the name instance. The naming conflict is resolved by renaming every occurrence in CTC++ of instance to INSTANCE. This renaming can easily be done with the following script.

```
#!/bin/bash
#Script for renaming every occurence of INSTANCE into INSTANCE
#execute in csp directory, works recursively

update()
{
  if [ -f $filename ]
  then
    echo update file : $filename
    cat $filename | sed -e 's/INSTANCE/INSTANCE/g' > $filename.tmp
    mv -v $filename.tmp $filename
  fi
}

for filename in `find .`
do
  update
done
```

The files in the csp/util/buildingblocks where not included in the compile script. These were included so that the library also consists out of the buildingblocks specified in that directory.

CTC++ can be compiled with different compilers. For instance, DGPP and GCC. CTC++ is compiled in a library. In order to used the CTC++ elements the library will be linked together with the application. Due to a linker incompatibility between DJGPP and GCCC the defual constructors of all the used classes have to be changed. The default contructers need to be appended with “void” between the braces. The next example illustrated it for the channel of integer.

In the header file Channel_of_Integer.h

```
Channel_of_Integer();  → Channel_of_Integer(void);
```

In the cpp file Channel_of_Integer.cpp

```
Channel_of_Integer::Channel_of_Integer() → Channel_of_Integer::Channel_of_Integer(void)
```

This has to be done for all the classes.

B.4 Changes to Rhapsody Framework component OMThread

This appendix shows the changes needed to `OMThread.cpp` in order to change the behavior `startbehavior()` method.

The function that needs to be changed is the main thread event scheduler:

```
OMReactive * OMThread::execute()
```

The section:

```
if (!dest->shouldDelete()) dest->setShouldTerminate(TRUE);
else delete dest;
```

Needs to be changed into:

```
if (!dest->shouldDelete()) dest->setShouldTerminate(TRUE);
else {
  delete dest;
  toTerminate = TRUE;
  break;
}
```

After applying this change the framework needs to be recompiled with the following command:
In the LangCpp type:

```
/usr/share/rhapsody/etc/linuxmake linuxbuild.mak
```

B.5 Setup to use Linux as deployment system for Rhapsody

This document will describe the necessary steps to setup a Linux machine as deployment system for Rhapsody 4.0.1.

- The development system is a system with a Windows NT operating system.
- The deployment system is a system with a Linux OS operating system.

On both systems, development system and deployment system, changes need to be made. First the requirements and setup of the deployment system will be elaborated on. Second the requirements and setup of the development system will be elaborated on.

The goal is to have an easy access to the Linux deployment system. After completion of this setup the Linux deployment system can be used in the following manner:

- Configurations tab:
  Set the directory to the mapped network drive.
  Select Linux as environment.

Now generating, building and running is a press on the button.

Deployment system, Linux os

Requirements:

- On most Linux system the GNU GCC compiler is installed, if not install this compiler.
- A valid login and a directory with read-write-executable access. The source files need a place on the machine to be compiled and to be run. Default every user on a Linux system has a home directory with read-write-executable access so no specials steps are needed to create that environment.
- SSH daemon (sshd), this daemon allows remote logins through the ssh protocol. This is necessary so that the development system can call the make scripts on the deployment system and run the binary.
- (A network connection ;-)

Optional requirements:

- File sharing daemon (smbd), common on most Linux systems and usually the home directories are shared by default. This creates an easy way to copy the source files from the development system to the deployment system. If this daemon is not present the files can be copied using the ssh protocol.

Setup:

- Rhapsody ships the necessary library –and framework files for Linux. The files can be found in the LinuxShare.tar archive located in the installroot directory of Rhapsody on the development system. (If this archive cannot be found somebody forgot to install the Linux sources during the install of Rhapsody.) Copy the LinuxShare.tar archive to the deployment system and extract the archive to a logical location. This can be in the home directory of the user or the system wide directory like '/usr/share/Rhapsody' in case of multiple users.
- Adaptation of bashrc file (configuration file for the bash). This adaptation is necessary to run the binary automatically after the run script (see development topic) is called. The run script creates an executable file in the users' home directory.
  The adaptation results in the following behavior upon login:
  - if there is an executable run file, copy to temporary file, remove executable flag, execute temporary file.
  - no executable run file, do nothing.

  Next the adaptation of the bashrc file is listed:

  ```
  if [ -x ~/run ]; then
    cp ~/run ~/.run.tmp
    chmod -x run
    ~/.run.tmp
  fi
  ```

Development system, windows os

Requirements:

- Plink, this is a free command line ssh client, which allows to make a connection to a remote host and execute commands on the remote host. This client can be downloaded from : http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Optional requirements:

- Map a drive locally originated from the remote host. This allows Rhapsody to save the generated code directly on the remote host. In case this is not possible the 'linuxmake' script should be extended in order to copy the source files through the ssh protocol.

