University of Twente

EEMCS / Electrical Engineering
Control Engineering

Design of animation and debug facilities for gCSP

Hans van der Steen

MSc report

Supervisors:
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
ir. M.A. Groothuis

June 2008

Report nr. 020CE2008
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

Communicating Sequential Processes (CSP), is a notation for describing concurrent systems, which can be analyzed algebraically. To use the theory of CSP for designing embedded software, the gCSP (graphical CSP) tool was created. gCSP is capable of creating CSP models by drawing diagrams (gCSP diagrams) according to the CSP language. By using the CSP algebra, the FDR2 (Failure Divergence Refinement) tool can check a gCSP model on deadlocks or livelocks. gCSP is capable of generating C++ code of its diagrams. Combining this code with the CT (Communicating Threads) execution framework, an executable (CT program) can be created. The CT program can be executed on a PC or (real-time) hardware platform, for testing its execution behavior. Because the CT execution framework only supports textual debug output as feedback on the execution of a CT program, it is hard to test and validate its execution behavior.

The goal of this project is to improve this feedback, to give the user more insight about the execution behavior of a CT program. To achieve this, animation is chosen to be appropriate for this purpose. By animation of a CT program, the execution of the program will be graphically shown. Animation can be used to analyze or debug the execution behavior of a CT program. To make this possible, an animation framework was designed and implemented.

Analysis is performed on the way the animation framework should be designed and implemented. Both gCSP and the CT execution framework are extended to be usable in the animation framework. gCSP is used to graphically show the execution of a CT program in its diagrams. The CT execution framework is used to generate information on the execution of a CT program. This information is used by gCSP for animation purposes.

A TCP / IP connection is used to connect a CT program to gCSP. A CT program uses this connection to send the state of its CT processes, CT constructs and CT channels; and the contents of its CT channels and the ready-queue of the CT kernel, to gCSP.

To support a user friendly way of analyzing a CT program while it is executing, the CT execution framework is extended with debug functionality. A CT program can be started and stopped, it can be paused by setting breakpoints and it can be executed step by step (stepper).

To animate the states of processes, constructs and channels color-based animation is used. A different color represents a different state. To animate the contents of channels and the ready-queue of the CT kernel, textual based animation is used. The content of a channel is shown in a tooltip window, when the user hovers the mouse over the channel. To be able to use animation for later analysis, a textual-based history can be saved as a text file on hard-disk.

All the specifications are designed and implemented. Tests were performed on the practical use of animation, the execution behavior of gCSP models and the C++ code generator of gCSP. These tests were discussed shortly and a conclusion was given. From these tests it was concluded that animation in gCSP is useful. Animation can be used to analyze the execution behavior of a CT program, and it can be used to find the cause of strange execution behavior. Without animation it would be probably harder, to analyze and "debug" the execution behavior of a CT program. The goal of the project has been met.

Recommendations for future work are to redesign the internal database and to improve the C++ code generator of gCSP. The user should be given the choice to enable or disable the optimizations of the CT execution framework. A real-time logger should be implemented in the CT execution framework for off-line analysis of the execution behavior of a CT program. A simulator should be added to the animation framework, which can use the data from the real-time logger as input for simulating a previous executed CT program.

Samenvatting

Communicating Sequential Processes (CSP), is een notatie voor het beschrijven van concurrent systemen, dat algebraïsch geanalyseerd kan worden. Om de CSP theorie te gebruiken bij het ontwerpen van embedded software, is de gCSP (grafisch CSP) tool gemaakt. gCSP kan grafische CSP diagrammen (gCSP diagrammen) tekenen. Gebruikma_kend van de CSP algebra, kan de FDR2 (Failure Divergence Refinement) tool een gCSP model controleren op deadlocks of livelocks. gCSP kan C++ code van zijn diagrammen genereren. Door deze te combineren met het CT (Communicating Threads) execution framework, kan een executabel (CT programma) gemaakt worden. Deze kan dan op een PC of een (real-time) hardware platform gedraaid worden, om het gedrag te testen. Omdat alleen textuele debug informatie ondersteund wordt door het CT execution framework, is het lastig om het gedrag van een CT programma te testen.

Het doel van dit project is om betere informatie te geven, zodat de gebruiker meer inzicht krijgt over het gedrag van een CT programma. Het gebruik van animatie is geschikt bevonden om dit doel te bereiken. Door het animeren van een CT programma, wordt het gedrag grafisch getoond. Dit kan gebruikt worden voor het analyseren en debuggen van zijn gedrag. Om dit mogelijk te maken is een animatie framework ontworpen en geïmplementeerd.

Er is geanalyseerd hoe het animatie framework ontworpen en geïmplementeerd moet worden. gCSP en het CT execution framework zijn uitgebreid, om gebruikt te kunnen worden in het animatie framework. gCSP wordt gebruikt om de uitvoering van een CT programma grafisch te tonen in zijn diagrammen. Het CT execution framework wordt gebruikt voor het genereren van informatie over het gedrag van een CT programma, om deze vervolgens te delen met gCSP.

Een TCP / IP verbinding wordt gebruikt om een CT programma te verbinden met gCSP. Een CT programma gebruikt deze verbinding om de status van zijn processen, constructies en kanalen; en de inhoud van zijn kanalen en ready-queue van de CT kernel, te versturen naar gCSP

Om een gebruiksvriendelijke manier voor het analyseren van een draaiend CT programma te ondersteunen, is het CT execution framework met debug functionaliteit uitgebreid. Dit maakt het mogelijk om een CT programma te starten of stoppen, op pauze te zetten door middel van breakpoints en het stap voor stap te doorlopen van het programma.

Voor het animeren van de status van een proces, construct en kanaal worden verschillende kleuren gebruikt. Textuele animatie wordt gebruikt voor het animeren van de inhoud van kanalen en de ready-queue van de CT kernel. Als de gebruiker met zijn muis over een kanaal beweegt, wordt de inhoud van het kanaal getoond als tooltip. Om animatie te gebruiken voor analyse achteraf, kan een textuele historie als tekst bestand worden opgeslagen op harde schijf.

Alle specificaties zijn ontworpen en geïmplementeerd. Tests zijn uitgevoerd op het praktisch gebruik van de animatie, op het gedrag van gCSP modellen en de C++ code generator van gCSP. Een korte discussie is gehouden over deze tests en een conclusie is gegeven. Er is geconcludeerd dat animatie in gCSP handig is. Animatie kan voor analyse op het gedrag van een CT programma gebruikt worden, en voor het vinden van de oorzaak van ongewenst gedrag. Zonder animatie zou het waarschijnlijk lastiger zijn om het gedrag van een CT programma te debuggen of te analyseren. Het doel van het project is bereikt.

Aanbevelingen voor de toekomst zijn om de interne database van gCSP te herontwerpen en om de C++ code generator van gCSP te verbeteren. De gebruiker moet de mogelijkheid krijgen om de optimalisaties van het CT execution framework in of uit te kunnen schakelen. De implementatie van een real-time logger in het CT execution framework, om offline het gedrag van een CT programma te kunnen analyseren. Een simulator in de animatie framework, die data van de real-time logger als invoer kan gebruiken om een CT programma te simuleren.

Preface

This report marks the final step in my life as an Electrical Engineering student. After the four years I spend getting my Bachelor degree in Electrical Engineering at the Hogeschool Utrecht, I decided to study at the University of Twente trying to get a Master of Science degree. After a half year of pre-master classes, one and a half year of master classes and almost nine months of working at my MSc project, I have reached the end of my life as an Electrical Engineering student. I enjoyed these last three years, especially the specialization course Embedded System Design and my MSc project at the Control Engineering group.

I am thankful for all the support I received during my study. I would like to thank my direct supervisor Marcel Groothuis for his support and comments during my MSc project. My special thanks go to Jan Broenink, who provided me with this assignment and gave me the opportunity to work after my graduation at the Control Engineering group. I would also like to thank the rest of the people at the Control Engineering group, especially my fellow MSc students. I enjoyed the working atmosphere and the discussions at the coffee machine or during lunch and the 'MSc-weekly' meetings.

I would also like to thank my family, who were of great support for me throughout the years. Without them, I would not have made it to the point where I am now. Last but certainly not least, my thanks go to all my friends and fellow students. They certainly have given me an enjoyable time here.

T.T.J. van der Steen Enschede, June 2008

Contents

1 Introduction

This chapter starts by presenting the problem dealt within this project. Next, the project goal and motivation are given. This chapter concludes by presenting the approach followed in this project and the outline of the rest of this report.

1.1 Problem statement

Nowadays many machines are controlled by *embedded systems*. Because the complexity of embedded systems increases every year, methods are needed to structure the design of these systems. Defining a hierarchy and properly structuring the concurrency within this hierarchy can help dealing with the problem of this complexity. *Concurrent programming* provides such a hierarchy and therefore results in high quality software for complex embedded systems.

In concurrent programming, single tasks of the system are divided into a number of subtasks, which can for example be executed in parallel or in sequential. Describing the relationships between these tasks in a concurrency structure, it can be analyzed by means of formal methods like using a process algebra.

CSP (Communicating Sequential Processes) (Hoare, 1985) is a formal language for describing patterns of interaction in concurrent systems, which can be analyzed algebraically. To be able to use CSP in concurrent programming, the *gCSP (graphical CSP) tool* has been created. This tool is used at the Control Engineering group and the MSc course Real-Time Software Development (RTSD), to design embedded (control) software using the CSP theory.

gCSP is capable of creating CSP models by drawing graphical CSP diagrams. Different kinds of code can be generated from these diagrams. *CSP machine code (CSPm)* can be used by the FDR (Failure Divergence Refinement) tool, to formally check a CSP model on *deadlocks* or *livelocks*. C++ code can be used by the *CT (Communicating Threads) execution framework* for creating an executable (*CT Program*), which can be executed on a PC or a (real-time) hardware platform.

Verification and *validation* of an executing CT program is done by analyzing its execution behavior. It will be tested whether the program behaves like it is supposed to do. Because the CT execution framework only supports textual debug output, it is currently hard to get a clear view about the execution behavior of a CT program. To make it easier analyzing the execution behavior of a CT program, a more effective way of giving feedback on the execution of a CT program should be supported.

1.2 Goal of the project

The goal of this project is to improve the feedback on the execution of a gCSP-generated CT program. The intention is to give the user an effective way to get insight in the execution behavior of the CT program. The goal of this project is accomplished, if it can be demonstrated that the improved feedback provides people with better insight in the execution behavior of CT program than before.

1.3 Motivation of the project

Maljaars (2006), used gCSP to design and implement the control software of a production cell setup (ten Berge, 2005). During the execution of his CT program, some undesired execution behavior was observed, which could not be explained. Because only textual debug output as feedback on the execution of the CT program was available, it was difficult finding the cause of this behavior.

gCSP is also used for education purposes, during the MSc course Real-Time Software Development (RTSD). This course is used to get the students familiar with concurrent programming and the problems which can occur using it. The students will learn how they can deal with these problems by using the gCSP tool. During the assignments, the students had some difficulties understanding the way of using the CSP language. The CSP concepts were hard to understand, because they could not get a clear view about the execution behavior of their CT programs, using the textual debug output of the CT execution framework.

Because improvement of the MSC course RTSD and the development of embedded software at the CE department is desired, this project was stared.

1.4 Assignment approach

As mentioned in section 1.2, this project aims at improving the feedback on the execution of a gCSP-generated CT program. The use of animation was selected to be appropriate for achieving this, because "A picture is worth a 1000 words" and "An animation is worth a 1000 pictures".

To see what animation has to offer and in which way animation can be used, the programs Rhapsody (Telelogic, 2008), Ptolemy II (Eker et al., 2008) and 20-sim (Controllab Products, 2008) will be analyzed on their animation facilities. Analysis will be performed on the CT execution framework and the gCSP tool. What kind of feedback on the execution of a CT program is useful for animation? And in which way this should be animated in gCSP? From this analysis, a specification list will be created and will be used as a guideline for the design and implementation of the animation framework. After the implementation of the animation framework, tests will be performed to evaluate the animation and to demonstrate the advantages of having animation in gCSP. The practical use of animation and the execution behavior of CT programs with the use of animation will be tested. The results of the tests will be used to give a conclusion on this project and to give some recommendations for future work.

1.5 Report outline

Chapter 2 describes the theoretical and practical background information about the software design trajectory, CSP, the CT execution framework and the gCSP tool. In chapter 3 the analysis of the project is given. The results of the analysis are used to create a list of specifications. The specification list is used as a guideline for the design and implementation of the animation framework, which is described in chapter 4. Chapter 5 describes the performed tests to evaluate the animation framework and to demonstrate the advantages of having animation in gCSP. The results of these tests are discussed and a conclusion is given. Finally, chapter 6 presents the conclusions of this project and the recommendations for future work.

Additional information is given in three appendices. Appendix A gives a detailed description of the CT execution framework. Appendix B gives a detailed description of the design and implementation of the animation framework from chapter 4. Appendix C gives a user manual of how to use the animation framework, for the animation of a gCSP-generated CT program.

2 Background

This chapter describes the theoretical and practical background information of this project. The first section describes the software design trajectory of the CE group. The last sections introduce the concepts of CSP, the CT execution framework and the gCSP tool.

2.1 The software design trajectory of the CE group

The Control Engineering group has divided the design trajectory of embedded (control) software into four different steps (Broenink and Hilderink, 2001). Each step consists of one or multiple iterations and is verified by simulation or validated by testing. A graphical representation of this design trajectory is shown in figure 2.1

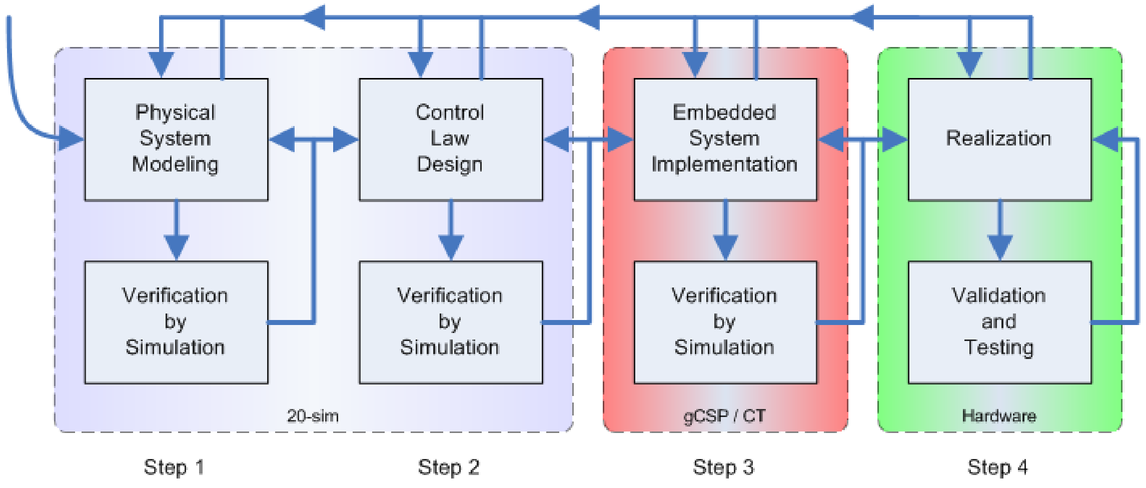


Figure 2.1: The software design trajectory of the CE group

A short description of each step is given below.

Physical System Modeling. The physical system is modeled by describing its dynamics. The model is verified by comparing the simulation results and the desired behavior of the model.

Control Law Design. The control law(s) are developed using the physical system model of step one. They are verified by simulation to check whether the desired control system behavior is achieved.

