

University of Twente

EEMCS / Electrical Engineering
Control Engineering



Analysing gCSP models using runtime and model analysis algorithms

Maarten Bezemer

MSc report

Supervisors:

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

November 2008

Report nr. 034CE2008
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

Nowadays the amount of embedded systems is getting bigger and bigger. The systems themselves also become more complex. To aid developers of embedded systems a formal language called Communicating Sequential Processes (CSP) was designed.

At the Control Engineering (CE) group a graphical tool called gCSP is developed to create CSP models. This tool allows for code generation. The generated code can be compiled with the Communicating Threads (CT) library into an executable. Users of gCSP tend to create a separate process for each task. From the model point of view this is excellent, but from a software point of view this consumes too many resources.

First of all a new gCSP file format is created, such that an algorithm is able to read model information without too much effort. Next algorithms are designed and implemented, which perform analysis on models using different techniques.

This assignment results into two analysis tools, which are able to perform an analysis of gCSP models. The first tool is a runtime analyser and uses the compiled executable to get information of the order in which the processes are running. The second tool is a model analyser which directly analyses a gCSP model. It creates a dependency graph and will try to group processes which are related to each other, resulting in a schedule for multi-core target systems. For single-core target systems the results can be used to recreate the model with a few bigger processes.

Both tools are tested to see whether they both are functional and usable. Functional tests are executed using a simple model to be able to manually check the analysis results with the expected outcome. The usability tests perform tests using a series of models and interpret the outcome of the analysers.

From these tests it is concluded that both tools are complementing each other and are suitable for analysing gCSP models in order to be able to create optimised models. The main recommendations are to implement the automatic creation of a new gCSP model using the results of the analysers and to improve the set of rules of both analysers even more to get better results.

Samenvatting

Tegenwoordig wordt het aantal ‘embedded’ systemen groter en groter. De systemen zelf worden daarbij ook nog eens complexer. Om ontwikkelaars hierbij te helpen is er een formele taal ontwikkeld genaamd ‘Communicating Sequential Processes’ (CSP).

De Control Engineering (CE) vakgroep heeft een tool genaamd gCSP ontwikkeld om CSP modellen te maken. Met deze tool is het mogelijk om broncode te genereren. Deze broncode kan worden gecompileerd samen met de ‘Communicating Threads’ bibliotheek tot een uitvoerbare applicatie. Gebruikers van gCSP hebben de neiging om een proces te maken voor elke afzonderlijk taak. Vanuit het oogpunt van het modelleren is dit uitstekend, maar vanuit het oogpunt van de software gebruikt het te veel resources.

Ten eerste is een nieuw gCSP formaat nodig, zodat een algoritme in staat is om de model informatie zonder te veel moeite te kunnen lezen. Vervolgens zijn er algoritmes ontwikkeld die met verschillende technieken analyses op de modellen uitvoeren.

Het project heeft geresulteerd in twee tools waarmee analyses op gCSP modellen uitgevoerd kunnen worden. De eerste tool is een runtime analyse tool die gebruikt maakt van de gecompileerde applicatie om informatie over de uitgevoerde volgorde van de processen te vinden. De tweede tool is een model analyse tool die direct het model analyseert. Het bepaalt de afhankelijkheden en zal proberen de processen die samenhangen te groeperen, resulterend in een indeling voor multi-core systemen. Voor single-core systemen kunnen de resultaten gebruikt worden om van de kleine processen aan aantal grotere processen te maken.

Beide tools zijn getest om te bepalen of ze beide functioneel en bruikbaar zijn. Functionele testen zijn uitgevoerd met behulp van een simpel model, zodat het mogelijk is om handmatig de resultaten te controleren. De bruikbaarheidstesten zijn gedaan met verscheidene modellen en de resultaten hiervan zijn geïnterpreteerd.

Uit de testen kan geconcludeerd worden dat beide tools elkaar aanvullen en dat ze bruikbaar zijn om gCSP modellen te analyseren om uiteindelijk geoptimaliseerde modellen te maken. De belangrijkste aanbeveling is om automatisch de geoptimaliseerde modellen te maken aan de hand van de resultaten van de analyse tools. Een andere belangrijke aanbeveling is om de set met regels van de beide analyse tools te verbeteren, zodat betere resultaten verkregen kunnen worden.