Setup:

- Configuration of Rhapsody.
  Rhapsody needs the locations of a make-script and an executable-script. These locations can be set in the project properties menu or in the `siteC++.prp` file. Setting these locations in the project properties menu means that the settings are only available in the current project. Setting these locations in the `siteC++.prp` file, located in the `installroot/Share/properties` directory, means that the settings will be available for all projects.
  The properties containing the locations are the `InvokeExecutable` and `InvokeMake` properties found in the `Project Properties menu -> CPP_CG -> Linux`. The next listing shows a working configuration for a `siteC++.prp` file: (The `$OMROOT\ETC` directory contains more scripts and is a logical choice for these additional scripts.)

```
Subject CPP_CG
   Metaclass Linux
    Property InvokeMake String "$OMROOT\\etc\\linuxmake.bat $makefile
 $maketarget"
    Property InvokeExecutable String "$OMROOT\\etc\\linuxexec.bat $executable"
    end
 end
```

- Compile script in order to compile the source on the deployment system. Run script in order to run the created application on the deployment system. The listings of these scripts can be found on the next pages. These scripts need the following information in order to function properly:

  Information about the localhost:
  path to plink.
  the drive letter of the mapped drive.

  Information about the remote host:
  hostname or ip-address
  login information: username and password.
  path to the linuxmake script

File listing of linuxmake.bat

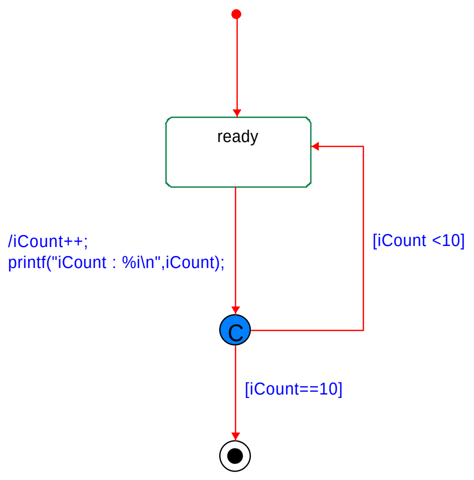
```
@echo off

REM ************************************************************
REM This batch is created by P.M.Visser on 2 april 2002
REM Place this file in  the Share\etc directory of rhapsody under name
linuxmake.bat
REM This batch will only work on a NT prompt.
REM This batch will invoke the linuxmake script on the remotehost for the current
project
REM the stdout and stderr will readable in the output window of Rhapsody
REM
REM
REM
REM *** REQUIREMENTS ***
REM * Localhost *
REM plink.exe, opensource program to execute commands on a remote host
REM adjust in the project properties -> CPP_CG -> LINUX -> INVOKEMAKE to
Rhapsdir\etc\share\linuxmake.bat $plus whatwastherealready
REM * Remotehost*
REM The rhapsody library (linuxShare.tar) extracted on the remote host
REM sshd on the remote host
REM *************************************************************

setlocal

REM *** change settings if necessary ***
set username=username
set password=password
set remotehost=rtat092
set linuxmake_path="/usr/share/rhapsody/etc/"
set plink_path="c:\progra~1\rhapsody\share\etc"
set mapped_drive=z

REM add plink_path to path
set path=%path%;%plink_path%

echo linuxmake %1 %2
REM create compilecommand
plink -pw %password% %username%@%remotehost% "echo cd '%~p1' | sed -e 's/\\/~\//'
|sed -e 's/\\/\//g' >compile && echo '%linuxmake_path%linuxmake %1 %2'>>compile"
REM execute compilecommand
plink -pw %password% %username%@%remotehost% "chmod +x compile && ./compile
1>stdout.txt 2>stderr.txt"

echo ** stdout **
type %mapped_drive%:\stdout.txt
echo ** stderr **
type %mapped_drive%:\stderr.txt

endlocal
```

File listing of linuxexec.bat

```
@echo off

REM ************************************************************
REM This batch is created by P.M.Visser on 2 april 2002
REM Place this file in  the Share\etc directory of rhapsody under name
linuxexec.bat
REM This batch will only work on a NT prompt.
REM This batch will invoke the linuxmake script on the remotehost for the current
project
REM !!the stdout and stderr will readable in a the plink console
REM
REM
REM
REM *** REQUIREMENTS ***
REM * Localhost *
REM plink.exe, opensource program to execute commands on a remote host
REM adjust in the project properties -> CPP_CG -> LINUX -> INVOKEEXECUTABLE to
Rhapsdir\etc\share\linuxexec.bat $executable
REM * Remotehost*
REM The rhapsody library (linuxShare.tar) extracted on the remote host
REM sshd on the remote host
REM *************************************************************

setlocal

REM *** change settings if necessary ***
set username=username
set password=password
set remotehost=rtat092
set plink_path="c:\progra~1\rhapsody\share\etc"

REM add plink_path to path
set path=%path%;%plink_path%

echo linuxexec %1
REM create runcommand
plink -pw %password% %username%@%remotehost% "echo cd '%~p1' | sed -e 's/\\/~\//'
|sed -e 's/\\/\//g' >run && echo './%1'>>run && chmod +x run"
REM execute runcommand (solved by .bashrc file on remotehost)
plink -pw %password% %username%@%remotehost%
pause

endlocal
```