Embedded System Implementation. By stepwise refinement, the control law(s) of step two are converted to software. Also the control algorithms are integrated with user interfaces and system specific functionality (i.e. command structures). Each refinement step is verified through simulation.

Realization (Hardware). The software of step three is used to create an executable which can be executed on the real hardware or a target. The software behavior is validated by testing, whether the desired behavior is achieved.

This project focuses on the last two steps, because they are part of the software design trajectory of gCSP. The next sub-section describes these two steps in more detail.

2.1.1 The software design trajectory of gCSP

Figure 2.2 illustrates the software design trajectory of gCSP. It shows the work flow from creating a CSP model in gCSP, to the creation and testing of a gCSP-generated CT program.

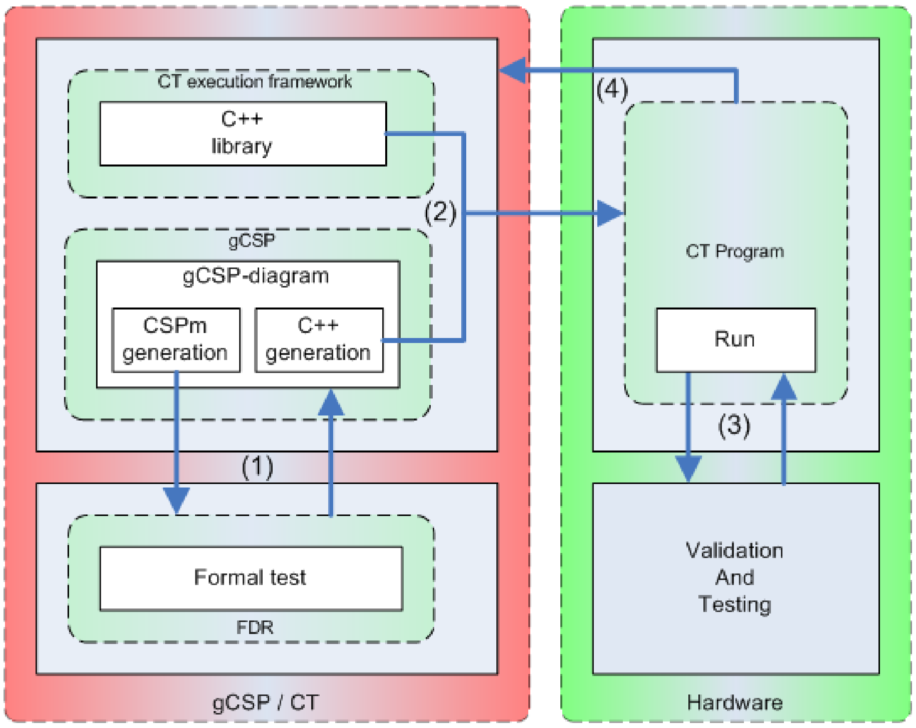


Figure 2.2: The software design workflow by using the gCSP tool

gCSP is capable of generating different kinds of code from its gCSP diagrams. CSPm code can be used by the FDR2 tool, to formally check the gCSP model on *deadlocks* or *livelocks* (see (1) in figure 2.2). C++ code together with the CT execution framework can be used for the creation of a CT program (2). The program can be tested by evaluating its execution behavior on a PC or hardware platform (target) (3).

To decrease the amount of iterations from a gCSP model to testing the execution behavior of a CT program on hardware (2, 3, 4), it is important to have good testing or analysis methods before the actual on-target test. Because this project focuses on improving "Validation and Testing", the next sub-sections give some methods to accomplish this.

2.1.2 Testing and analysis methods

To test or analyze the execution behavior of a CT program, detailed information about its execution is needed. To obtain this information, several methods can be used which are introduced here shortly.

Testing

"Testing is the process which is used to asses the quality of software. It starts with known conditions, uses predefined procedures, and has predictable outcomes. Software testing is the trajectory to verify the functionality of the software, and is therefore used in association with verification and validation." (Wikipedia, b)

Debugging

"Debugging is a methodical process of finding and reducing the number of bugs or defects in a program, thus making it behave as expected. Most of the times, debugging starts from an unpredicted outcome of a test, with the purpose to find the cause of it. Debugging of software often comes with: starting and stopping the program, setting *breakpoints* and walking step by step (*stepping*) through the program." (Wikipedia, a)

Data logging
"Data logging is the retrieval of data for a certain period of time. It is a passive process which is not supposed to change the behavior of a program or react at certain events. The data for all timestamps is stored and is available for post-processing after data logging has been finished. This data can then be used for further analysis." (Posthumus, 2007)

Monitoring
"Monitoring is the process of retrieving data at run-time, which can directly be used for post-processing. It is an active process which can react on certain events for example to prevent a system from damaging itself. Monitoring can be compared with sampling; only data at the sampled timestamps is available (snapshots). Monitoring is used for diagnosis or maintenance." (Posthumus, 2007)

Tracing
"Tracing is a specialized form of data logging, which is used to record information about the execution of a program. This information is typically used by programmers for debugging purposes, to diagnose where problems occur within their software." (Wikipedia, c) Trace information can be used, for example in:
- later analysis of the execution behavior of a program.
- the animation or debugging of a program during its execution.

2.2 The CSP language

"CSP (Communicating Sequential Processes) is a notation for describing concurrent systems (i.e., ones where there is more than one process existing at a time) whose component processes interact with each other by communication." (Roscoe, 1997)

CSP is a calculus for studying processes, which can be analyzed algebraically. It offers a process-oriented design of concurrent systems, a fundamental architectural vocabulary of building blocks: processes for capturing functional software components, synchronous (waiting rendezvous) channels for communication between processes and operators for composing order of execution among the processes.

2.2.1 CSP diagrams

A CSP diagram is a graphical notation of CSP, based on the GML language proposed by Hilderink (2002, 2003, 2005). The communication and composition aspects of CSP are described by the use of graphical representations. Section 2.4 gives an introduction how CSP is graphically represented within gCSP.

2.3 The CT execution framework

The CT execution framework from the University of Twente (Jovanovic et al., 2003; Hilderink, 2005), provides a C++ implementation of a process-oriented framework based on CSP. The next sub-sections give a short introduction of the most important parts of the CT execution framework. Detailed information about these parts can be found in Appendix A.

2.3.1 CT processes

CT processes are independent objects performing a specific task. Processes only interact with their environment through their communication interfaces which means that they do not know about the existence of other processes. Processes communicate with each other via channels (section 2.3.2) which are connected to the communication interfaces. A process can be composed of other processes and / or constructs which makes it a complex or parent process. Child processes communicate with each other using internal channels. If a child process wants to communicate to another process, the interface of its parent is used. Figure 2.3 shows how processes communicate with each other by using their communication interfaces.

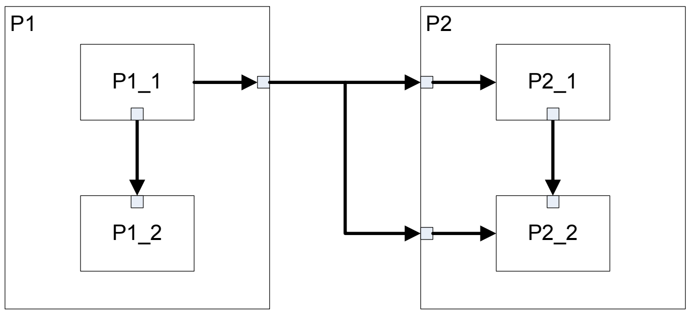


Figure 2.3: The way CT processes communicate with each other

The processes are represented as squares with their name in it. The small squares at the side of a process are representing the communication interfaces of a process. Processes P1_1 and P1_2 are using an internal channel of process P1 (1), to communicate with each other. To let process P1_1 communicate with process P2_1, the communication interface of process P1 (2) and P2 (3) are used.

2.3.2 CT channels

CT Channels are passive objects which are used by processes for rendezvous data communication. Parallel processes connected to a rendezvous channel, are synchronized (and scheduled) on channel communication. Channel communication is usually read- or write-only, making channel communication unidirectional. A channel can carry various data types like integers, doubles or floats, but just one type at the time. To support that a channel can be used by more then two processes at the same time, channels can be shared (one-to-any) or joined (any-to-one), but basically channels are used for one-to-one communication.

2.3.3 CT constructs

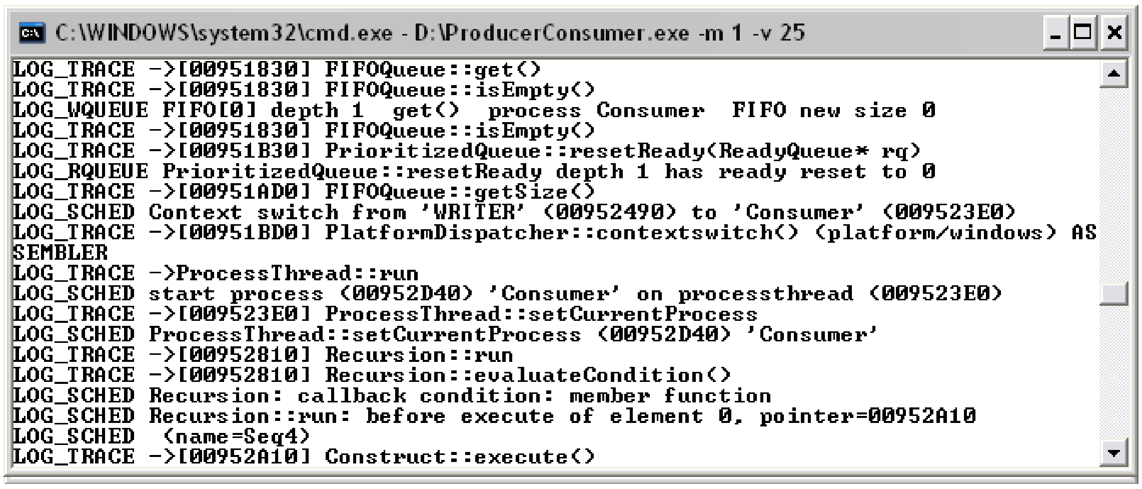
CT constructs are special processes, which execute their child processes in a specific order. A construct does not have communication interfaces. Its child processes are directly connected to a channel. Constructs together with the synchronization on channels are responsible for scheduling their child processes. The constructs are implementations of the CSP operators, sequential, (pri)alternative and (pri)parallel.

2.3.4 CT kernel

The CT kernel provides the low level means of supporting scheduling, ruled by the compositional constructs and channels. Because this kernel is integrated in the CT execution framework, the scheduling behavior is independent of the operating system it is executing on.

2.3.5 Textual debug output

The CT execution framework supports textual debug output, which can be used for debugging purposes. This output shows for example: *Context switches*, blocked processes and scheduler events. Figure 2.4 shows the debug output of a simple producer-consumer example.



```
C:\WINDOWS\system32\cmd.exe - D:\ProducerConsumer.exe -m 1 -v 25
LOG_TRACE ->[00951830] FIFOQueue::get()
LOG_TRACE ->[00951830] FIFOQueue::isEmpty()
LOG_WQUEUE FIFO[0] depth 1  get()  process Consumer  FIFO new size 0
LOG_TRACE ->[00951830] FIFOQueue::isEmpty()
LOG_TRACE ->[00951B30] PrioritizedQueue::resetReady(ReadyQueue* rq)
LOG_RQUEUE PrioritizedQueue::resetReady depth 1 has ready reset to 0
LOG_TRACE ->[00951AD0] FIFOQueue::getSize()
LOG_SCHED Context switch from 'WRITER' (00952490) to 'Consumer' (009523E0)
LOG_TRACE ->[00951BD0] PlatformDispatcher::contextswitch() (platform/windows) AS
SEMBLER
LOG_TRACE ->ProcessThread::run
LOG_SCHED start process (00952D40) 'Consumer' on processthread (009523E0)
LOG_TRACE ->[009523E0] ProcessThread::setCurrentProcess
LOG_SCHED ProcessThread::setCurrentProcess (00952D40) 'Consumer'
LOG_TRACE ->[00952810] Recursion::run
LOG_TRACE ->[00952810] Recursion::evaluateCondition()
LOG_SCHED Recursion: callback condition: member function
LOG_SCHED Recursion::run: before execute of element 0, pointer=00952A10
LOG_SCHED  (name=Seq4)
LOG_TRACE ->[00952A10] Construct::execute()
```

Figure 2.4: The textual debug output of a simple producer-consumer model

2.4 The gCSP tool

This section introduces the elements and features of the gCSP tool relevant to this project. Detailed information on using the tool is described in the PhD thesis of Jovanovic (2006) and the user manual of gCSP (Maljaars, 2007). The gCSP tool is a standard window SDI (Single Document Interface) application and programmed in Java (Sun, 2008), permitting availability on different platforms. Figure 2.5 shows the main window of gCSP.

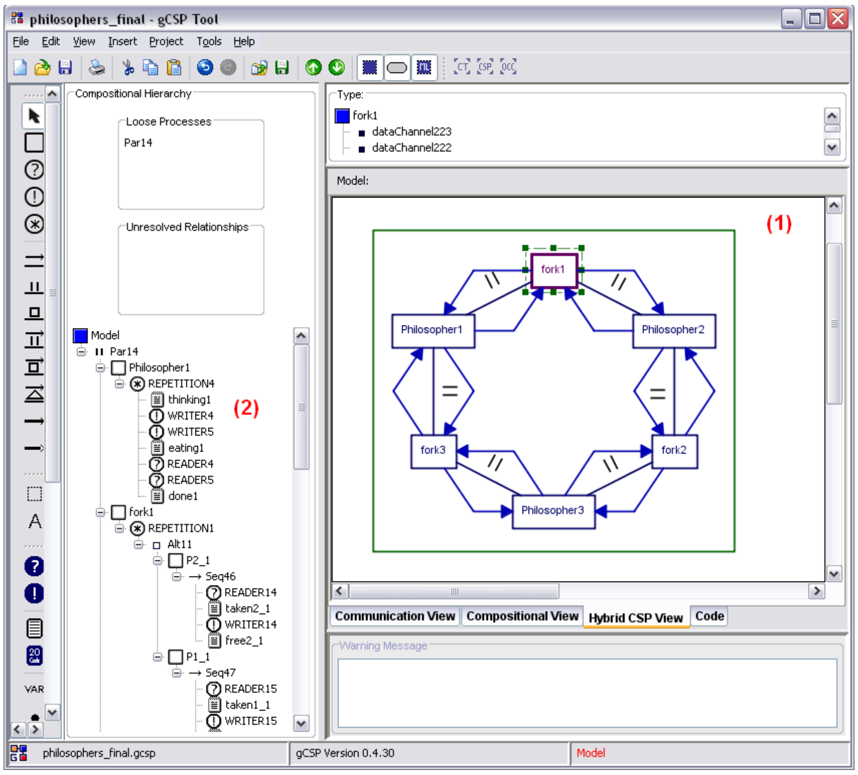


Figure 2.5: The user interface of gCSP, displayed in hybrid view

The two most important areas are the right pane with the CSP diagrams editor (1) and the left pane where the composition tree (C-tree) is shown (2). The next sub-sections introduce the different elements in the CSP diagrams editor and the last sub-section introduces the C-tree.

2.4.1 gCSP Processes

A gCSP process is the graphical representation of a CT process. All processes in gCSP are divided into two main groups: primitive processes and complex processes. Primitive processes can not contain children and their functionality is predefined. For example a writer is only writing on a channel and a reader only reads from a channel. A complex process on the other hand is composed of one or multiple processes (its children). An overview of the way processes are used in gCSP is described in the user-manual of gCSP (Maljaars, 2007).