Contents

1	Introduction	1
1.1	Context	1
1.2	Goals of the assignment	1
1.3	Report outline	3
2	Background	4
2.1	Related work	4
2.2	Tools and workflow	4
2.3	gCSP file format	5
2.4	Multi-core environments	5
3	Runtime Analyser	7
3.1	Introduction	7
3.2	Algorithms	8
3.3	Design and implementation	12
3.4	Results	13
3.5	Conclusions	20
4	Choice of the model input mechanism	22
4.1	XML readers	22
4.2	Scanner-Parser generators	23
4.3	Conclusion	25
5	Meta-model creation	26
5.1	EMF projects	26
5.2	Meta-model requirements	26
5.3	The gCSP2 meta-model	27
5.4	Testing of the model	28
5.5	Converting gCSP to gCSP2	29
5.6	Conclusions	29
6	Model analyser	30
6.1	Introduction	30
6.2	Algorithms	31
6.3	Design and implementation	34
6.4	Results	35
6.5	Discussion	39

6.6	Conclusions	40
7	Conclusions and Recommendations	41
7.1	Runtime analyser	41
7.2	The meta-model	41
7.3	Model analyser	41
7.4	Recommendations	41
A	EMF project files	44
A.1	Available project files	44
A.2	Explanation of the EMF related files	44
B	UML diagram of the meta-model	46
C	Heap scheduler	47
C.1	Heap scheduler notations	47
C.2	Decision rule 1	48
C.3	Decision rule 2	50
C.4	Recommendations	51
D	Core scheduler	53
D.1	Index blocks	53
D.2	Heap placement	54
D.3	Ready queue	55
D.4	Recommendations	55
E	Runtime analyser code description	57
E.1	The analyse package	57
E.2	The UI package	57
F	User manual runtime analyser	59
F.1	Controlling the analyser	59
F.2	The model tree	59
F.3	Unordered processes list	60
F.4	The chained processes list	60
F.5	Informative panels	61
F.6	Runtime analyser settings	61
G	Model analyser code description	62
G.1	The analyse package	62
G.2	The UI package	62

H User manual model analyser	63
H.1 Model tree	63
H.2 Controlling the analyser	63
H.3 Schedule information	64
H.4 Dependency graph	64
H.5 Algorithm settings	65
H.6 Exporting the results	66
Bibliography	68

1 Introduction

1.1 Context

More and more machines and consumer applications contain embedded systems to perform their main operations. Embedded systems are special purpose computers which are embedded into a machine or product. They are fed by signals from the outside world and use these signals to control the machine. Designing embedded systems becomes more and more complex since the requirements grow.

To aid developers with this complexity Hoare (1985) and Roscoe (1997) developed a formal language called Communicating Sequential Processes (CSP). It can be used to describe systems with several computational processes running at the same time, called concurrent systems. Embedded systems typically are such systems, so CSP is of great use for developers of embedded systems.

At the University of Twente, the Control Engineering group has created a tool to graphically design CSP models, called gCSP (Jovanovic et al., 2004). gCSP is able to generate code which can be executed on a PC or on a (real-time) hardware target. The generated code makes use of the Communicating Threads (CT) library (Hilderink et al., 1997), which provides a framework for the CSP language. The CSP application can be monitored in gCSP through an animation functionality (van der Steen, 2008). The complete overview is shown in Figure 1.1.

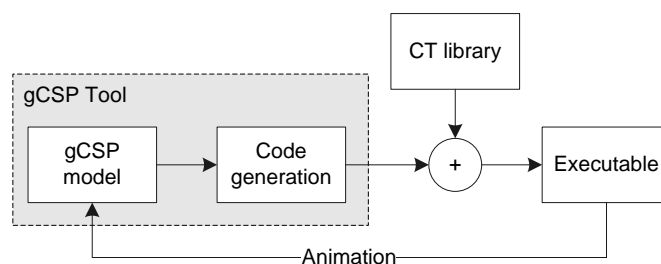


Figure 1.1. Overview of the gCSP & CT framework

A number of undesired features are introduced by the current CSP implementation:

- The models tend to get extensive for large systems. This might result in sub-optimal models, because the software designer is unable to think of all details.
- The control software designers tend to put every controlling part in a small process. This is good in a design point of view, but in an executing point of view it results in a lot of wasted resources. To prevent wasting resources, the model ideally should be flattened into a single process without losing its desired functionality.
- The CT library is designed to run on a single-core target. The scheduler and all of its processes run in the same software thread. Nowadays (computer) systems do not get much faster processors anymore. Instead the processors consist of multiple cores and gain higher speeds by allowing parallel execution of the threads. Distributed systems also make use of multiple codes, distributed over a networked system.