B.6 Ambiguity with condition connector

Figure 58 shows a statechart that demonstrates a simple loop with use of a condition connector. Table 3 shows the output of the implementation of the statechart. The statechart makes believe that the loop is iterated 10 times while the output shows clearly that the loop is iterated 11 times. On a first glance it appears to be a behavioral error but on a closer inspection on the semantic of the condition connector the behavior is actually correct. The semantic of the condition connector dictates that the guards of the outgoing transitions need to be evaluated before the incoming transition to the condition connector is taken. The choices for this semantic will not be elaborated upon apart from the fact that certain cases need this semantic for a correct behavior.



```
iCount : 1
iCount : 2
iCount : 3
iCount : 4
iCount : 5
iCount : 6
iCount : 7
iCount : 8
iCount : 9
iCount : 10
iCount : 11
```

Figure 58 : Simple loop in statechart

Table 3 : Output Condition Connector Ex.

Now the question arises how to use a condition connector without indiscrepancy between the view of the statechart and the implementation. Figure 59 shows a different view of the statechart found in Figure 58. This shows that actions found on the incoming transition can be translated into actions on the outgoing transitions. Keeping this in mind Figure 60 can be created, which shows a statechart with the intended behavour that again is WYSIWIG.

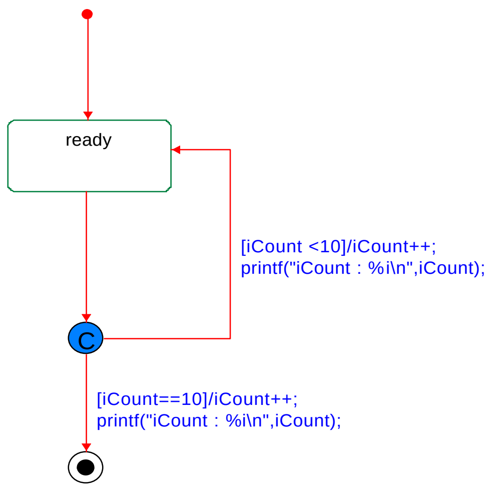

ready
[iCount <10]/iCount++;
printf("iCount : %i\n",iCount);
C
[iCount==10]/iCount++;
printf("iCount : %i\n",iCount);

Figure 59 : Different view of statechart

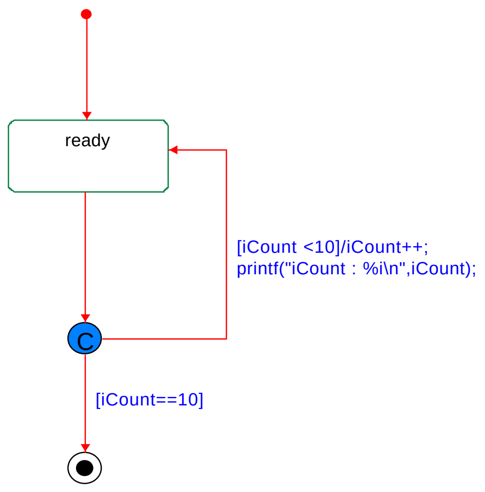


Figure 60 : Statechart without ambiguity

In conclusion, never use an action on an incoming transition to a condition connector to avoid ambiguous behavior.

B.7 Statechart elements of Rhapsody.

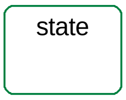
This appendix describes the elements, which can be used in the statecharts of Rhapsody. For each element the symbol, the symbol description, the label (if applicable) and its description/features are given.

The following elements can be used in a Rhapsody statechart:

- State
- Termination state
- Transition
- Default transition
- And line
- Condition connector
- History connector
- Termination connector
- Junction connector
- Join Synch bar
- Fork Synch bar

State:

Symbol:



Symbol description: A square box with rounded corners.

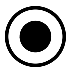
Label: Name of the state.

Description and features:

A state can have an action upon entry or exit. A state can have reactive behavior called 'reaction in state'. When a state has an action or reaction the name will have a trailing greater than sign. A state can have sub-states; the state with sub-states is represented as a separate statechart. When a state has sub-states the name will have three trailing dots.

Termination state:

Symbol:



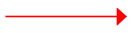
Symbol description: A circle with a black dot in the middle.

Description and features:

The termination state is a kind of a state. A termination state works as a 'local termination'. The object they are is not destroyed. A Termination state cannot have any outgoing transitions or any actions within the state.