2.4.2 gCSP data-channels

A data-channel is the graphical representation of a CT channel. Data-channels are also known as communication relationships within gCSP, and are represented as an arrow connected between processes. They are used to synchronize the processes connected to it. Since gCSP processes access data-channels through unidirectional communication interfaces (ports), data-channels are only used for unidirectional communication. This is the reason why in gCSP channels are drawn as a one way arrow with only one arrow head (figure 2.6).

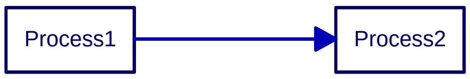


Figure 2.6: Two processes communicating via a unidirectional data-channel

To enable the use of bidirectional communication, a second data-channel can be drawn. This data-channel can be connected in opposite direction from Process2 to Process1. An example can be seen in figure 2.5 between the processes Fork1 and Philosopher1.

2.4.3 Compositional relationships

In gCSP, compositional relationships between processes are used to specify in which kind of CT construct processes needs to be placed. Each process has to be placed in a compositional relationship with another process. Compositional relationships are represented as straight lines adorned with a symbol of a CSP/CT operator. Figure 2.7 shows an example of three processes connected with compositional relationship using a parallel construct.

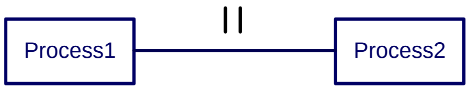


Figure 2.7: Two processes executing in parallel

Process1 and Process2 will be executed in parallel. If more then two processes need to be executed in parallel, multiple compositional relationships needs to be drawn. Figure 2.8 shows an example of a parallel construct, containing three processes.

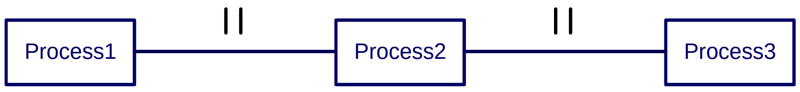


Figure 2.8: Three processes executing in parallel

One compositional relationship with a parallel CSP operator is drawn between Process1 and Processe2 and a second one is drawn between Process2 and Process3. The number of compositional relationships need to be drawn, is equal to the amount of processes composed in a CT construct, minus one.

2.4.4 Grouping

Grouping shows how CT constructs will be composed, and therefore which execution behavior is desired by the user. If different compositional relationships are used at the same hierarchy level, the desired execution order of the processes can be interpreted differently. Figure 2.9 shows an example of a gCSP diagram where it is not clear which execution behavior is desired.

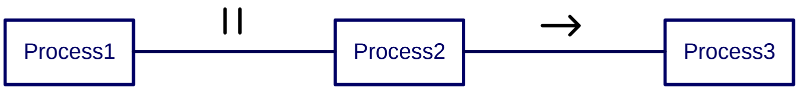
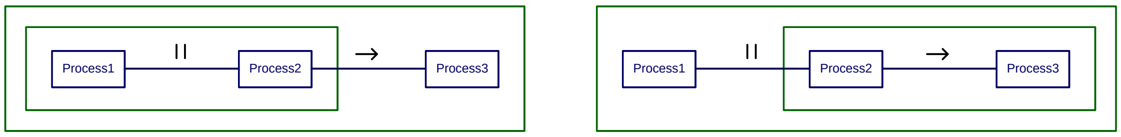


Figure 2.9: Compositional ambiguity

Two different compositional relationships are connected to Process2. It is not clear if Process2 should execute in sequence with Process3 and then in parallel with Process1, or the other way around. Grouping is used to solve this problem. A group is drawn as a green rectangle around processes and is used the same way as parenthesis in formulas. This is illustrated in figure 2.10.



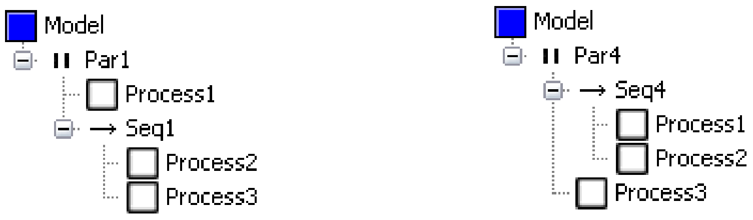
(a) Parallel-sequential processes (b) Sequential-parallel processes

Figure 2.10: Different ways of grouping

In CSP, figure 2.10(a) can be written as: $((Process1||Process2) \rightarrow Process3)$. Process1 and Process2 will first be executed in parallel and then in sequence with Process3. Figure 2.10(b) can be written as: $(Process1||(Process2 \rightarrow Process3))$. Process2 and Process3 will first be executed in sequential order and then in parallel with Process1.

2.4.5 The Composition tree

The Composition tree (C-tree) is a tree-based representation of the hierarchy of a gCSP model. Different ways of grouping will result in different C-trees. Figure 2.11 shows the C-trees which were created of the gCSP models of 2.10.



(a) Parallel-sequential (b) Sequential-parallel

Figure 2.11: Different ways of grouping

Differences between these C-trees can clearly be seen. In figure 2.11(a), Process1 executes in parallel (Par1) with the sequential (Seq1), containing Process2 and Process3. In 2.11(b), Process 3 executes in sequence (Seq2) with the parallel (Par2), containing Process1 and Process3. The execution behavior will differ if from both gCSP models the CT program is executed.

2.5 Conclusions

In this chapter the theoretical and practical background information of the software design trajectory of the CE group, the concepts of CSP, the CT execution framework and the gCSP tool has been given. The next chapter presents the analysis of this project, where the background information from this chapter is used.

3 Analysis

Animation can improve the feedback on the execution of a CT program. To make this possible, an animation framework needs to be designed and implemented. This chapter discusses the analysis of the parts which are useful for the design and implementation of the animation framework. This analysis is used to create a list with specifications, which is used as a guideline for the design and implementation of the animation framework.

3.1 Real-time animation

In this report, real-time animation is defined as the animation of a CT program, which is running at full speed. Because animation should be usable for analysis on the execution behavior of a CT program, it is important that all information is shown and that it is shown in the same order the program executes. A problem is that a CT program can execute faster than the human eyes and mind can process. This means that animating at full speed is not efficient to be used for the analysis of the execution behavior of a CT program while it executes. Maljaars (2006) has experienced the same kind of problem. The textual debug output of his CT program was printed faster to the screen then he could analyze. This is the reason why real-time animation should not be used in this project. Instead, the ability to real-time record (log) the execution of a CT program is desired. The logged data can then be used for later analysis after the CT program is finished or stopped its execution.

3.2 The animation framework

In the software design trajectory of gCSP, the generated C++ code of the gCSP-diagrams is used together with the CT execution framework for the creation of a CT program. To be able to animate the execution of a CT program, the animation framework should support the following:

- A graphical user interface to show animation in a graphical way.
- The simulation of a CT program, where its output can be used for animation purposes.

The most effective way is to use gCSP as the graphical user interface for the animation framework. For the simulation of a CT program, two options are possible:

1 A CT simulator in gCSP itself.
2 An external CT simulator.

In the first option, the animation framework can be implemented completely in gCSP. Because gCSP is written in java, a java version of the CT execution framework is needed for the simulation. A CT execution framework written in Java (*CTJ*) does exist, but it is not updated since the year 2003-2004. A lot of work is needed to get it back up to date again. Another issue is that the behavior of the Java version is difficult to compare with the behavior of the C++ version. A lot of testing should be done to verify that both execution frameworks do behave the same.

In the second option an external CT simulator is used. Making use of the C++ version of the CT execution framework is the most effective way. To let the CT execution framework function as a CT simulator for gCSP, it needs to be extended with:

1 A communication protocol to send information on the execution of a CT program to gCSP.
2 The ability to generate information on the execution of a CT program.

3.2.1 Conclusion

The most effective way is to make use of gCSP and the CT execution framework for designing and implementing the animation framework. The CT execution framework should be able to connect a CT program to gCSP, generate information about the execution of the CT program and send this information to gCSP. gCSP should be able to use this information for animation purposes.

3.3 The connection between a CT program and gCSP

To enable communication between a CT program and gCSP, a connection between them is needed. For choosing an appropriate communication protocol used for this connection, the communication protocol should meet the following communication requirements:

- The arrival of messages has to be guaranteed, using them for logging or analysis purposes.
- Messages needs to be received in order, which makes it usable for animation purposes.
- Remote connection should be possible, enabling animation of a CT program running on a real target, which does not support a graphical user interface.

The TCP/IP and UDP communication protocols are already supported by the CT execution framework and gCSP (java). They are analyzed to check whether they are appropriate to be used in the animation framework. They support remote connects, are relatively fast and easy to implement and are supported by a lot of platforms and operation systems.

3.3.1 TCP/IP versus UDP

The primary difference between TCP/IP and UDP lies, in their respective implementations of reliable messaging. TCP/IP includes support for guaranteed delivery, meaning that the recipient automatically acknowledges the sender when a message is received, and the sender waits and retries in cases where the receiver does not respond in a timely way.

UDP, on the other hand, does not implement guaranteed message delivery. A *UDP datagram* can get "lost" on the way from sender to receiver, and the protocol itself does nothing to detect or report this condition. UDP is sometimes called an unreliable transport for this reason.

UDP also works unreliably in the receipt of a burst of multiple datagram's. Unlike TCP/IP, UDP provides no guarantees that the order of delivery is preserved. In practice, UDP datagram's arrive out-of-order relatively infrequently, generally only under heavy traffic conditions.

3.3.2 Conclusion

The receiving order of messages should be guaranteed and all messages should be received. TCP/IP is chosen as communication protocol used for the connection between a CT program and gCSP. TCP/IP meets the communication requirements and UDP does not.

3.4 The execution feedback of a CT program

Feedback on the execution of a CT program should be created and used by gCSP for animation purposes. A selection has to be made of the information which is useful to know, for analysis on the execution behavior of a CT program. The next sub-sections describe which parts of the CT execution framework are important and can be used for this analysis.

3.4.1 CT Processes

The main task of a CT program can be divided into sub tasks, which can be executed by one or multiple processes. To know which task a CT program is executing at a specific time, the status of all processes of that CT program is needed. This information can be obtained by transferring the process-state of all processes. For example, the state of a process shows whether the process is running or blocked? To know which task is executing, the current running process should be known. Therefore it is important to be able to have access to the state of a process.

3.4.2 CT Channels

In CT, scheduling of the processes is performed on channel communication events. Depending on the state of a channel, a process which tries to use a channel may continue executing or will be blocked. What happens to the process can be predicted by observing the state of the channel. For example, a process will be blocked if it tries to write on a channel which is 'not ready'. The process which is 'ready to run' and listed at the top of the *ready-queue*, may use the CPU instead.

Besides the state of a channel, written data on a channel (contents) can also influence the way processes are scheduled. For example, the alternative compositional relationship reacts on process guards which can be set to monitor and react on the data written on a channel. If data is read from a channel and the value is put into a variable, depending on the guards of the alternative and this value, the alternative may execute a certain guarded process.

Another way a channel can influence the scheduling of processes, is when buffered channels are used. For example, if the buffer of a channel is full and a process tries to write data on the channel, the process will be blocked until there is free space in the buffer again.

Both the state and the contents of a channel can be used for analysis and prediction of the behavior of the processes, which makes it useful information while analyzing the execution behavior of a CT program.

3.4.3 CT Compositions

As described in section 2.4.4, different ways of grouping or using different kinds of compositional relationships can result in different execution order among processes. If the state of a composition is known, it is also known which processes may execute and in which order they will be executed. This information can be used to predict the way processes will be executed, making it useful information during analysis on the execution behavior of a CT program.

3.4.4 CT kernel

An important part of the CT kernel is the ready-queue of its scheduler. The ready-queue holds the list in which order the processes may execute on the CPU. This list will change during the execution of the CT program. Knowledge about the contents of the ready-queue can be used to validate the execution behavior of a CT program and also the scheduling of the CT execution framework. Another way it can be used is for prediction of the next running process if the current running process will be blocked.

3.5 Debug functionality

A CT program can execute faster then the human eyes or mind can process (section 3.1), which makes it hard to analyze a CT program while it is executing at full speed. To make it easier for the user, methods are needed which can deal with this problem. The next sub-sections will introduce some methods / functionality to accomplish this.

3.5.1 Selecting a part of interest

A simple way of making it easier for the user to analyze the execution behavior of a CT program is to decrease the amount of feedback information used for the animation. The information which is not of interest to the user should be filtered out. Not every part of a CSP model is relevant. For example, one part can be more critical than the other, which makes it more interesting for analysis. Because a CSP model is divided into processes, the user should be able to select which processes should be animated.

3.5.2 Setting breakpoints

In software development, a breakpoint is an intentional stop or pause location in a program, put in place for debugging purposes. During the interruption, the user has time to obtain the state of a program, to find out whether the program functions as expected.

3.5.3 Stepping

Stepping refers to the common debugging method of executing code, one line at a time. The state of a program can be examined, and related to the data before and after execution of a particular line of code. This allows evaluation of the effects of that instruction in isolation and thereby gaining insight in the behavior (or misbehavior) of the program.

3.6 Animation in gCSP

As already mentioned in section 1.4, animation will be used to graphically show the execution of CT program in a user friendly way. Because gCSP is already a graphical tool, it is obvious to design and implement animation in the gCSP diagrams of gCSP.

3.6.1 Animated gCSP diagrams

Different ways of animation can be used. Analysis is carried out to see which one fits for using it within gCSP. The following tools are used to get an indication of how animation can be used.
- Rhapsody (Telelogic, 2008)
- Ptolemy II (Eker et al., 2008)
- 20-sim (Controllab Products, 2008)

These tools all use colors to show the state of a program or model. In Rhapsody, a state of a state-chart will be colored purple if the state is activated and colored brown if it was stopped. Ptolemy II is using the same concept in its state-charts; a state will be colored red if it is activated. In 20-sim, colors are used to show the activity (state) of bondgraphs. The use of different colors at different states is giving a clear view of what a program or model is doing. Therefore this method will be prototyped for animation in the gCSP diagrams.

Besides the state of a program, the contents of channels and the ready-queue should also be available for animation. These do not have a finite number of states, which makes the use of colors not an option. Instead, they should be 'animated' in a textual way.

3.6.2 Execution history

The states of processes, channels and constructs are changing during the execution of a CT program. The previous state can not be seen if only color based animation is used. For analysis on the execution behavior of a CT program, it could be useful to look back in the history of the program. It should be possible to store the feedback on the execution of a CT program in gCSP and save it on hard-disk, which then can be used for later analysis.

3.7 Specifications

This section gives the list of the specifications, created on the basis of the previous sections.

1 **Communication requirements**
- The TCP/IP protocol should be used for the connection between gCSP and a CT program.

2 **Color based animation in the gCSP diagrams and the C-tree**
- State of processes, constructs and both ends of channels.

3 **Textual based animation in gCSP**
- The contents of a channel
- The contents of the ready-queue of the CT kernel

4 **Debug functions of the CT execution framework**
- Usable in a user friendly way.
- Selection of the part of the gCSP model which needs to be animated
- Setting breakpoints to pause the execution of a CT program
- Stepping through the execution of a CT program

5 **Textual representation of the execution history of a CT program**
- Save to hard-disk

3.8 Conclusions

Analysis is performed on how the animation framework should be designed and implemented and which communication protocol should be used for the connection between a CT program and gCSP. Which information about a CT program is useful for analysis on its execution behavior and in which way this should be animated in gCSP. On the basis of this analysis, a list of specifications is created (section 3.7). This list is used as a guideline for chapter 4, for the design and implementation of the animation framework.

4 The design and implementation of the animation framework

To design and implement the animation framework, both gCSP and the CT execution framework are extended to be used for animation. A CT program is also part of the animation framework, because the feedback on its execution is used for the animation in gCSP. Figure 4.1 illustrates where the animation framework is located in the design trajectory of gCSP.