The first two points can be solved by aiding developers with the creation of the models. The last point is addressed by taking steps to make the CT library multi-core aware (Molanus, 2008; Veldhuijzen, 2008). In the future, the library is going to take advantage of these new developments. This project assumes the multi-core awareness of the library already.

1.2 Goals of the assignment

The goals of the assignment are to design and implement algorithms to analyse gCSP models. The analysis results must be usable to see how the models can be run on multi-core target systems.

For both single and multi-core targets the results must also be usable to see how the model can be optimised in order to save resources. Two methods of analysis will be implemented as shown in Figure 1.2.

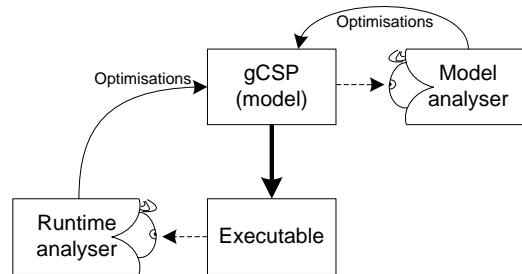


Figure 1.2. Overview of the locations of the analysers

The runtime analyser looks at the executing application, especially at its performance and behaviour. Its results can be used to determine a static order of running processes. The model analyser looks at the models directly and schedules the processes on a number of available cores.

Both analysers should get different results due their different points of view. These results can be combined and used to make an optimised version of the model.

1.2.1 The runtime analyser

The runtime analyser should use runtime information to get a process order. In order to be able to get runtime information, the analyser should be able to connect to a running gCSP model. So it can retrieve information about the available processes. This information should be fed to an algorithm which will analyse the running model and produce chains of processes to show the running order of the process.

When the algorithm is designed and implemented in a tool, the results should be usable

- to get information in order to get a better understanding of the model
- to ideally provide enough information to optimise the model. For example to combine several sequential processes into one larger process.

1.2.2 The model analyser

The model analyser should be able to analyse the model during design time. It loads a model and uses its information to create a dependency graph. This graph can be used to schedule the processes in such a way that a optimal schedule is obtained for a given target system.

In order to be able to read the model information, this information should be easily accessed. This is not the case in the current gCSP models, so a new format will be created. This format should be described by a meta-model. A meta-model is a model of a model, so the contents of the actual model are predefined. The new meta-model should be able to contain all required data for the analyser. It should provide functionality to store and load the model and it can function as a basis for an improved gCSP application.

At the end of the trajectory the tool implementing the model analysis algorithm should produce information

- which indicates how the processes are related to each other
- how the processes could be scheduled on the given target system consisting of at least one core

In order to put the processes into one big process, their order should be predefined. Therefore, the resulting schedule should be made sequential by adding time information to each process.

1.3 Report outline

Before starting to describe the algorithms and their requirements, a background chapter explains the environment more extensively. Chapter 2 contains this background information. Chapter 3 describes the design of the runtime analyser, its algorithm, implementation and results. Chapter 4 and 5 describe the choices of input mechanisms for the new gCSP meta-model and the meta-model itself.

This information is required for the model analyser, which is described in Chapter 6. Like the runtime analyser chapter, this chapter also describes the design of the algorithm, the implementation and the analysis results. More information about the model analysis algorithms can be found in Appendices C and D.

Finally Chapter 7 summarises the conclusions and presents recommendations to improve the analysers and how they can be used for other purposes. User manuals of both tools can be found in Appendices F and H.

2 Background

This chapter describes the background of the assignment. The first section contains information about related work, on which parts of the solutions are inspired. The next section describes the tools and workflow used at the CE group. Followed by a short description on the gCSP file format. The chapter ends with a section providing information about multi-core environments.