Transition:

Symbol:



Symbol description: An arrow with a filled in arrowhead.

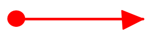
Label: trigger[guard]/action

Description and features:

A transition is the mechanism to go from one state to another. Three features specify the behavior of a transition: a trigger, a guard and an action. A transition doesn't have a name.

Trigger:An event or a triggered operation that can cause the transition to be taken.
Guard: A Boolean expression that must evaluate to true for the transition to occur.
Action: Code statement(s) to be executed when the transition is taken.

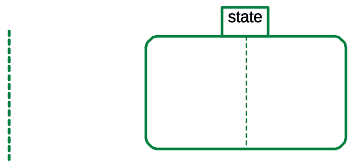
Default transition:

Symbol: 

Symbol description: An arrow with a filled in arrowhead. The origin of the arrow is a bead.

Description and features:
A special type of transition, the default transition, is a transition to the initial state of the statechart. (Thus setting a default transition to a state will make that state the initial state.) The default transition will be taken when the statechart is activated and is obligatory in a statechart. The default transition can have neither a trigger nor a guard since it needs to be taken.

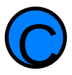
And line:

Symbol: 

Symbol description: a dotted line. (In a state)

Description and features:
An ‘and line’ can create one or more parallel state machines within one state. The name of the state will be placed on top of the state in a separate box.

Condition connector:

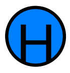
Symbol: 

Symbol description: A circle with the character ‘C’ in the middle.

Description and features:
A Condition connector is a way of visually representing if-then-else conditions. The following rules apply to a condition connector:

- Only one entering transition.
- Any number of "if" conditions, but only one else condition.
- Branching segments can be nested.
- Branches entering a condition connector can contain triggers.
- Branches exiting a condition connector cannot contain triggers.

History connector:

Symbol: 

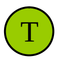
Symbol description: A circle with the character ‘H’ in the middle.

Description and features:
A history connector recalls the most recent active configuration of a state and its sub-states. A history connector transitively restores all of the sub-configurations that originated in the state. A transition originating in a history connector denotes the history default. The history default transition is taken if no history existed prior to entry into the history connector. Each state can have only one history

connector. Thus a history connector can have only one outgoing transition. (However, a history can have multiple incoming transitions.)

Termination connector:

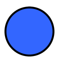
Symbol:



Symbol description: A circle with the character ‘T’ in the middle.

Description and features:
The termination connector terminates the statechart and destroys its own instance. Thus a transition to a termination connector causes an object to delete itself. A termination connector cannot have any outgoing transition. (Note for Activity diagrams; while terminations connectors have the same appearance as a termination states in Activity diagrams, termination states do not destroy the object they are in.)

Junction connector:



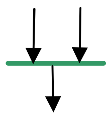
Symbol:

Symbol description: A solid circle.

Description and features:
A junction connector joins more than one transition into a single, outgoing transition. This allows you to combine several segments into a single graphical description or use a common transition suffix.

Join Synch bar:

Symbol:

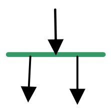


Symbol description: A thick solid line.

Description and features:
A Join synch bar consists of two or more incoming transitions joined into one outgoing transition. The incoming transitions must be null transitions, the outgoing transition can be specified. The join synch bar ‘synchs’ the incoming transitions and the outgoing transition can be taken if all the incoming transitions are taken.

Fork Synch bar:

Symbol:



Symbol description: A thick solid line.

Description and features:
A Fork synch bar consists of one incoming transition forked into two or more outgoing transitions. The incoming transition can be specified, the outgoing transitions must be null transitions. The fork synch bar ‘forks’ the incoming transition into multiple outgoing transitions.

B.8 Rhapsody Specifics

The browser only shows properties that are created by the user on default. This default behavior can be changed in order to let the browser display properties that are generated by the tool itself. This greatly improves the understanding of the tool and the project under development. E.g. a relation between two classes will now show the addition of the methods “getItsClassname” and “setItsClassname”. These two methods would otherwise be invisible in the browser.

To display all the properties in the browser:

```
Project Properties->Subject “CG”->Metaclass “CGGeneral”-> Set “GeneratedCodeInBrowser”   True
```

There is a bug, or a hidden property, in Rhapsody that will restrict the “generated code in browser” only to be shown in base classes. In other words, no generated code will be shown for nested classes. The only other option to view the properties of a nested class is to look in the generated code itself by selecting the class and editing it.

Don’t forget to give attributes an initial value. Although this suggestion isn’t specific to Rhapsody but applies more to C/C++ it is very important to remember. By default the tool Rhapsody doesn’t set the initial value of an attribute to a specific value, this can lead to unpredicted behavior if not taken into account.