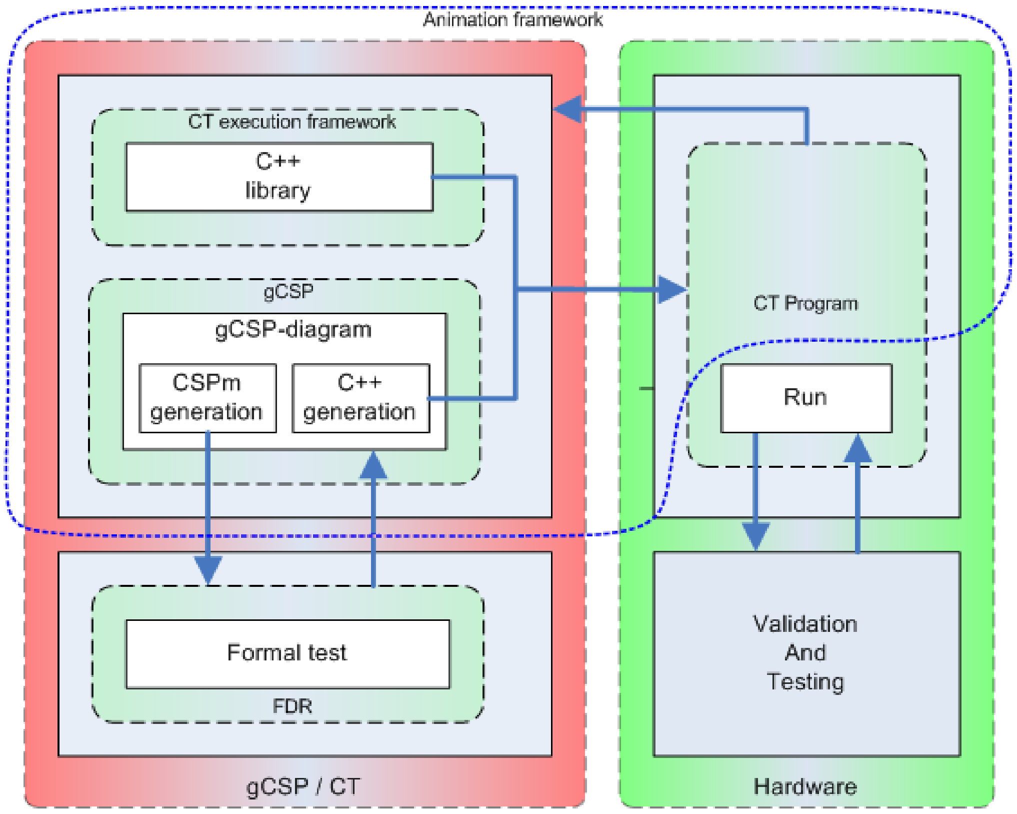


Figure 4.1: Animation framework in the design trajectory of gCSP

The next sections give a global description about the way the animation is designed and implemented. A more detailed description is given in Appendix B.

4.1 The connection between gCSP and a CT program

To let gCSP and a CT program communicate with each other, a TCP/IP connection (section 3.3) is created. Figure 4.2 illustrates the way this connection is used.

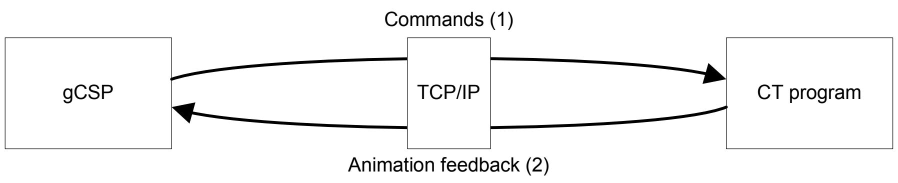


Figure 4.2: The connection between gCSP and a CT program

The connection is used for bidirectional communication. The CT execution framework sends information (animation feedback) on the execution of the CT program to gCSP (2), used for the animation in the gCSP diagrams. gCSP calls the debug functions of the CT execution framework (section 3.5) by sending command messages (1) to the CT program. Before the actual animation can be started, some initialization is needed first. This is illustrated in Figure 4.3.

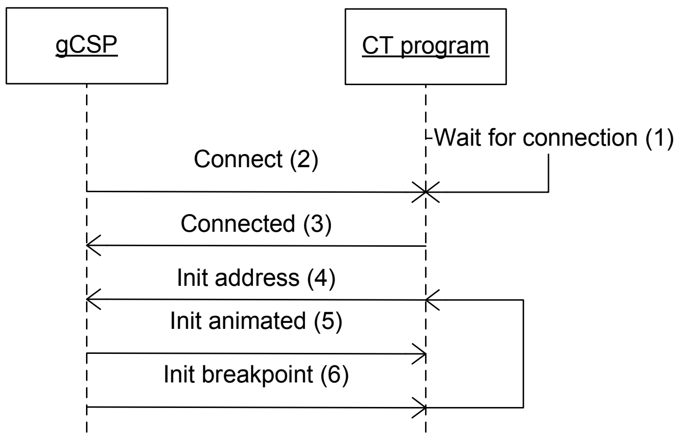


Figure 4.3: Initialization of the animation

First, the CT program needs to be started. The CT execution framework starts a TCP/IP server, to connect gCSP (the TCP/IP client) to the CT program. The CT program will then pause itself (1) and wait until gCSP is connected (2 and 3). After the connection is created successfully, the CT program and gCSP will exchange initialization data. To identify processes, constructs and channels within the animation framework, their *object pointers* (addresses) within the CT program are used. Because gCSP has no knowledge about these addresses, this information is sent first as initialization data by the CT program to gCSP (4). If an object with the same name exists in gCSP, the address is taken as valid. gCSP will then notify the CT program if the object needs to be animated (5) and if a breakpoint (6) is set in gCSP. If all objects are initialized, the CT program is ready to be executed and animated.

4.2 The extended graphical user interface of gCSP

The graphical user interface of gCSP is extended with new tools and windows, for using the animation framework in a user friendly way. The following sub-sections will describe them shortly.

4.2.1 The animation toolbar

To use the debug functionality 3.5 of the CT execution framework, the gCSP toolbar is extended with new tools. Figure 4.4 shows the extended toolbar, where the new tools are marked by red ellipses.

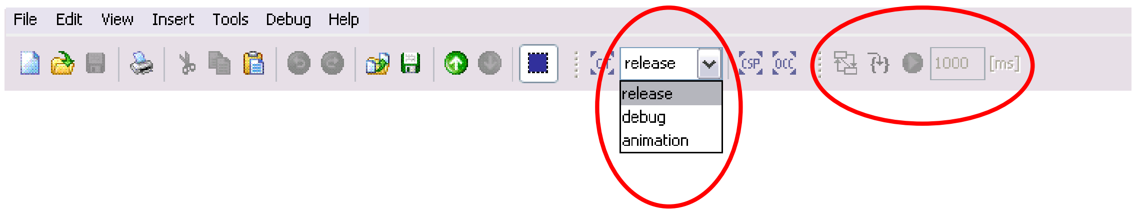


Figure 4.4: The animation toolbar in gCSP

Certain animation tools can be accessed by keyboard shortcuts, which mean the mouse pointer can be used for other purposes. The animation tools together with their keyboard shortcuts are described in table 4.1.

Table 4.1: gCSP animation toolbar

Tool	Description
Build Option	**release**: create a CT program which will finally run on the target **debug**: create a CT program with debugging support. Textual debug information will be printed to the screen. **animation**: create a CT program usable by the animation framework.
⧉	Start the animation by trying to connect to a CT program which is waiting for gCSP [*F*5]. If animation is already started it will be stopped [SHIFT-F5].
{•}	Step to the next running process of the CT program [*F*11].
▶ / ⏸	Start or stop periodic stepping with the frequency given in milliseconds.

4.2.2 The animation drop-down menu

To select a process to be animated and to configure breakpoints in gCSP, the drop-down menu of processes and constructs is extended with two new options: animate and breakpoint. This drop-down menu is shown in Figure 4.5, where the new options are marked by a red ellipse.

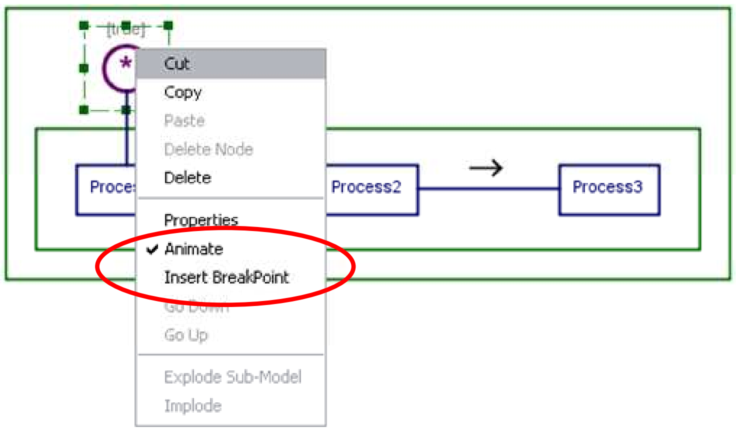


Figure 4.5: The animation drop-down menu in gCSP

Setting breakpoints

A breakpoint can be set on any kind of process or construct. By using the animation drop-down menu option "insert breakpoint" or pushing [*F*9] on the keyboard, a breakpoint will be set on the selected process or construct. In the C-tree, breakpoints are marked with a symbol in front of the process or the construct its name. Placed breakpoints can be enabled or disabled like other debuggers. A breakpoint which is enabled, is marked by an asterisk (*). A single quotation mark (') is used when a breakpoint is disabled. Figure 4.6 shows an example of two breakpoints, one enabled and the other one disabled.

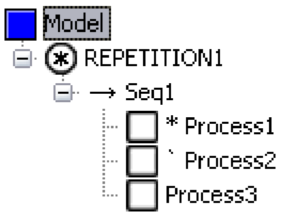


Figure 4.6: An example of two breakpoints, one enabled and one disabled

Both on Process1 and Process2 a breakpoint is set. The breakpoint on Process1 is enabled and the breakpoint on Process2 is disabled. The CT program will be notified by gCSP if a breakpoint is set or removed. The CT program will then set a breakpoint using the breakpoint function of the CT execution framework. Detailed information about the implementation of breakpoints in the animation framework can be found in Appendix B.

Enable or disable animation

Processes or constructs can be selected to be animated or not. If a process or construct is disabled for animation, it will be colored in the default color of the gCSP model during the animation. Default, all the processes and constructs are enabled for animation. Disabling or enabling animation of a complex process, will also disable or enable the animation if its children. This makes it faster for the user to select a process including all its children to be animated or not.

4.2.3 Output windows

The right lower pane of the gCSP user interface is extended with four tabbed windows, next to the existing warning tab (see figure 2.5). They give the user extra information during the animation of a CT program. Table 4.2 describes the windows which are available now.

Table 4.2: Output windows of gCSP

Output Window	**Description**
Warnings	Displays all warnings that occur during the design of a gCSP model. For example, if a process is not grouped (section 2.4.4) in a construct.
BreakPoints	A list of all the breakpoints is shown. Breakpoints can also be configured (enabled / disabled) and removed here.
Execution history	Displays the execution history of processes and construct. Channel communication is also shown here. The data can be saved as a text file on hard-disk, which can be used for later analysis.
Watch	The contents of the channels and the ready-queue of the CT kernel are shown here.
Debug Log	If a debug version of a CT program is created (section 4.2.1) it normally prints debug information to the screen. Now this information will be fetched by gCSP and displayed in this window. This information can than be saved on hard-disk so it can be used for later analysis.

4.2.4 Animation feedback

To animate the execution of a CT program in gCSP, information about its execution is needed. *Animation macros* are added to the CT execution framework, which are capable of generating information about the execution of a CT program. An advantage of using macros is that they all can be enabled or disabled at compile time, using just one define. A release and an animation version of the CT execution framework can be created in a straightforward way.

Different kinds of macros are placed on specific locations in the CT execution framework, like state changes of a process or communication of a channel. For example, if the state of a process changes, a macro notifies it by sending the new state of the process to gCSP. gCSP will fetch this information and use it for the animation of the process in the gCSP diagrams. Appendix B.2 gives a description off all animation macros added to the CT execution framework.

4.3 Animated gCSP diagrams

The animation feedback received from the CT program is used for color based animation in the gCSP diagrams. The next sections will describe the way the gCSP diagrams are animated.

Note: *Because in grayscale the differences between several colors can be hard to see, the state of processes, constructs and channels are displayed as text between brackets near them. In the real case only colors are used!*

Note: *In the next sections the default colors are used. If the user wants to use other colors for the states of processes and channels, they can be configured via the menu of gCSP: Tools -> Options and then go to the animation section (see Appendix C.4).*

4.3.1 Animated processes

Depending on the state of a process, a process is drawn with a different color in the gCSP diagrams. The states which are less important are drawn with a less intensive color. To detect discrepancies between a gCSP model and a CT program, an extra process state is introduced in gCSP, namely the 'undefined' state. For example, if the state of a gCSP process is 'undefined', it means that the process is not initialized by the CT program (see section 4.1). This occurs if there is a mismatch between a gCSP model and a CT program. Table 4.3 shows the colors used for the different states of a gCSP process.

Table 4.3: The colors of an animated gCSP process

Process state	**Color**	**Description**
new	light-grey	initialized correctly by the CT program
ready	light-blue	ready to be executed on the CPU
running	green	currently running on the CPU
blocked	light-grey	blocked while it was running
finished	light-grey	finished its execution
undefined	red	not initialized by the CT program

To be able to see clearly which process is 'running', the colors used for different states should be well chosen. Because the 'blocked' and 'finished' state are used a lot and are less important than the 'running' state, they are colored in light-grey (less intensive than green). This should make it clear to see the 'running' process while animating. Figure 4.7 illustrates the way processes are colored in the gCSP diagrams by using a simple gCSP model. The animation flow is shown by a red arrow between the models.