2.1 Related work

This section describes related work on scheduling models for usage on multiple processing units. Firstly Boillat and Kropf (1990) designed a distributed mapper algorithm that runs on the target system itself. It maps the processes in an optimal way, by starting to place the processes on the network. It measures the delays and calculates a quality factor. The processes, which have a bad influence on the quality factor, are remapped and the quality is determined again. This optimising analysis method is called post-game analysis. The used target system is a transputer network of which the communication delays and network connections differ depending on the nodes which need to communicate. Points of interest are:

- the algorithm to find a mapping such that the processes on a transputer only need connections to their neighbouring transputers. A transputer only can communicate to its neighbouring transputers, other transputers can be reached by finding a path using the neighbouring relations. This obviously is undesired, since it delays multiple other transputers as well.
- the optimisation for load balancing and communication minimisations, so the available resources are optimally used.
- the fact that the mapping algorithm is designed to be distributed. This might save analysis time for large transputer networks with a lot of possibilities.

Magott (1992) tried to solve the optimisation evaluation by using Petri nets. First a CSP model is converted to such a Petri net, which forms a kind of dependency graph, but still contains a complete description of the model. His research added time factors to these Petri nets and using these time factors he is able to optimise the analysed models.

Finally van Rijn (1990) describes an algorithm to parallelise model equations in his masters thesis. He defined equation dependencies and used those to schedule the equations on a transputer network. Equations with a lot of dependencies form heaps, which are a kind of groups. These heaps should be kept together to minimise the communication costs. Furthermore, his scheduling algorithm tries to find a balanced load for the transputers, but keeping the critical path as short as possible.

This assignment mainly makes use of the work of van Rijn. Especially since the equations he tried to optimise could be compared to the CSP processes. His algorithms work without the need of the target system during analysis, although it is possible to run the algorithms on the target system as well if required. This is not the case for the work of Boillat and Kropf, their algorithm must be run on the target system, since the timing information needs to be measured. Their distributed analysis algorithm is not usable in this situation, since single-core target systems must be supported as well.

2.2 Tools and workflow

At the Control Engineering (CE) group a trajectory to design embedded software is developed (Broenink et al., 2007). Figure 2.1 shows this trajectory. The dashed box shows the steps in the trajectory where this assignment could be placed in.

Each step can be iterated by verifying the results and use it to improve the model. When a step reached the desired result, the following one can be started.

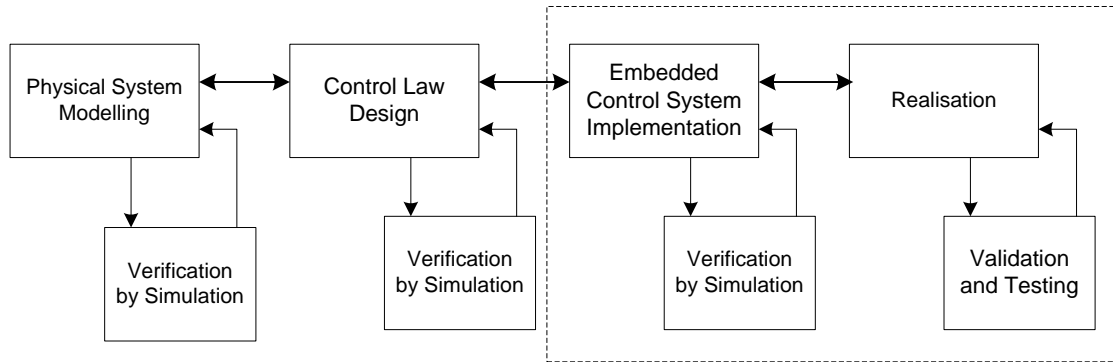


Figure 2.1, Control software design trajectory

Short descriptions of the steps are:

- **Physical System Modelling**
A dynamic behaviour of the system is modelled. This can be done using bond graphs and 20-sim (Controllab Products, 2008).
- **Control Law Design**
The model containing the dynamic behaviour is converted into a set of control laws.
- **Embedded Control System Implementation**
The control laws are changed into (software) algorithms. For this task a graphical tool called gCSP (Jovanovic et al., 2004) is used.
- **Realisation**
The gCSP model is converted into an executable which can be run on a computer, ADSP, FPGA or ARM. CSP execution functionality is added by the CT library (Hilderink et al., 1997).

gCSP is a graphical tool to build models containing some sort of control algorithm, shown in Figure 2.2. More information on the CT library and the graphical modeling language can be found in (Hilderink, 2005) chapter 3.