By default Rhapsody uses I/O streaming (“iostream.h”). Functions like cout and cin are used to write and read to and from the standard output and the standard input. It is important to add the “endl” after each cout function in order to display the message timely or at all. If the “endl” operator is omitted the message will be written to the standard out on the next “endl” or when a certain amount is written to the standard out. In case of a single message written to the standard out without an “endl” operator the message may not be visible on the screen at all.

Instead of using the streaming I/O routines, the standard I/O routines could be used. Since standard I/O is an ANSI standard it should be used in cases where the deployment target is other than win32 to ensure integrity.

Although three different access methods of an attribute (public, protected and private) can be specified the access method of the attribute will always be of type protected. The specification of the access method only applies to the autogenerated methods to access the attribute; getAttribute and setAttribute. This means that the autogenerated methods to access attributes should always be used since the attribute itself is protected.

The default transition in a statechart cannot have a trigger or a guard. Rhapsody allows filling in the trigger –and guard field but after entering the value it will complain that no triggers or guards are allowed on a default transition.

The browser is the actual repository of the project. For example, when removing an association in the browser all the occurrences of the association in all the views will be removed. On the other hand when an association is removed from an object model diagram the occurrence of this association is not removed from the browser. In this case the association is no longer visible in the object model diagram although the association is still there. This implication of this suggestion is twofold; be careful when removing relations etcetera and remember that the different views offered may not reflect the model completely.

Customizing project properties

Rhapsody has a large amount of project properties that can be customized. Usually the project properties are customized by selecting the project properties in the menu and changing the desired value. However, properties that need to be changed on a regular basis can have there default values changed.

The site.prp file allows to change these default project properties. These settings override the default settings in the factory*.prp files. See the help file of Rhapsody for more information on this file.

The site.prp at the bottom of this page reflects some usefull changes which are :

- All attributes, methods, initializations that are autogenerated are visible in the browser.
- The concurrency of an object is set to active.
- An object diagram only displays the object names and instance names.

Listing of site.prp file:

```
Subject CG
      Metaclass CGGeneral
            Property GeneratedCodeInBrowser Bool "True"
      end
      Metaclass Class
            Property Concurrency Enum "sequential,active" "active"
      end
end
Subject ObjectModelGe
      Metaclass Class
            Property ShowAttributes Enum "All,None,Public,Explicit" "none"
            Property ShowOperations Enum "All,None,Public,Explicit" "none"
      end
end
```

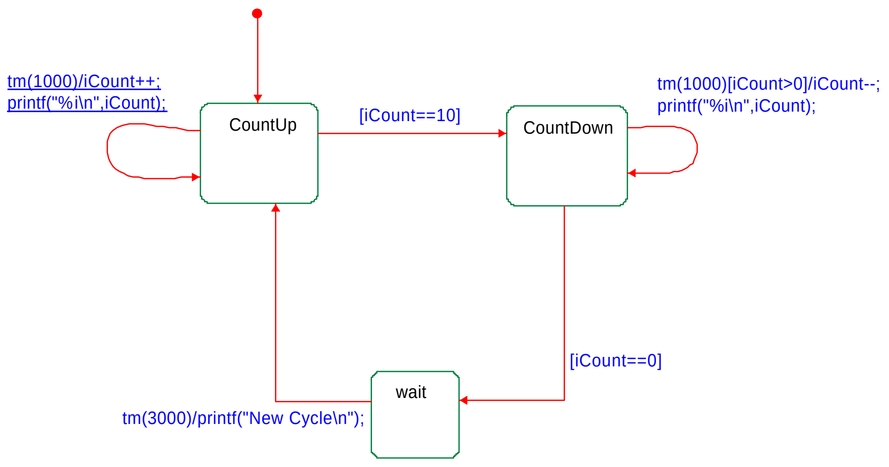
B.9 Counter Example

This example demonstrates how to design a simple counter in Rhapsody. First the design process will be described in a straightforward fashion. Second some comments on the design process and Rhapsody will be made.

The behavior of the counter to be designed:
1. Count to ten with intervals of a second.
2. Count down to zero with intervals of a second.
3. Wait for 3 seconds.
4. Repeat.

Design in Rhapsody
- Create a new project with name “counter”.
- Expand component node and rename “DefaultComponent” to “CounterApp”.
- Expand CounterApp node, expand Configurations node, select DefaultConfig, go to settings tab:
    - Change instrumentation from “none” to “animation”.
    - Add “stdio.h” to the Standard Headers section.
- Expand packages node, select package Default and add new class with name “Counter”.
- Select class Counter and add a new attribute of type “int” with name “iCount”.
- Select class Counter and add a new constructor, to the implementation section add “iCount=0;”.
- Select class Counter and create a new statechart.

Create the statechart like this:



- Select package Default and add new instance of Class Counter.