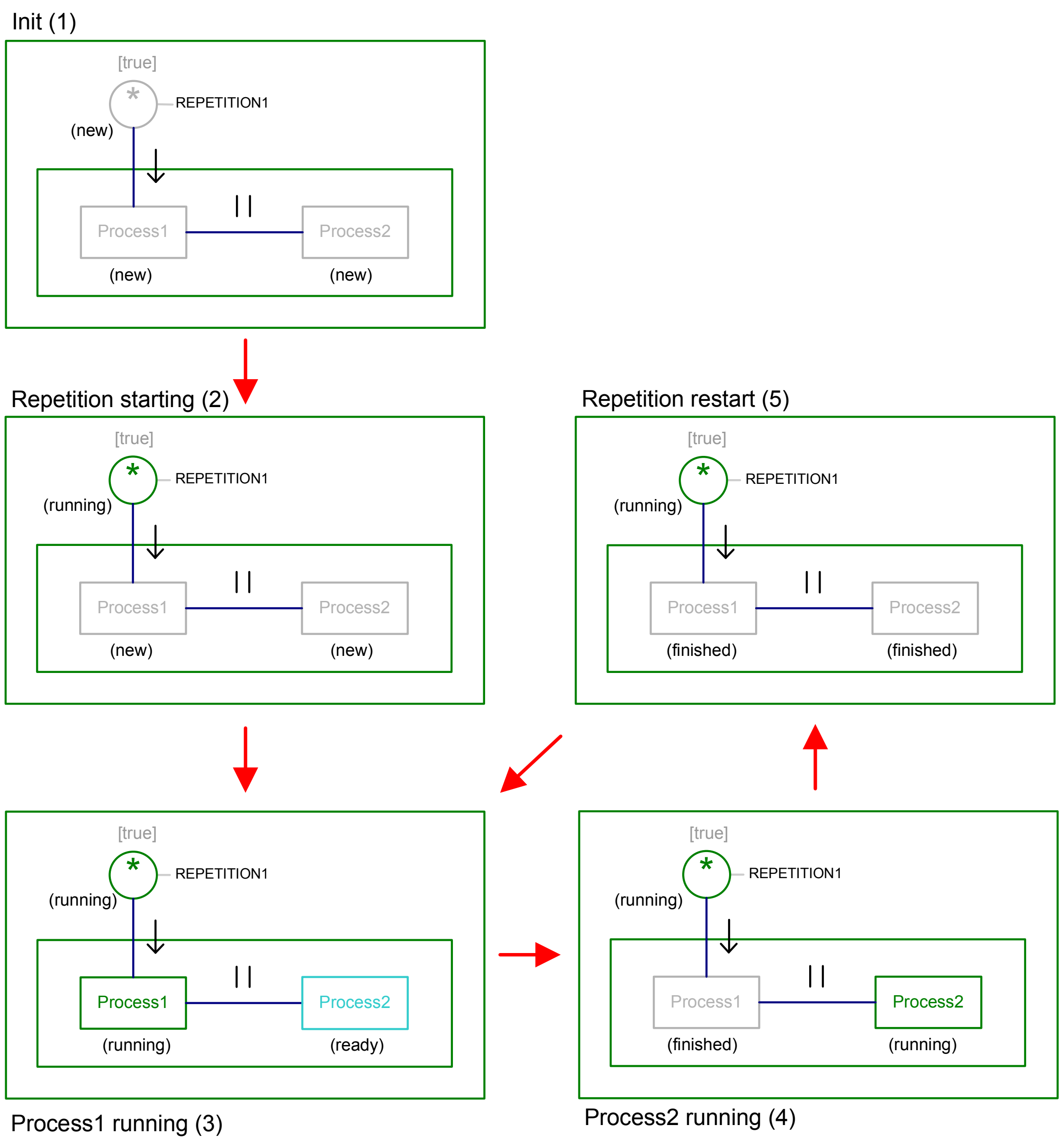


Figure 4.7: Animated processes example

Two processes are executed in parallel and will be repeated forever because of the repetition. If the CT program is started for the first time, the processes will be colored grey (1), which means that they are initialized successfully. The repetition will start (2) the parallel construct, which will execute Process1 (3) and Process2 (4) in parallel. If both processes are finished (5) the parallel construct is also finished with its execution which makes the program go back to its repetition. The repetition restarts the parallel construct and a new iteration is started (5 ⇒ 3).

4.3.2 Animated Compositional relationships

Compositional relationships (the straight lines between the processes) are animated using the same colors as animated processes, because both go through the same states (Appendix A). The main difference is that multiple compositional relationships in gCSP can be grouped into one CT construct (section 2.4.4). For example, if one sequential CT construct containing four processes is activated, it means that three compositional relationships need to be animated. Figure 4.8 shows an example of three processes executing in sequence.

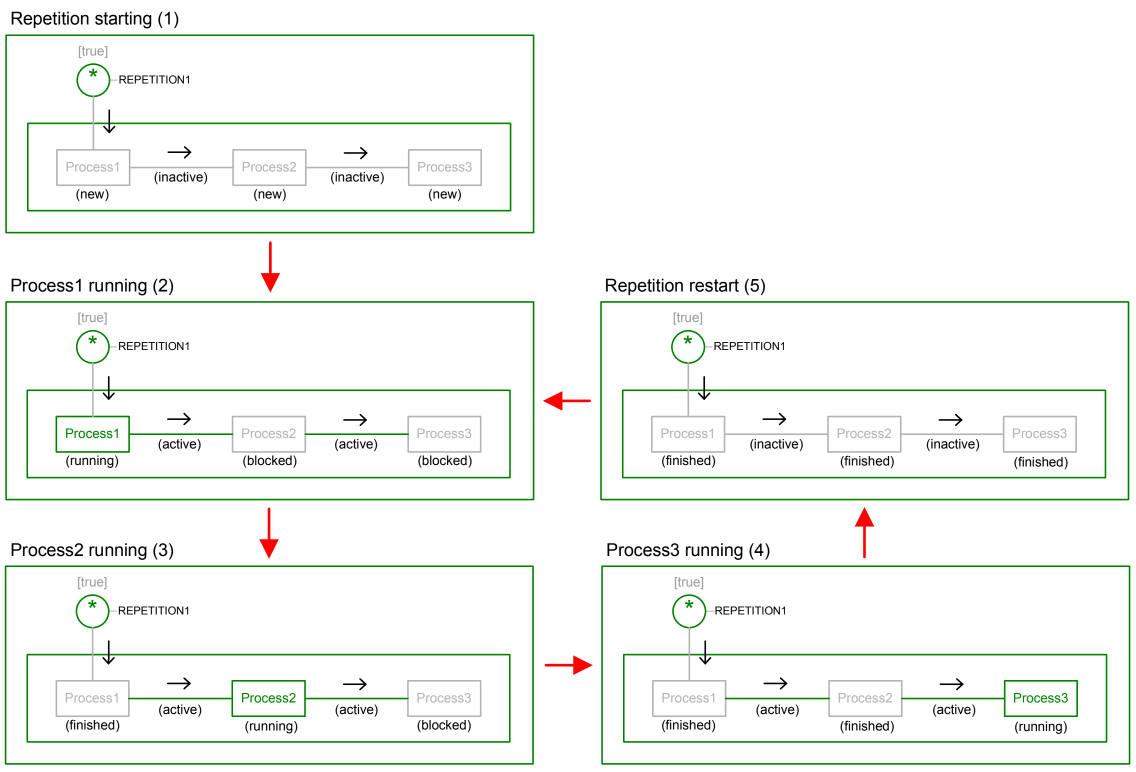


Figure 4.8: Animated compositional relationship example

Process1, Process2 and Process3 are grouped in one sequential construct (see group rectangle), and therefore they are executed in sequence (2 to 4). If the last process (Process3) has finished its execution the sequence construct is finished also and the construct will be deactivated (5). The repetition will restart the sequential construct which will start the processes to be executed in sequence again (5 ⇒ 2). It can be clearly seen that if one of the processes is running (2 to 4) the sequential construct is active. This means that the construct is "running" too, so both sequential compositional relationships will be colored green.

4.3.3 Animated Channels

In gCSP, a channel is drawn as a unidirectional arrow with an arrow shaft and an arrow head. The arrow shaft is defined as the writing-end and the arrow head is defined as the reading-end of the channel. Both ends of a channel can go through two different states (Appendix A.3.2); the state of a channel can be defined in four ways. To detect discrepancies between a gCSP model and a CT program, two extra channel states are introduced in gCSP. A 'new' state if a channel is created for the first time, and an 'undefined' state if the channel is not initialized by the CT program (section 4.1). Therefore, a channel can be colored in six different ways, see table 4.4.

Table 4.4: The states and colors of a channel

Channel state	w-end	r-end	Description
new	grey	grey	Initialized correctly by the CT program.
ready	green	green	Both writing and reading ends may be used.
written	orange	green	The channel has been written, the writing end is set to be 'not ready'. if a process to write on the channel, it will be 'blocked'.
read	green	orange	The channel has been read, the reading send is set to be 'not ready'. if a process tries to read from the channel, it will be 'blocked'.
rendezvous	orange	orange	The channel has been written and read, which means rendezvous can occur.
undefined	red	red	Not initialized by the CT program, because the CT program is out of sync.

If a channel is initialized, both channel-ends will be colored grey. If an error occurs between a gCSP model and a CT program, such that a channel will not be initialized, both ends of the channel will be colored in red. Furthermore, a channel-end which is colored green means that a process (reader or writer) may use the channel-end without being blocked. If a channel-end is colored orange, it means that it is already used by another process. If a process tries to use this channel-end, the process will be blocked until the channel-end is free again.

The contents of a channel
A channel is normally connected (directly or via communication interfaces (section 2.3.1)) between a writer and a reader. To know what data should be written to a channel and where data read from a channel should be stored, both the writer and the reader are connected to a variable. This is illustrated in figure 4.9.

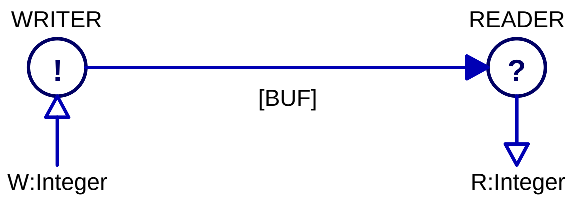


Figure 4.9: Channel connected between a writer and a reader

The value of the variable which is connected to the writer (W) will be written on the channel. The reader will read data from the channel and put it into the variable connected to it (R). In case of a buffered channel, a channel contains a buffer ([BUF]) where data written on the channel can be stored.

To know which data is written to a channel or which data will be read from a channel, it should be possible to obtain the contents of these variables and buffer (in case of a buffered channel). While animating a CT program the mouse cursor can be hovered over a channel, which will show the contents in a tooltip window near the channel. The text shown in the tooltip window is formatted as: "**W [BUF] R**". The 'W' represents the data which is ready to be written on the channel. In case of a buffered channel, '[Buf]' represents the contents of the channel buffer. The 'R' represents the data ready to be read from the channel which than can be stored in the variable connected to the reader.

Example of animated channels

Figure 4.10 shows an example of a animated non-buffered rendezvous channel used in a producer-consumer gCSP model.

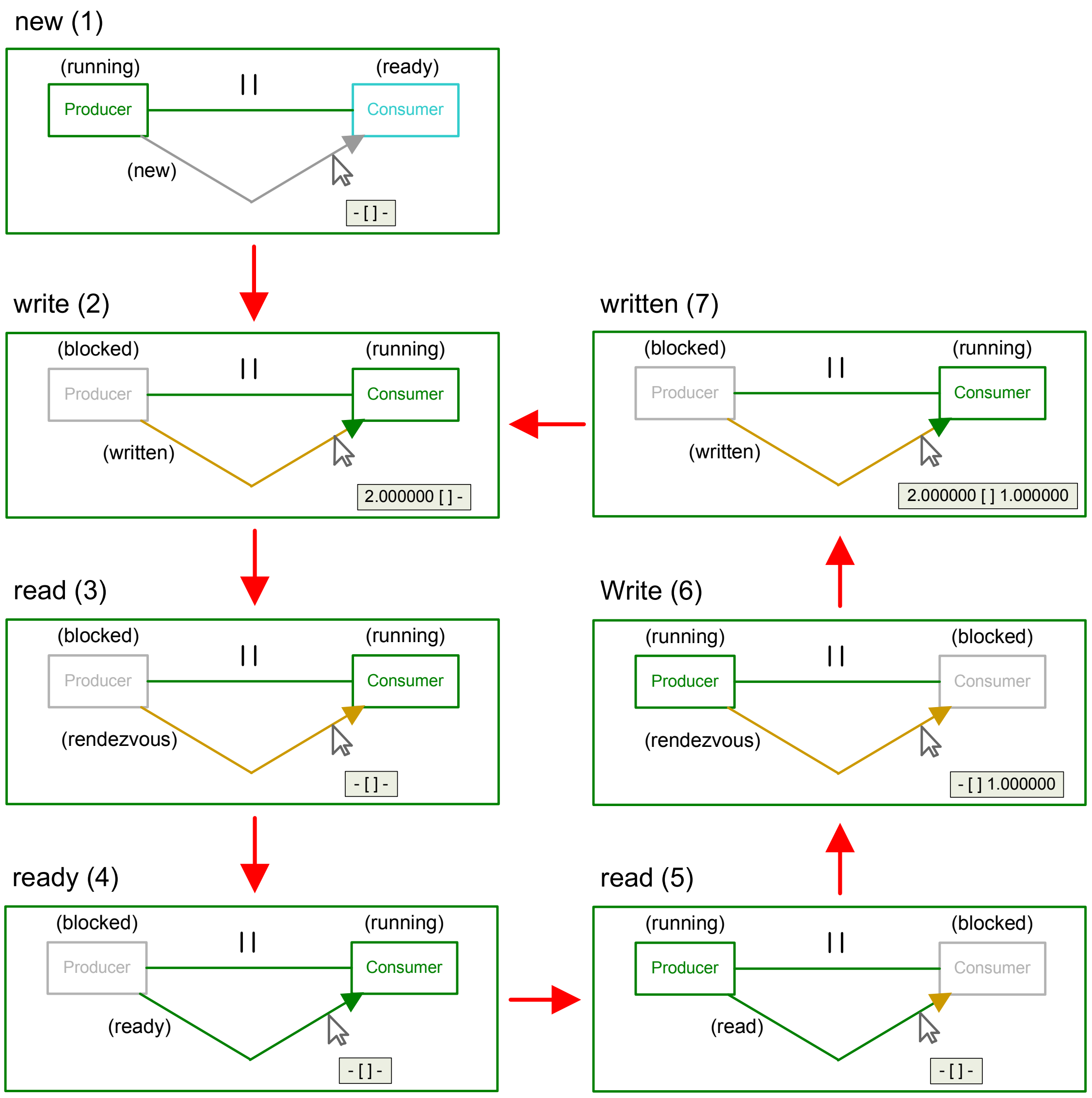
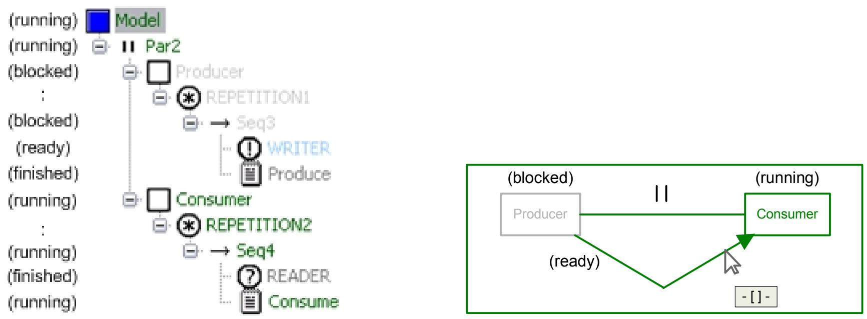


Figure 4.10: Animated channels example

Before the Producer can write to the channel, the channel is initialized (1). Next the Producer tries to write data on the channel, but because the Consumer is not ready yet, the data will be placed in 'W' (2). The state of the channel is changed 'written', the Producer is blocked and the Consumer may run. The Consumer will then read the data from the channel and remove it from 'W' again (3). Rendezvous occurred and both sides are freed again, which results that the channel will be set to be 'ready' (4). Because the Consumer is not blocked, it tries to read another time from the channel. The channel will set to be 'read' (5) and the Consumer will be blocked because there is no data available. Next the Producer may write twice. One so that rendezvous occurs (6) and one to set data ready to be written (7). The Consumer then may read this data (2) and a new iteration will be started again.

4.4 Animated C-tree

To be able to show the state of child processes without the user letting to look directly into their parent process, animation is used in the C-tree. The C-tree is animated just like the processes and the constructs. This creates the possibility to show the state of every process and construct in one view. An example of an animated C-tree is given in figure 4.11.



(a) Animated C-tree (b) Model of the C-tree

Figure 4.11: Animation of the C-tree

In figure 4.11(b) it can only be seen that the Consumer process is 'running'. It can not be seen in this view, which child process of the Consumer is running. Figure 4.11(a) represents the animated C-tree of figure 4.11(b). The state of the child processes ('Reader' and 'Consume') are shown in the C-tree. The Reader process is 'finished' and the Consume process is 'running'. By using animation in the C-tree, the state of all the processes and constructs in the gCSP model are shown in one view.