In order to create an application from a gCSP model the Communicating Threads (CT) library is used as execution framework. This library defines the CSP constructions so they can be used by the application without the need of implementing them manually. The overview of the relations between gCSP, CT library and this assignment can be seen in Figure 1.2.

2.3 gCSP file format

When a model is drawn in gCSP, it can be stored in the Extensible Markup Language (XML) (World Wide Web Consortium, 2008) format. This is a well-defined format and very usable for the job.

However, by using XML one is free to choose how the data itself is stored within XML. gCSP chooses to store its memory directly into the XML format using the Java XML serialiser. This results in large files which are hard to read with an application other than gCSP. These files also get excessively large because the model objects have several copies of itself in the memory which all get stored separately.

Since this project is all about creating two tools (external applications) to analyse gCSP models this will be a problem. Hence the choice in the previous chapter to create a new meta-model, with a better and simpler defined structure.

2.4 Multi-core environments

Modern (embedded) systems have multiple cores available. A core is a processing unit which is able to run software. An application can be split into multiple threads. A thread is a part of the

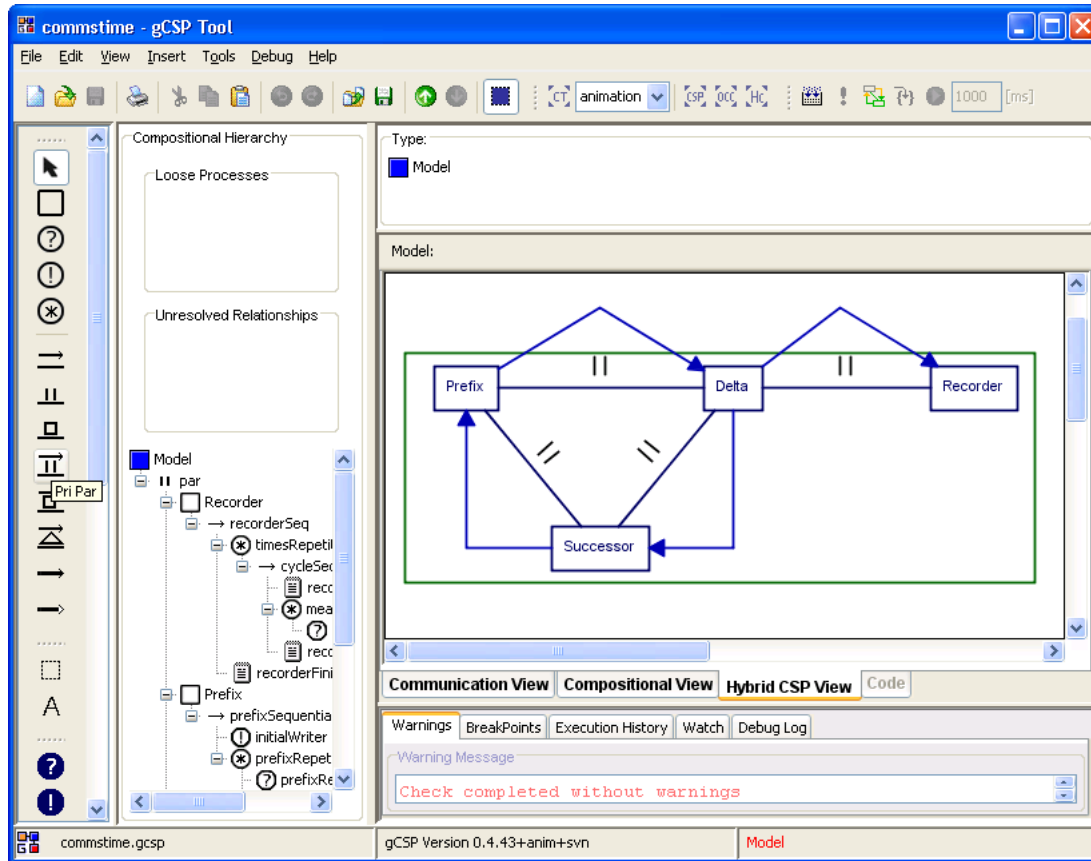


Figure 2.2, gCSP user interface, showing parallel processes connected with channels

program which has its own task and is able to perform it without (much) need of other threads. Each thread could be executed by its own core, resulting in real parallel running software blocks.