Now in the Code menu tab select generate/make/run. The animation toolbar will appear, press go. In the Tools menu tab select Animated Statechart, select counter[0]. The CounterApp application will run in a dos-window while the animated statechart shows the current state of the application.

Build & Run Counter on Linux OS

- Add new configuration named “LinuxConfig”.
- Open configuration:
  - Change directory to a directory on the mapped drive of the Linux machine.
  - Select Linux in the environment tab.
- Set “LinuxConfig” as active configuration.

Now in the Code menu tab select generate/make/run. The animation toolbar will appear, press go. In the Tools menu tab select Animated Statechart, select counter[0]. This time the CounterApp application runs on the remote Linux machine. The output is visible in the plink shell while the animated statechart shows the current state of the application.

B.10 Implementation of CTC++ package with postprocessing

This appendix desribes the implementation of the CTC++ package with postprocessingen. The implementation constist out of the following three parts:

1. The generated CTCPP source files need to be replaced by the “real” CTCPP source files.
2. No object files need to be generated for the Rhapsody CTCPP package sources.
3. The “real”CTCPP package library needs to be linked to the Rhapsody application.

The removal of the files and the adaptation of the makefile can be automated. This automation is also described in this appendix.

Replacement

The replacement of the generated source files by the “real” source files will be done by deleting the generated source files and adding the CTCPP sources to the include path. The include path can be adapted in the configuration node in the main browser tree of Rhapsody.

No Object Files

The project makefile needs to be adapted in order not to generate object files for the empty CTCPP classes.
In the makefile there is an “OBJS=” label followed by all the objects that need to be made.
`OBJS=Parallel.o Default.o …`
The empty CTCPP classes need to be removed.

Later in the makefile followed by the specification of the object:

```
Parallel.o : Parallel.cpp Parallel.h
             @echo Compiling Parallel.cpp
             $(CREATE_OBJ_DIR)
             @$(CC) $(ConfigurationCPPCompileSwitches)  -o Parallel.o\ Parallel.cpp
```

The empty CTCPP classes need to be removed.

The non-CTCPP objects that need to be created that use CTCPP elements will have a specification in the makefile containing references to the header files of the empty CTCPP generated classes.
Default.o : Default.cpp Default.h Parallel.h Channel_of_Integer.h

```
            @echo Compiling Default.cpp
            $(CREATE_OBJ_DIR)
            @$(CC) $(ConfigurationCPPCompileSwitches)  -o Default.o Default.cpp
```

The references to header files to the empty CTCPP classes need to be removed.

Linking

The linking of the library can be achieved by specifying the library in the configuration node in the main browser tree of Rhapsody.

Automation

The automation consist out two scripts `PostProcessScript` will remove the abundant source files and calls the `UpdateMakefile.sed` script to update the make file.

Postprocess Script

Note the script will remove files, beware of calling the script in the wrong directory.

The script can be called in the following manner:
./PostProcessScript “Makefile.mak”

```
#!/bin/bash
#PostProcessScript for the CTCPP package

Makefile=$1

echo Update makefile : $1

cp $1 $1.tmp
~/scripts/UpdateMakefile.sed $1.tmp > $1
rm $1.tmp

remove()
{
  if [ -f "$filename".cpp ]
  then
    rm -fv "$filename".cpp
  fi

  if [ -f "$filename".h ]
  then
    rm -fv "$filename".h
  fi
}

for filename in Process \
                Parallel \
                Sequential \
                Alternative \
                PriParallel \
                PriAlternative \
                Channel_of_Any \
                Channel_of_Boolean \
                Channel_of_Byte \
                Channel_of_Char \
                Channel_of_Double \
                Channel_of_Dword \
                Channel_of_Float \
                Channel_of_Integer \
                Channel_of_Long \
                Channel_of_Object \
                Channel_of_Reference \
                Channel_of_Word \
                Reader \
                Writer
do
  remove
done
```

UpdateMakefile.sed script

```
#!/bin/sed -f
#
# this file will update the makefile for use of the CTCPP library

s/Process.h//
/Process.o/d
/Process.cpp/d

s/Parallel.h//
/Parallel.o/d
/Parallel.cpp/d

s/Sequential.h//
/Sequential.o/d
/Sequential.cpp/d

s/Alternative.h//
/Alternative.o/d
/Alternative.cpp/d

s/PriParallel.h//
/PriParallel.o/d
/PriParallel.cpp/d

s/PriAlternative.h//
/PriAlternative.o/d
/PriAlternative.cpp/d

s/Channel_of_Any.h//
/Channel_of_Any.o/d
/Channel_of_Any.cpp/d

s/Channel_of_Boolean.h//
/Channel_of_Boolean.o/d
/Channel_of_Boolean.cpp/d

s/Channel_of_Byte.h//
/Channel_of_Byte.o/d
/Channel_of_Byte.cpp/d

s/Channel_of_Char.h//
/Channel_of_Char.o/d
/Channel_of_Char.cpp/d

s/Channel_of_Double.h//
/Channel_of_Double.o/d
/Channel_of_Double.cpp/d

s/Channel_of_Dword.h//
/Channel_of_Dword.o/d
/Channel_of_Dword.cpp/d

s/Channel_of_Float.h//
/Channel_of_Float.o/d
/Channel_of_Float.cpp/d

s/Channel_of_Integer.h//
/Channel_of_Integer.o/d
/Channel_of_Integer.cpp/d

s/Channel_of_Long.h//
/Channel_of_Long.o/d
/Channel_of_Long.cpp/d
```

```
s/Channel_of_Object.h//
/Channel_of_Object.o/d
/Channel_of_Object.cpp/d

s/Channel_of_Word.h//
/Channel_of_Word.o/d
/Channel_of_Word.cpp/d

s/Reader.h//
/Reader.o/d
/Reader.cpp/d

s/Writer.h//
/Writer.o/d
/Writer.cpp/d
```