4.5 Conclusions

Both gCSP and the CT execution framework are extended to create an animation framework for animating the execution of a CT program in gCSP. The specifications of section 3.7 are used as a guideline to accomplish this. A CT program is capable to share information about its execution with gCSP. gCSP uses this information to create color- and textual- based animation in the gCSP diagrams. The CT execution framework is extended with some debug functionality, which can be used by the user from within gCSP. This creates a user friendly way of debugging or analyzing a CT program while it is executing. All specifications have been met.

To be able to conclude that animation indeed improves the feedback of an executing CT program and that it is useful while developing CSP models some tests needed to be done. The next chapter describes the results of some tests and tries to demonstrate the advantages to be able to animate the execution of a CT program in gCSP.

5 Testing and discussion

This chapter presents the results obtained from the tests performed during this project. Next, the results of these tests are discussed and a conclusion about them is given.

5.1 Testing

Tests are performed on the:
1 pratical use of animation, to validate the correctness of the animation.
2 execution behavior of gCSP models, to test and validate their behavior by using animation.

While creating models to be tested on these two points, for certain models, gCSP failed to generate correct C++ code. The C++ code generator of gCSP is tested, to find the cause of this.

5.1.1 Practical use of animation

All the parts of the CT execution framework (section 3.4) are tested on their animation, by using small gCSP models. First processes, constructs etc., are tested separately, followed by combining them in different ways. Combinations like, processes executing in parallel, multiple processes using shared channels; and complex processes containing multiple children. Also more complex models, like the production cell ((Maljaars, 2006)) and a model of the "‘philosopher problem" (see figure 2.5) are tested on their animation.

Another way the practical use of animation is tested is to experience with some functionality of the CT execution framework. The *yield()* function and the two different schedulers of the CT kernel (compositional and occam) are tested. The actual context switching causes by the yield() function and the context switch methods of the two different schedulers, can now be seen by animating the execution of a CT program.

5.1.2 Execution behavior

Certain constructions in the gCSP models caused undesired execution behavior of their generated CT programs. To find the cause of these behaviors, small gCSP models were created containing these constructions. They were analyzed by animating the execution of their generated CT programs. The following constructions are analyzed:
1 processes communicating via rendezvous channels.
2 multiple processes communicating via shared or joined channels.
3 processes executing in parallel.

Processes communicating via rendezvous channels

The producer-consumer model of figure 4.10 was used. Each repetition cycle, the channel was written and read twice in stead of one. While using animation for searching the cause of this behavior, it was noticed that the producer and the consumer was not always blocked directly after using the channel. This behavior is caused by an optimization in the CT execution framework to decrease the amount of *context switches* (Hilderink, 2005, p203-205). If rendezvous on a channel occur, the process which is using the channel may continue its execution.

Processes communicating via shared or joined channels

The Fair policy first-come, first-served (FCFS), is used for shared and joined channels. The process which first tries to write or read the channel may use it first. Other processes need to wait until the channel is free again. During the animation of processes communicating via shared or joined channels, it is observed that only the first process was using the channel without letting the other processes through. An error is found in the CT execution framework on the implementation of shared and joined channels. This error was fixed during this project, which resulted in the correct behavior of shared and joined channels.

processes executing in parallel
The gCSP model of figure 4.7 was used. Each repetition cycle, the first process (Process1) of the parallel construct was executed first. The processes are running in parallel, so they are given the same priority. It should not matter which parallel process executes first. An optimization in the scheduler of the CT execution framework causes this static execution order. If the execution order does not matter, why not always execute the first process of a parallel construct first?

5.1.3 C++ code generator of gCSP

Certain structures which can be drawn in gCSP, fails to generate C++ code. This gave a few unintended limitations, which resulted that not every gCSP model could be tested. By testing the C++ code generation with different gCSP model, the following limitations were found:

1 A shared or joined channel can not be used by its parent via their (ports).
2 A channel may only be shared or joined by using just one splitter.
3 A var-channel can not go through the ports of a process.

Figure 5.1 illustrates these structures.

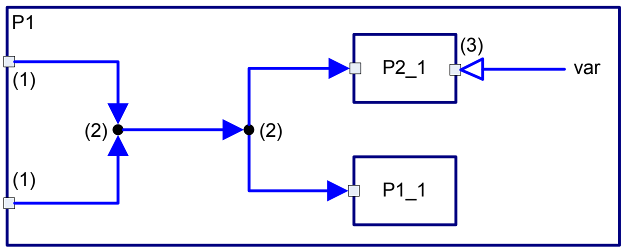


Figure 5.1: gCSP code generation failures

5.2 Discussion

5.2.1 Practical use of animation

Animation is used in the MSc course Real-Time Software Development, which was started at the end of this project. In comparison with previous years, the students had fewer questions about CSP and the way their CT programs were executing. In my opinion, animation is an effective way of giving feedback on the execution of a CT program. It gave more insight in the execution behavior of a CT program, because it is graphically shown now. This resulted in better understanding of the CSP concepts and the way processes are communicating with each other by using rendezvous channels.

5.2.2 Optimizations in the CT execution framework

The optimizations in the CT execution framework were implemented for practical purposes. It is a point of discussion if it should be able to set these off, because the CT execution framework is also used for education purposes, like in the MSc course Real-Time Software Development.

5.2.3 C++ code generator of gCSP

The C++ code generator of gCSP should be capable to generate correct C++ code of all the structures drawn in gCSP. Should these structures be disabled for drawing? Or should the user be notices that the C++ code generator fails if one of these structures is drawn?

5.3 Conclusions

From the previous sections it has become clear that animation is useful. Animation can be used to analyze the execution behavior of a CT program, and it can be used to find the cause of undesired execution behavior. Without animation it would be probably harder, to "debug" and analyze the execution behavior of a CT program.

6 Conclusions and Recommendations

6.1 Conclusions

The goal of the project is to design and implement supporting tools to improve the feedback on the execution of a CT program. The use of animation was selected as an appropriate way to do this. To achieve this goal, an animation framework has been created by extending gCSP and the CT execution framework for animation purposes. The diagrams of gCSP are used to create animation in a graphical way. The execution framework generates feedback on the execution of a CT program, which is used by gCSP for its animation.

Analysis was performed on the way the animation framework should be designed and implemented. A list of specifications was created and was used as a guideline to design and implement the animation framework. The animation framework makes the following possible:

- A CT program can share information about its execution with gCSP.
- gCSP uses this information for color and textual based animation in the gCSP diagrams.
- a CT program can be debugged and analyzed during its execution.
- The execution history of a CT program can be saved to hard-disk.

To demonstrate the advantages of the animation framework for analyzing the execution behavior of a CT program, several gCSP models were tested and analyzed with the use of animation. From these tests it can be concluded that animation:

- is appropriate for giving feedback on the execution of a CT program.
- can be used for validating the execution behavior of CT program.
- can be used to find the cause of undesired execution behavior of a CT program.

Animation is successful for testing, validating and analyzing the execution behavior of a CT program. It was demonstrated that animation gives better insight in the execution behavior of a CT program, which concludes that the goal of this project is achieved.

6.2 Recommendations

It is recommended that the internal database of gCSP will be redesigned. During implementation of the animation framework in gCSP, a lot of preparation world was needed to get familiar with gCSP. The documentation about the implementation of gCSP is bad or even not available. Also the internal database of gCSP is designed badly and therefore is hard to use. Because of the poor database design and the bad documentation, new extensions are hard to implement and bugs are hard to solve. If the CE-department continues with gCSP and wants to extend it more in the future, it is recommended that the internal database of gCSP will be redesigned. Because gCSP depends a lot on this database, this results in redesigning gCSP. Re-use the good parts and redesign the bad parts, for getting a better and more stable gCSP.

During the tests performed in chapter 5, some disadvantages were observed in the CT execution framework and gCSP. To solve these problems future work is needed:

- Let the user choose to disable or enable the context switch optimization, to give more insight in context switch methods of the CT execution framework.
- The implementation of parallel CT constructs which starts its processes in random order.
- Let the user choose which scheduler of the CT kernel should be used.
- Improve the C++ code generator of gCSP.

Improvements for the animation framework are:

- The simulation of a previous executed CT program by using the history of its execution.
- Real-time logger in the CT execution framework for off-line analysis (logged data can be used as input data for the simulator).

A The CT execution framework

A.1 CT processes

A.1.1 Process types

Table A.1: Process types

Process state	Description
Process	object performing a specific task, which is defined by its children
Writer	reads the value from a variable and writes it on a channel
Reader	reads data from a channel and put the value into a variable
Recursion	repeats its child processes if they are all finished
Code	special process with user defined source code

A.1.2 Process states

A process can be in one of the states given in table A.2

Table A.2: The state of a process

Process state	Description
new	if a process is created it will start in this state
ready	a process is ready to run on the CPU
running	the process is running on the CPU
blocked	the process is blocked, probably because a source is not available.
finished	the process is finished with its task and set to idle
terminated	the process is removed from the memory

Changing the state of a process is mostly done by an external resource. For example a scheduler can decide that a process may run by setting its state to 'running' and gives the CPU to the process. Although a process is in one state at the time, there are some connections between them. For example, a process can not be blocked if it was not running before. The way the state of a process changes, is illustrated with a state-diagram shown in figure A.1.

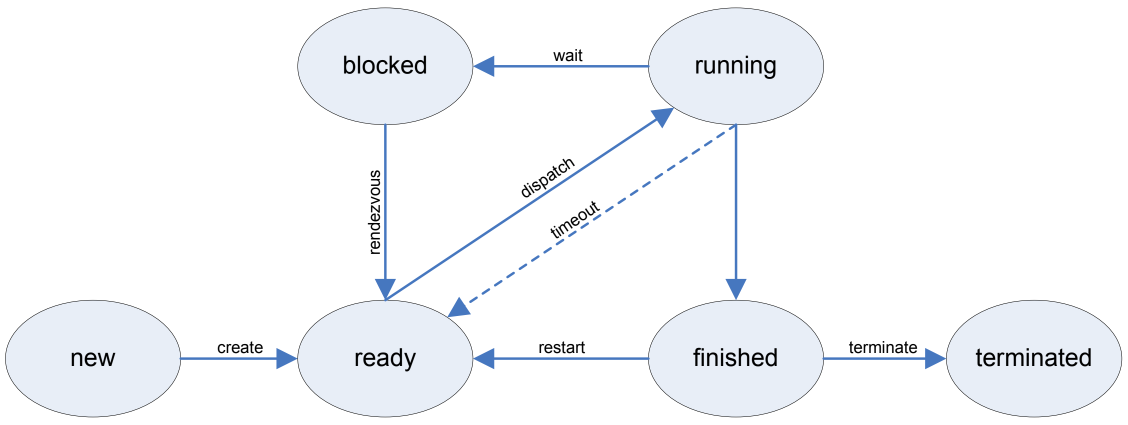


Figure A.1: The various states of a CT process displayed in a state diagram

If a process is created ('new'), it needs to wait for the scheduler to set its status to 'ready' and load it into the main memory. Once the process has been assigned to the CPU, a context switch is performed and the process state is set to 'running'. If a process needs to wait for a source (data-channel) to get ready, it is moved into the 'blocked' state until no longer waiting is needed, and then moved back into the 'ready' state. Once the process finished execution it is moved to the 'finished' state. If a process is terminated the state is moved to the 'terminated' state and the process is removed form the main memory.

A.2 CT constructs

A.2.1 Construct types

Table A.3: Constructs used by compositional relationships in gCSP

Symbol	Construct	Description
$\longrightarrow$	Sequential (SEQ)	The processes will be executed sequentially in the order the arrow indicates.
$\|$	Parallel (PAR)	The processes are executed in parallel (concurrently). Each process has an equal chance (priority) to execute.
$\square$	Alternative (ALT)	Works on guards instead of processes. Each guard has a process associated with it. If a guard is ready the associated process will run. If multiple guards are ready at the same time only one process may run.
$\overleftarrow{\|}$	PriParallel (PRIPAR)	A prioritized version of the parallel. The process the arrow is pointing at is given a higher priority.
$\overleftarrow{\square}$	PriAlternative (PRIALT)	A prioritized version of the alternative. The guard the arrow is pointing at is given a higher priority.
$\overrightarrow{\triangle}$	Exception (EXCEPTION)	Handle exceptions thrown in another process. The arrow is pointing to the exception handling process.

A.2.2 Construct states

Constructs are special processes, which go through the same states as normal processes (section A.1.2). If two processes are created in a sequential construct and one of the two processes is running, it automatically means that the sequential construct is running too.

A.3 CT channels

A.3.1 Channel types

Table A.4: Channel types

Channel Type	Description
One2One	single writer to single readers.
Any2One	extension of the One2One channel to support multiple writers
One2Any	extension of the One2One channel to support multiple readers
Any2Any	extension of the One2One channel to support multiple writers and readers
Buffered	extension of the channels above, which is capable of buffering data

Buffered channels

A buffered channel can be configured in three ways which are given in Table A.5.

Table A.5: Buffered Channel Properties

Property	**Description**
Buffer-size	the amount of data to be buffered
Overwrite	if true, the oldest data will be overwritten if the buffer is full, else the writer will be blocked and wait till there is free space in the buffer again.
Non-blocking read	if true, always return from its reading action ('null' will be returned if no data is available), else the reader will be blocked if the buffer is empty

A.3.2 Channel states

A channel is divided into two channel ends, a reading and a writing end. Both ends can go through two states which are given in A.6.

Table A.6: Channel States

Channel-end state	**Description**
ready	channel-end is ready to be used by a process
not-ready	channel-end is already in use, so a process which tries to use it will be blocked

If the process is using a channel, the state of the channel-ends will determine if the process may continue or will be blocked. If the writing-end of a channel is already used by another process, the process will be blocked. If a channel-end is free (ready), the process may use it and depending on the channel type (rendezvous, buffered), the process may continue or will be blocked.

Channel communication

The ends of a channel are normally connected between a writer and a reader process which is illustrated in figure A.2

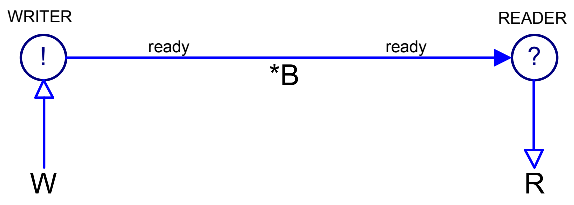


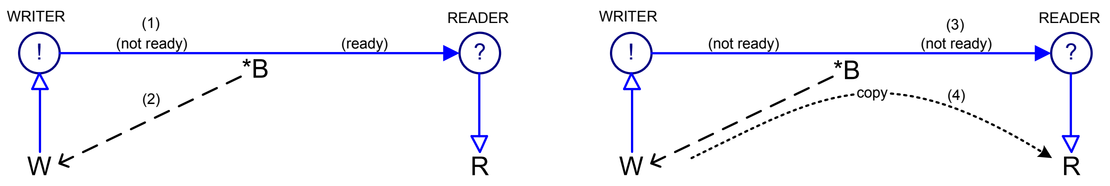
Figure A.2: Channel connected between a writer and a reader

The value of 'W' will be used to write on a channel, 'R' will be used to store the data read from a channel. '*B' is the pointer of a channel, which can point to 'W' or 'R'.