It might be clear that this is speeding up the execution of applications greatly, which is a desired feature for control software as well. An example of control software that needs to run in parallel on a distributed system is the software for TULip (3TU, 2008). It is a (soccer) robot which has multiple controllers connected to each other, in order to control the complete robot.

However, designing a (CSP) application for a multi-core system is tricky (Spath et al., 2006) since the available processes should be scheduled into threads. When the processes are scheduled in a bad way the performance is not improving as much as possible and might even be dropping. So this project will assume the availability of a CT library with multi-core support and will present analysers which will help scheduling processes into threads which can be run on the available cores of the target system.

Another problem is that the current CT library is not yet supporting multi-core target systems. A gCSP model is generated into a single application and the CT library puts all processes into the same thread. Currently the CE group is working on solving issues to make the CT library run on multi-core systems (Molanus, 2008; Veldhuijzen, 2008).

3 Runtime Analyser

This chapter describes the algorithms and the implementation of the ‘gCSP Runtime Analyser’ or Runtime Analyser for short. As described in Section 1.2.1, this tool should try to find execution patterns by analysing a compiled gCSP program. These patterns can be used to optimise the model.

3.1 Introduction

The Runtime Analyser is an application which runs next to a gCSP generated model executable, or executable for short, as shown in Figure 3.1.

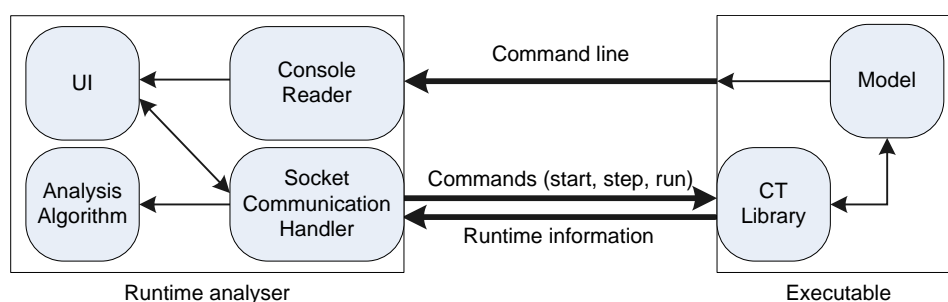


Figure 3.1. Architecture of the analyser and the executable

The main part of the analyser consists of the implementation of the used algorithm, which is explained in Section 3.2. In order to feed information to the algorithm, communication with the executable is needed. Two types of communication are used:

- Command line communication, which consists of texts normally visible on the command line.
- TCP/IP communication, which is send over an TCP/IP channel between the CT library and the Communication Handler. It originally is used for animation purposes (van der Steen, 2008).

The implementation of both types of communication is described in Section 3.3.1. Last but not least a User Interface (UI) is added to provide interaction between the analyser algorithms and the user. A picture of the UI is shown in Figure 3.2.