C Appendices Chapter 5

C.1 Naming conventions for safeguard design.

- The abbreviation “ei” stands for electronic interface.
- The abbreviation “con” stands for controller.
- The abbreviation “sm” stands for safeguard monitors.
- The channels connected to the electronic interface are named “input_ei_#sensor#” and “output_ei_steering”. The “input” channels refer to communication coming from the electronic interface. The “output” channels refer to communication going to the electronic interface.
- The channels connected to the controller are named “ouput_con_#sensor#” and “input_con_steering”. The “output” channels refer to communication going to the controller. The “input” channels refer to communication coming from the controller.
- The channels between the input shutter and the safeguard monitors are named “sm_#sensor#” and “input_shutter_#”. The channels named “sm_#sensor” are intended to pass information from the input shutter to the safeguard monitor. The channels named “input_shutter_#” are intended to receive information. (Reference point the input shutter.)
- The channels between the safeguard monitors and the output shutter are named “sm_#” and “output_shutter_#”.
- The channels connected to the safeguard monitors are named:
  “#monitor#_#sensor#”, channels as sensor input for the monitor.
  “#monitor#_#is#_#”, channels as output to the input shutter.
  “#monitor#_#os#_#”, channels as output to the output shutter.
- The relation to channels that will be used as input are named:
  “chan_input_#source#_#type#” and “chan_input_#type#” if source is clear.
- The relation to channels that will be used as output are named:
  “chan_ouput_#dest#_#type#” and “chan_ouput_#type#” if destination is clear.

C.2 Coupling 20-sim to Rhapsody as test enviromnent

A Rhapsody model is designed to model a safeguard for JIWY. The model should first be properly tested before applied to JIWY. Different test scenarios can be created in Rhapsody but another possibility is to model the mechatronics in 20-Sim. This appendix describes how a 20-Sim model can be coupled to a model in Rhapsody.

20-Sim facilitates an easy mechanism to communicate with the environment. 20-Sim can call a dynamic link library (DLL) at every integration step while simulating. The values given to the DLL can be the states/variables of the system and the returned values can be the control signals for the model. In order to create the coupling between 20-Sim and Rhapsody, the application should contain a mechanism to read the values supplied by the DLL and return values to it.

Since both the DLL and executable reside somewhere in memory and there is no initial reference from the one to the other a simple file-access mechanism is used for the communication. Figure 61 shows an overview of the coupling mechanism used.

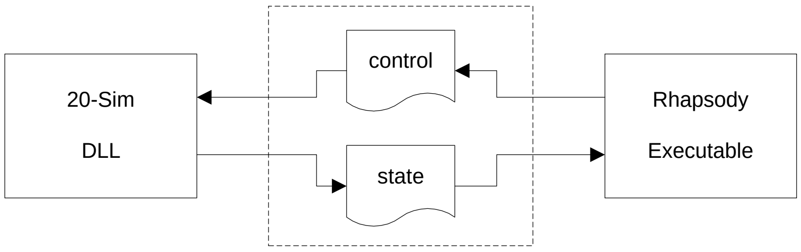


Figure 61: Coupling Mechanism

The coupling works in the following manner:

- The DLL writes the values to a file ‘state.txt’ and validates the file. Then the DLL will poll whether the file ‘control.txt’ is valid or not (and keeps polling until the file is valid). When the file is valid it will read the values and then invalidate the file.
- The Rhapsody executable will poll whether the file ‘state.txt’ is valid or not (and keeps polling until the file is valid). When the file is valid it will read the values and then invalidate the file. Then the executable will write the values to the file ‘control.txt’ and validates the file.

This implementation will not be very fast because of the file-access and the polling mechanism. The file-access can be avoided without making major changes to mechanism by implementing memory mapped I/O. The polling mechanism can only be effectively avoided in a different implementation strategy.