In case of a rendezvous channel, it can be used in 2 different ways:

- the writer will write data to a channel before a reader has read it
- the reader tries to read a channel before a writer has written data on it

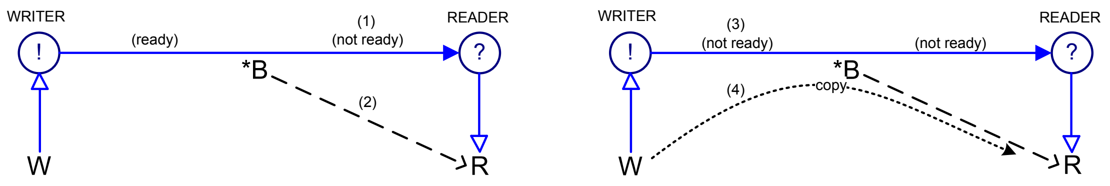
The following two figures, figure A.3 and figure A.4, illustrate what happens with the data internally.



(a) Point to writer variable (b) copy buffer

Figure A.3: Writer comes first

In case the writer comes first (figure A.3(a)), the writing channel-end will be set to ’not ready’ (1) and ’*B’ will be pointed to ’W’ (2). Next, the writer will be blocked. If the reader tries to read the channel (figure A.3(b)), the reading channel-end will be set to ’not ready’ (3) and the data pointed by ’*B’ will be copied to ’R’ (4). After the data has been copied, the writer will be unblocked and both channel-ends will be set to be ’ready’ again (figure A.2).



(a) Point to writer variable (b) copy buffer

Figure A.4: Reader comes first

In case the reader comes first (figure A.4(a)), the reading channel-end will be set to ’not ready’ (1) and ’*B’ will be pointed to ’R’ (2). Next, the reader will be blocked. If the writer writes on the channel (figure A.4(b)), the writing channel-end will be set to ’not ready’ (3) and the data of ’W’ will be copied to the variable where ’*B’ is pointing to (4). After the data has been copied, the reader will be unblocked and both channel-ends will be set to be ’ready’ again (figure A.2).

In case of a buffered channel, ’*B’ is a buffer with a predefined size. Written data will be stored in this buffer. If a reader reads from a buffered channel, the first written data will be read and will be removed from the buffer.

B The animation framework

B.1 The CT execution framework

B.1.1 Animation framework classes in the CT execution framework

The CT execution framework is extended with four classes which are added to the directory '/ctcpp/anim'. The following sub-sections describe these classes shortly.

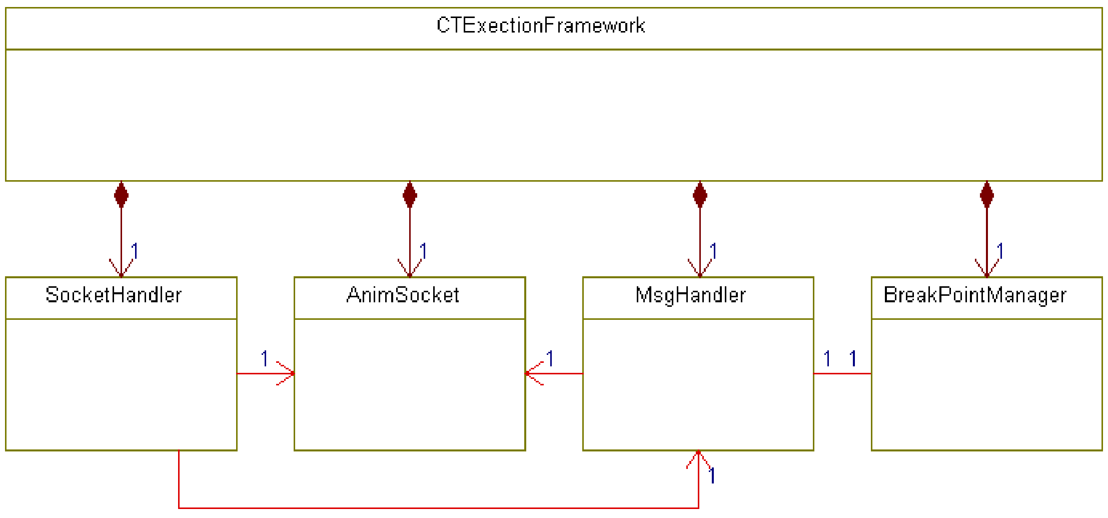


Figure B.1: UML diagram about the extension of the CT execution framework

Socket Handler (SocketHandler.h, SocketHandler.cpp)
A native thread polling the animation socket server (section B.1.1) for incoming messages. If a message is received, it is passed to the message handler for encoding. If the connection to gCSP is lost, the socket handler will terminate the CT program.

Animation socket server (AnimSocket.h, AnimSocket.cpp)
A TCP / IP socket server, which is used to communicate with gCSP to write and read animation messages (section B.4).

Message handler (MsgHandler.h, MsgHandler.cpp)
The message handler decodes and encodes animation messages. If a message is valid and decoded correctly, action will be taken depending on the type of the message (section B.4). If a message needs to be send to gCSP, the encode functions can be used which is able to encode a message according the format given in section B.4. If the message is encoded correctly, it will be send to gCSP by using the animation socket server.

Breakpoint manager (BreakPointManager.h, BreakPointManager.cpp)
The breakpoint manager contains the list of all breakpoints set by gCSP. The breakpoint manager is capable to check if a breakpoint is placed on a process or construct and pause the CT programs execution if needed (breakpoint is set).

If the state of a process changes, the function checkBreakPoint() of the breakpoint manager is called to see if a breakpoint is set on the process. If a breakpoint is found, the program will be pause its execution. This function is implemented as given in listing B.1.

Listing B.1: Implementation of the checkBreakPoint function

```
1  void BreakPointManager::checkBreakPoint(void* brk, int state)
2  {
3          if(stepper && state == RUNNING){
4                  breakPointSem->p(); //get the semaphore
5          }
6          else if(brk == NULL){
7                  msgHandler->NotifyBreakPoint(brk, state);
8                  breakPointSem->p(); //get the semaphore
9          }
10         else{
11                 breakPoint = breakPoints->find(brk);
12                 if(breakPoint != breakPoints->end()){
13                         states = breakPoint->second;
14                         st = states->find(state);
15                         if(st != states->end()){
16                                 msgHandler->NotifyBreakPoint(brk, state);
17                                 breakPointSem->p(); //get the semaphore
18                         }
19                 }
20         }
21 }
```

The CT program will be paused if one of the following is true:

- The stepper is active and the process is in its ’running’ state.
- ’brk’ is a ’null’ address.
- A breakpoint is found in the breakpoint-list.

To pause the CT program, this function uses a native Semaphore (line 4, 8, 17). Because this semaphore is always taken, taking this semaphore with the function p(), will pause the program. If the semaphore is released again, because for example gCSP wants to continue, the CT program will continue its execution again.

B.2 Animation macros (AnimMacros.h)

To create animation messages to send them to gCSP for animation purposes, the CT execution framework is extended with some animation macros. To enable the animation macros, *CT_CONFIG_ANIM* in config.h should be defined as ’1’. Table B.1 gives the list of all the animation macros added to the CT execution framework. The words in the description printed in bold, are used as arguments in the macros.

Table B.1: The animation macros of the CT execution framework

Functions	**Description**
ANIM_NEW	create objects needed for animation
ANIM_INIT	initialize the objects needed for animation
ANIM_DELETE	delete the animation objects
ANIM_SET_BREAKPOINT	set a breakpoint on **address** with the **state**
ANIM_REM_BREAKPOINT	remove a breakpoint from **address**
ANIM_CHECK_BREAKPOINT	check if a breakpoint with the **state** is set on **address** - if a breakpoint is found the program will paused
ANIM_SET_ANIM_PROCESS	**enable/disable** animation of a **process**
ANIM_GET_ANIM_PROCESS	check if a **process** is animated - return **true** if animated is enabled
ANIM_NOTIFY_ADDRESS	notify the **address** of the this pointer of an object and the **name** of the object
ANIM_NOTIFY_PROCESS_STATE	notify the **state** of a **process**
ANIM_NOTIFY_CHANNEL_WRITE	notifies that a writer has written **data** to a **channel**
ANIM_NOTIFY_CHANNEL_READ	notifies that a process has read **data** from a **channel**
ANIM_NOTIFY_CHANNEL_STATE	notifies both **state**s of the **channel**-ends
ANIM_WAIT	wait with the execution of the CT program till a next step is set
ANIM_SEND_LOG_MESSAGE	send a **log message**
ANIM_NOTIFY_READY_QUEUE	notify the **contents of the ready-queue**

B.2.1 Using the animation macros

To use the animation macros, they need to be placed on specific locations in the source code of the CT execution framework. Two examples are given below which illustrate the way these macros can be used.

1 A small section of the source code of a process is shown.

Process.cpp

```
1 if(state!=ABORT_SIGNALLED)
2 {
3         ANIM_NOTIFY_PROCESS_STATE(this, RUNNING);
4         state = RUNNING;
5         run();
6         ANIM_NOTIFY_PROCESS_STATE(this, FINISHED);
7         state = FINISHED;
8 }
```

Right before the state of a process changes (line 4, 7), an animation macros is added (line 3, 6). If one of these macros is executed, it will send an animation messages to gCSP about the state ('running' and 'finished') change of the process (this pointer).

2 A small section of the source code of a one-to-one channel is shown.

One2OneChannel.cpp

```
bReaderReady = true;
ANIM_NOTIFY_CHANNEL_STATE((ChannelImpl*)this, CHANNEL_READY,
CHANNEL_NOT_READY);
```

After the state of a channel changes (line 1), an animation macro is added (line 2, 3). If these macros are executed, an animation message is send to gCSP about the state change of the channel. The first argument is the 'this pointer' of the current channel. The second argument is the state ('ready') of the writing channel-end. The third argument is the state ('not ready') of the reading channel-end.

B.3 The gCSP tool

B.3.1 Animated objects

In gCSP, processes, channels, compositional relationships etc. are created as objects. To use these objects for animation in the animation framework, three new variables were added to their class files. These variables are described in table B.2.

Table B.2: The animation variables of the gCSP objects

Name	**Type**	**Description**
animated	Boolean	determines if the object is selected to be animated
animColor	Color	the color in which the object should be drawn if the animation is running
animState	Integer	the state a object is in, for example 'running' or 'blocked'

For example, if the animation is running and a process object needs to be drawn, color given by 'animColor' if 'animated' is set to true is used. If 'animated' is set to false, the process will be drawn in its default color.

B.3.2 Animation framework classes in gCSP

gCSP is extended with nine classes, which are placed in a debug package (gml.debug) of the gCSP project. The following sub-sections describe each class shortly.

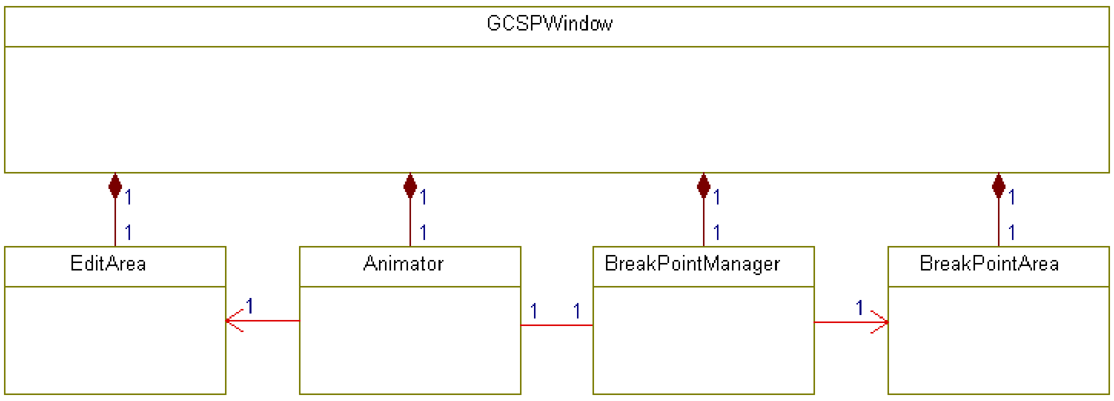


Figure B.2: UML diagram of the top level of gCSP

Edit area (EditArea.java)
Data can be written to this window, which is enabled to be saved on hard-disk. This class is used by the tracing and logging windows in gCSP.

Animator (Animator.java)
The animator is the main class of the animation framework. It contains the functions to start and stop the animation and to animate all the objects of the gCSP diagrams (process, channels etc.).

Breakpoint manager (BreakPointManager.java)
Manages the breakpoints set in gCSP. It contains a list of all the breakpoints which are set. The breakpoint manager provides the functions to set / remove breakpoints and to check if a breakpoint is set.

Breakpoint area (BreakPointArea.java)
This window displays a list of the breakpoints. The breakpoints can be configured to be enabled/disabled or to be removed.

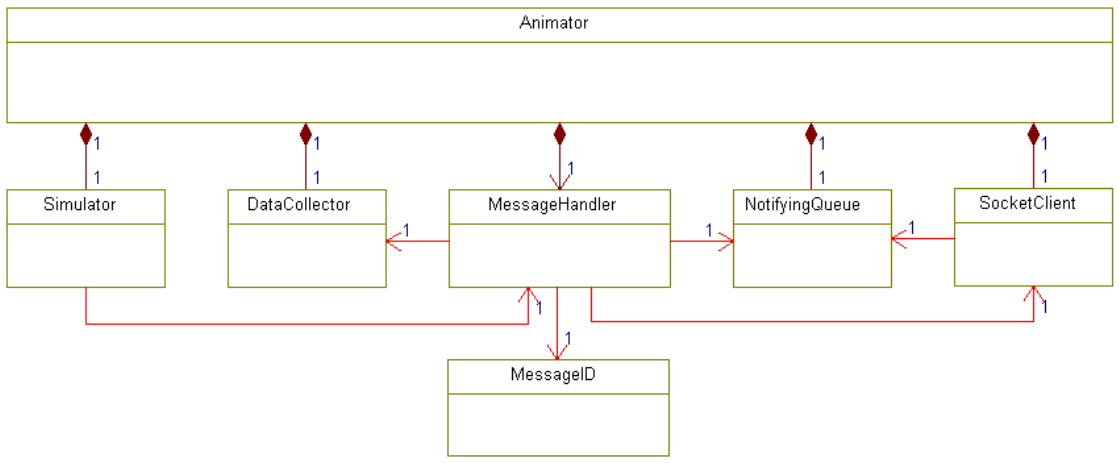


Figure B.3: UML diagram of the Animator

Simulator (Simulator.java)
This class provides the functionality for automatic stepping. The step time is defined in milliseconds.

Data collector (DataCollector.java)
The data collector collects all the data / references which are used by the animator. It collects process, contents of channels, addresses of processes within the CT execution framework etc.

Message handler (MessageHandler.java)
This class provides the encoding and decoding of animation messages, and take action depending on the type of animation message (messageID). It polls the MessageQueue for new incoming messages and uses the socketClient to send command messages to a CT program.

Message identities (MessageID.java)
Contains a list of all identities (IDs) used to encode or decode animation messages.

Notifying queue (NotifyingQueue.java)
A message queue to enable event based receiving of animation messages. If a java thread uses the notifying queue and tries to get a message from it, it will be blocked until a message is available.

Socket client (SocketClient.java)
This class provides a socket client to communicate with a CT program.

B.4 Animation messages

Animation messages are devided into three fields which are illustrated in figure B.4 and described in table B.3.

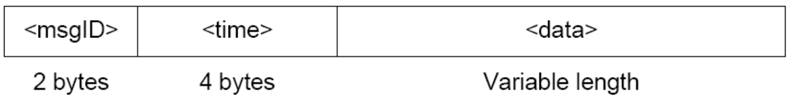


Figure B.4: Animation message

Table B.3: The fields of an animation message

Field	Description
msgID	1 = process state has been changed 2 = channel state has been changed 3 = channel is written 4 = channel is read 5 = the contents ready-queue has been changed 6 = breakpoint is found 7 = initialization of an address and name 8 = message 9 = log massage 10 = a breakpoint is set 11 = breakpoint is removed 12 = enable / disable animation of a process 13 = enable / disable animation of a channel
time	the time of the send message since system startup in micro-seconds formatted as a 32 bits number
data	comma separated string which contains the data belonging to the type of message (msgID)

Each field consists of ASCII formatted data, where each field is separated by a comma. For example: "7,1232132,0XBD12EDFF,Process1". This means that after 1232132 micro-seconds since system startup, a process object, named 'Process1', is created on address '0XBD12EDFF'.

C User manual of the animation framework

C.1 Code generation

Different kind of code can be generated from a gCSP model, using the code generation toolbar (marked by a red ellipse) shown in figure C.1.

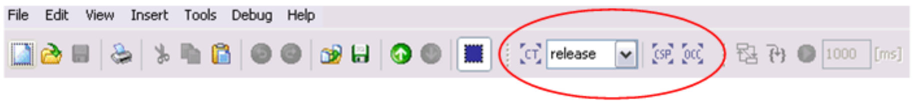


Figure C.1: Code generation toolbar of gCSP

Table C.1: Code generation

code	Description
C++ (CT)	generate C++ code, usable for creating an executable
CSPm (CSP)	generate CSPm code, which can be used by FDR to check the gCSP model on deadlocks or livelocks
Occam (OCC)	generate occam code

Note: *Generated code will be saved on hard-disk in a sub-directory with the same name as the gCSP model.*

To build a CT program ((**C++ code**) which is used for different purposes, different kind of builds can be chosen by selecting it in the dropdown box near the C++ code generation button. Table C.2 describes the build options.

Table C.2: Build option

build option	Description
release	create a CT program which will finally run on the target
debug	create a CT program with debugging support. Textual debug information will be printed to the screen.
animation	create a CT program usable by the animation framework.

C.2 Compiling in gCSP

gCSP is to compile and run an executable automatically after C++ code is generated. This option can be accessed via the menu *Tools->Options* and then go to the third tab of the code generation section.

Note: *These options are still in development, so they probably will not work 100% correctly.*

Generate shell script

This will generate a bash script for Linux and windows to compile the generated sources. The script will get the name compile.sh or compile.bat. To get a properly working script, at least the directory (CT path) to the CT-library has to be filled in.

Name of the executable

An executable will be created with the name given in this field.

Postprocess command
This option can be used under Linux as well as under windows, to compile the generated C++ code automatically. The following needs to be filled in:

- Windows older than vista: *cmd /c compile.bat*
- Linux: *./compile.sh*

Run the executable after post process command
The executable will be started, after the compiler has been finished.

C.3 Use animation in gCSP

C.3.1 Start the animation

Before the animation can be started, the CT program needs to be started first. The CT program will then wait till gCSP tries to connect to it. To start the animation the following button should be pressed. **Note:** *Animation can only be used if the build option (section C.1) is set to animation.*

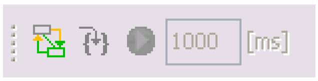


Figure C.2: Animation toolbar of gCSP

If the gCSP diagram changes color, the animator has successfully started. If no colors were changed and no error message has occurred you need to select the model part you want to enable for animation (see section C.3.2).

C.3.2 Select the part of the model (not) to be animated

In the Compositional tree (C-tree) at the left of the screen a selection can be made of which part of the model needs to be animated. This can easily be selected by clicking the right mouse button on the CTree or on a process and enable animate from the drop down menu (see figure C.3).

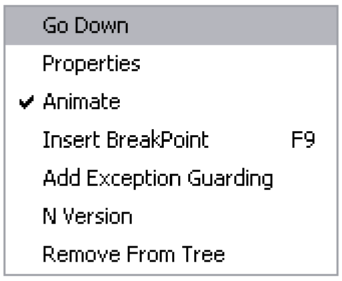


Figure C.3: Dropdown menu of gCSP

All underlying child processes will then be selected to be animated as well. **By default all processes are selected to be animated, but can be set off by selection as mentioned above.**

C.3.3 Setting breakpoints

A breakpoint can be set on any kind of process or construct, by using the right mouse menu of a process or construct (see figure C.3. In the C-tree, breakpoints are marked with a extra symbol in front of the process or the construct its name. Breakpoints are marked in two different ways, with an asterisk (*) or a single quotation mark ('). An asterisk means that the breakpoint is enabled and a single quotation mark means that the breakpoint is disabled. Figure C.4 shows an example of two breakpoints, where one is enabled and the other one is disabled.

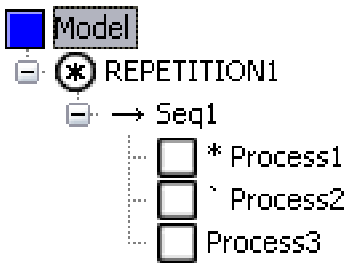


Figure C.4: An example of two breakpoints, one enabled and one disabled

The breakpoint on Process1 is enabled and the breakpoint on Process2 is disabled.

C.3.4 Start or continue the animation

If the gCSP is successfully connected to the CT program, using [F5] will start the CT program to be animated. If your model is paused at a breakpoint it can be continued with [F5] again.

C.3.5 Stepping through the animation

To step to the next current running process without setting a breakpoint on each process, a stepper is available. By pushing the step button from the button bar of gCSP or the [F11] button of the keyboard, a step is set to the next current running process.

To prevent pushing the [F11] button all the time, a period timer can be set which will enable to step automatically. The time between steps is set in milliseconds and is by default set to 1000 milliseconds (1 second). By pushing the play button the periodic stepper will be activated and the button changes to . If this button is pushed again the periodic stepper will be stopped.

A periodic time of larger than 100 ms is preferred.

C.3.6 Stop the animation

To stop the animation the same button used to start (section C.3.1) the animation can be used. If a CT program will be killed, the animation will also be stopped.

C.4 Animation options

This section deals with the adaptable options of the animation. The option window for the animation, as shown in figure C.5, can be accessed via the menu *Tools* -> *Options* and then go to the animation section.

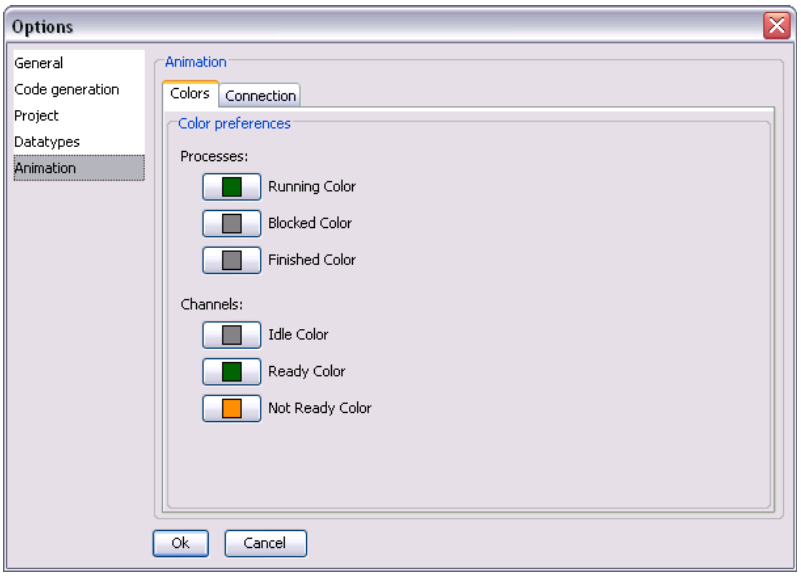


Figure C.5: Animation options

C.4.1 Colors

The colors used for the animation can be selected here.

Processes

Select the colors used for the different states of a process:

- running (default: green)
- blocked (default: grey)
- finished (default: grey)

Channels

Select the colors used for the different states of a channel

- idle (default: grey)
- ready (default: green)
- not ready (default: orange)

C.4.2 Connection

Select where gCSP needs to connect to for the animation of a CT program.

- Hostname - the hostname of where the CT program is going to be executed (default: localhost)
- Port - the used socket port (default: 10580)

C.5 Output windows

To get other feedback about the animation besides colors, some output windows are added to the lower screen of gCSP. The following sub-sections describe each output window shortly.

C.5.1 Warnings

This window displays the warnings which occur during the design of a gCSP model. For example, if a process is not grouped in a construct.

C.5.2 Breakpoints

A list off all breakpoints set in the model. Breakpoints can be enabled/disabled by selecting the check boxes or can be removed using the right mouse click menu. Figure C.6 shows the breakpoint output window.

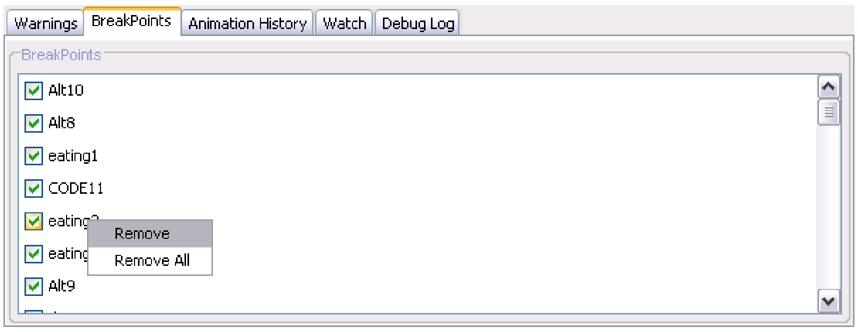
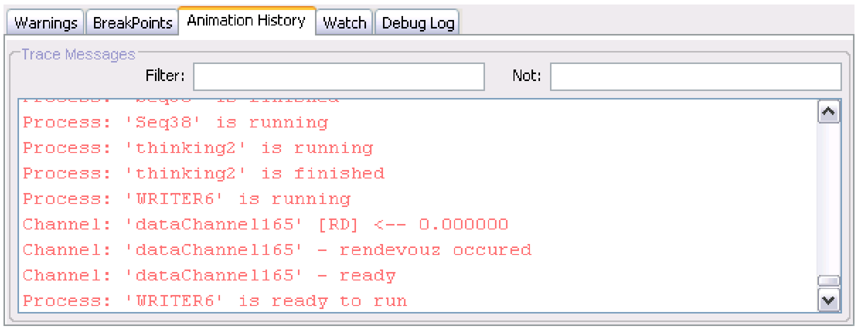


Figure C.6: Trace window

C.5.3 Execution history

Displays the execution order of processes and construct. Channel communication is also shown here.



```
Process: 'Seq38' is running
Process: 'thinking2' is running
Process: 'thinking2' is finished
Process: 'WRITER6' is running
Channel: 'dataChannel165' [RD] <-- 0.000000
Channel: 'dataChannel165' - rendevouz occured
Channel: 'dataChannel165' - ready
Process: 'WRITER6' is ready to run
```

Figure C.7: Execuction history window

Execution messages

A description about the messages shown in the execution history window is given in table C.3

Filtering

To be able to search for current data in the execution history, a filter is added to the window of the execution history. The following options are possible:

- *filter*: shows the execution messages which contains the string
- *not*: hides the execution messages containing the string

If the execution history should be filter on multiple strings, the strings should be separated by a ';'.

Save the execution history to a text file

The execution history can be saved on hard-disk, by right clicking the mouse on the window of the execution history. A popup menu will appear giving the following choices:

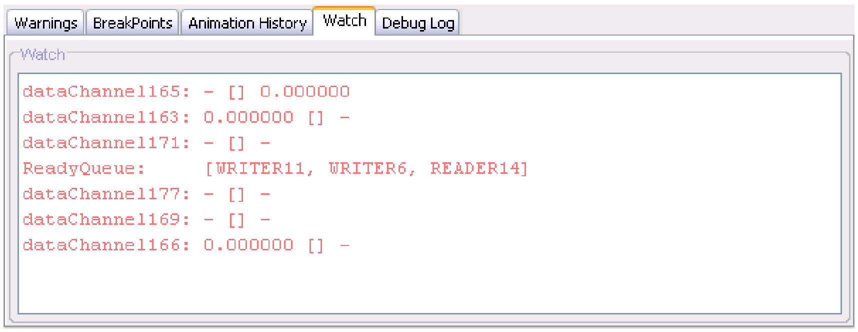
- Clear - clear the screen
- Save - save the displayed history to hard-disk
- Add to filter - add the selected string to the filter field

C.5.4 Watch

Table C.3: Execution Messages

Trace		Description
Process	$->$	the process is created on this address
	new	the process is initialized to its new state
	running	the process is executing on the CPU
	blocked	the process is blocked
	finished	the process has finished its task
Channel	*ready*	the channel is ready to be used for writing and reading
	writer side occupied	the channel has been written
	reader side occupied	the channel has been read
	rendezvous occurred	both channel sides are in use, rendezvous can occurred
	ready to write	in case of a buffered channel, the channel buffer is empty
	ready to read	in case of a buffered channel, the channel buffer is full
	$<--$	data is written to the channel
	$-->$	data is read from the channel

While the animation is running, this window shows a list of variables. This list includes the ready-queue of the running CT program and the contents of all the channels in the model.



```
dataChannel165: - [] 0.000000
dataChannel163: 0.000000 [] -
dataChannel171: - [] -
ReadyQueue:      [WRITER11, WRITER6, READER14]
dataChannel177: - [] -
dataChannel169: - [] -
dataChannel166: 0.000000 [] -
```

Figure C.8: Trace window

C.5.5 Debug log

(Not used at the moment)

Bibliography

ten Berge, M. H. (2005), Design Space Exploration for Fieldbus-based Distributed Control Systems, MSc Report 029CE2005, University of Twente.

Broenink, J. F. and G. H. Hilderink (2001), A structured approach to embedded control systems implementation, in *2001 IEEE International Conference on Control Applications*, Eds. M. Spong, D. Repperger and J. Zannatha, IEEE, México City, México, pp. 761–766.

Controllab Products (2008), 20-Sim, URL `http://www.20sim.com/`.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs and Y. Xiong (2008), The Ptolemy Project, URL `http://ptolemy.eecs.berkeley.edu/index.html`.

Hilderink, G. (2002), A graphical Specification Language for Modeling Concurrency based on CSP, pp. 255–284.

Hilderink, G. (2003), Graphical modelling language for specifying concurrency based on CSP, pp. 108–120.

Hilderink, G. H. (2005), *Managing Complexity of Control Software through Concurrency*, Phd thesis, University of Twente, Netherlands.

Hoare, C. (1985), *Communicating Sequential Processes*, Prentice Hall International.

Jovanovic, D. (2006), *Designing dependable process-oriented software, a CSP approach*, Phd thesis, University of Twente, Enschede, NL.

Jovanovic, D., B. Orlic and J. F. Broenink (2003), An automated transformation trajectory from a model of a controlling system to the control code, in *XLVII Conference ETRAN, XLVII Conference ETRAN*, volume I, Herceg Novi, Serbia and Montenegro, p. 4.

Maljaars, P. (2006), Control of the Production Cell Setup, MSc Thesis 039CE2006, University of Twente.

Maljaars, P. (2007), *gCSP documentation*, Electrical Engineering, University of Twente in the Netherlands.

Posthumus, R. (2007), Data logging and monitoring for real-time systems), Technical report.

Roscoe, A. (1997), *The Theory and Practice of Concurrency*, ISBN 0-13-674409-5.

Sun (2008), Java, URL `http://java.sun.com/`.

Telelogic (2008), Rhapsody, URL `http://modeling.telelogic.com/products/rhapsody/index.cfm`.

Wikipedia (a), Debugging, URL `http://en.wikipedia.org/wiki/Debugging`.

Wikipedia (b), Software Testing, URL `http://en.wikipedia.org/wiki/Software_testing`.

Wikipedia (c), Tracing, URL `http://en.wikipedia.org/wiki/Tracing_%28software%29`.