The 20-Sim DLL call is:

```
output = dll('coupling.dll','Communicate',input);
```

The source code for coupling.dll in case of one input and output:

```
#include <windows.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define DllExport __declspec( dllexport )

extern "C"
{

DllExport int Communicate(double *inarr, int inputs, double *outarr, int outputs,
int major)
{
  if( inputs != outputs ) return 1;

  FILE *fp;
  int iValid;
  int iWait;
  double dOutValue;
  double dInValue;
  dOutValue=inarr[0];

  //write data
  fp=fopen("c:\\temp\\state.txt","wb");
  iValid=0;
  fwrite(&iValid,sizeof(iValid),1,fp);
  fwrite(&dOutValue,sizeof(dOutValue),1,fp);
  fclose(fp);

  Sleep(2); //wait for rhapsody to return value

  //check if data is valid in control.txt
  iWait=1;
    while (iWait==1){
      fp=fopen("c:\\temp\\control.txt","rb");
      fread(&iWait,sizeof(iWait),1,fp);
      fclose(fp);
      Sleep(1); //maybe sleep here for a while
    }

  //read data from control.txt
  fp=fopen("c:\\temp\\control.txt","rb");
  fread(&iWait,sizeof(iWait),1,fp);
  fread(&dInValue,sizeof(dInValue),1,fp);
  fclose(fp);

  //invalidate data in control.txt
  fp=fopen("c:\\temp\\control.txt","wb");
  iValid=1;
  fwrite(&iValid,sizeof(iValid),1,fp);
  fclose(fp);

  outarr[0]=dInValue;
  return 0;
  }
DllExport int Initialize() {
  return 1; // Indicate that the dll was initialized successfully.
}
DllExport int Terminate() {
  return 1; // Indicate that the dll was terminated successfully.
}
}
```

References

Hilderink, G.H. (2002), A graphical Specification Language for Modeling Concurrency based on CSP, *Proc. Communicating Process Architectures 2002*, 15-18 Sep 2002, Reading UK, P. W. James Pascoe, Roger Loader, Vaidy Sunderam (Ed.), pp. 255-284, ISBN:

Jovanovic, D., G.H. Hilderink and J.F. Broenink (2002), A communicating Threads -CT- case study: JIWY, *Proc. Communicating Process Architectures 2002*, 15-18 Sep 2002, Reading UK, P. W. James Pascoe, Roger Loader, Vaidy Sunderam (Ed.), pp. 311-320, ISBN:

OXF-Guide (2002), *Rhapsody in C++, Execution Framework Reference Guide*, I-Logix,

Stephan, R.A. (2002), *Real-time Linux in Control Applications Area*, MSc, Dept. Electrical Engineering, University of Twente, Enschede.

UserGuide (2002), *Rhapsody User Guide*,

OMG-UML (2001), *OMG-Unified Modeling Language Guide version 1.4*,http://www.omg.com,

Drunen, J.M.v. (2000), *Realization of link drivers implementing CSP-channels on 20-controller*, MSc thesis, Faculty of Electrical Engineering, Control Laboratory, University of Twente, Enschede, Netherlands.

Hilderink, G.H., A.W.P. Bakkers and J.F. Broenink (2000), A Distributed Real-Time Java System Based on CSP, *Proc. The third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2000*, Newport Beach, CA, (Ed.), pp. 400-407, ISBN:

Schneider, S. (2000), *Concurrent and Real-Time Systems: The CSP approach*, Wiley,

Booch, G., J. Rumbaugh and I. Jacobson (1999), *The Unified Modeling Language, User Guide*, Addison Wesley, 0-201-57168-4.

Douglass, B.P. (1999), *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison Wesley Longman,

Douglass, B.P. (1998), *Real-Time UML: developing efficient objects for embedded systems*, Addison Wesley Longman, 0-201-32579-9.

Welch, P.H. (1998), Java Threads in the Light of occam/CSP, *Proc. Architectures, Languages and Patterns for Parallel and Distributed Applications, WoTUG-21*, Amsterdam, P. H. W. a. A.W.P.Bakkers (Ed.), pp. pages 259-284, ISBN:

Roscoe, A.W. (1997), *The Theory and Practice of Concurrency*, Prentice Hall,

Grebe (1993), *A Flexible High Speed Distributed Control System for Aircraft Testing, Transputer Applications and Systems*, IOS Press,

Welch, P. and G. Justo (1993), High-Level Paradigms for Deadlock-Free High-Performance Systems, *Proc. Transputer Applications and System*, T. U. o. Kent (Ed.), pp. ISBN:

Hoare, C.A.R. (1988), *INMOS Limited Occam 2 Reference Manual*, Prentice Hall International Series in Computer Science, 0-13-629312-3.

Hoare, C.A.R. (1985), *Communicating Sequential Processes*, Prentice Hall, 0-13-153271-5 (0-13-153289-8 PBK).