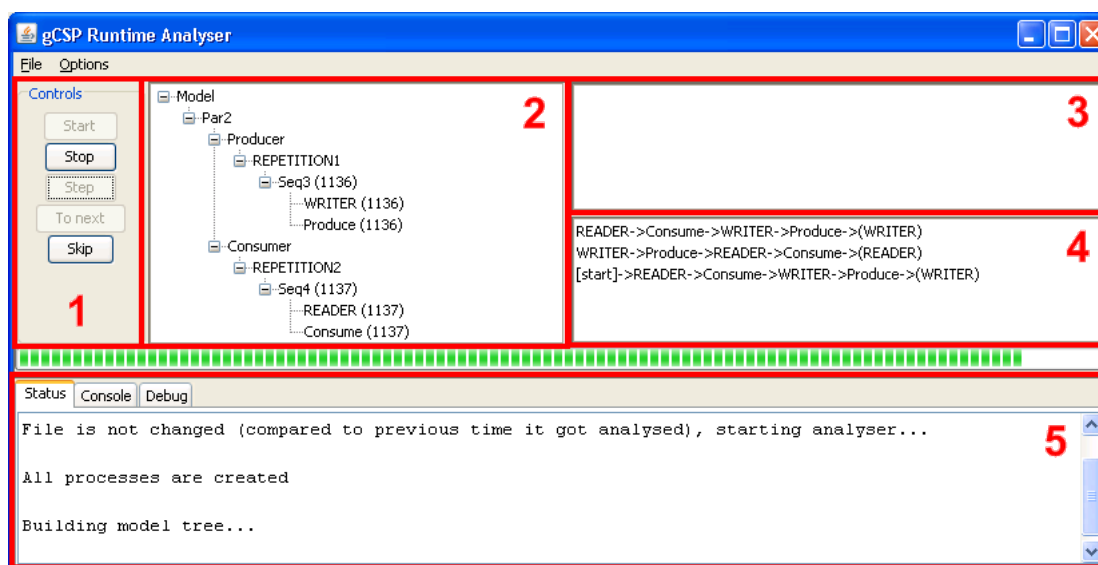


Figure 3.2. UI of the runtime analyser

The top left part, numbered 1, contains buttons to control the algorithm after a model is started. Number 2 is the output of the tree recreation algorithm described in Section 3.2.1. Panel 3 initially shows the processes available in the model and during the analysis the processes which are not placed in the model tree yet. The panel is empty in the figure since all processes are placed. The results of the process order algorithm, described in Section 3.2.2, are visible in panel 4. Finally part 5 shows the status of the analyser, other available views are the console line texts and a debugging view, mainly showing the TCP/IP communication. A more complete description of the user manual can be found in Appendix F.

3.2 Algorithms

As briefly mentioned before the runtime analyser makes use of two algorithms. One to reconstruct the model tree and one to determine the order of the execution of the processes. Both algorithms run simultaneously, but the model recreation algorithm finishes when the model tree is completed. The process order algorithm will not stop until the user stops it.

Both algorithms use the state change information of the processes. Before the algorithms can be started the executable needs to initialise its processes. The information send during the initialise stage is required to initialise the analyser as well. The newly created processes are stored in a list waiting to be placed in the model tree. When breakpoint information is received the analyser knows that the initialisation is completed and the executable is ready to run.

3.2.1 Model construction algorithm

Model reconstruction seems a little unnecessary, since the original gCSP model files will be present in most cases. Some reasons to still reconstruct a model are firstly the fact that the gCSP model files are complicated to read, so it is hard to extract information from them. Secondly it might be the case that the model files are not present or they might have been changed in a new version of gCSP. By not using them, the analyser will be not influenced by model file changes. Thirdly the executable which is going to be analysed might be different compared to the model being used. By being independent of the model this problem will not occur. Another advantage is that unused processes automatically become visible when they do not disappear from panel 3.

After the executable has created the processes and panel 3 contains these processes, an empty model tree is instantiated. This tree represents the compositional relationships between all processes. All grouped processes appear on the same branch and all groups themselves also appear on the same branches when belonging together. Processes are grouped to inform the scheduler in the CT library how they should be executed, for example in parallel or sequential order. Figure 3.3 shows such a reconstructed model.

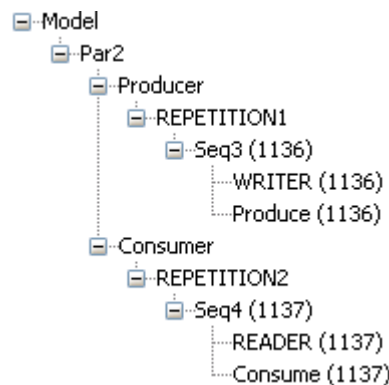


Figure 3.3, A reconstructed model tree

The leaves of the tree are the processes. The other parts are the groups which determine the process order. In this example the ‘WRITER’ and the ‘Produce’ processes are executed in sequential order

and the same goes for the processes 'READER' and 'Consume'. Both process groups are grouped with a repetition to make sure they keep repeating until the program is terminated. Finally the two repetition groups are grouped together to make sure they are executed parallelly.

The algorithm monitors the state change information and with a single rule it is able to reconstruct the model tree.

If a process is started or becomes ready for the first time, make it a sibling of the previously activated process **Rule 3.1**

As Rule 3.1 states a process should be added to the last activated process. The last activated processes is the last process which got finished or blocked, depending on the situation.

Each process that is put in the model tree is removed from the list shown in panel 3 of Figure 3.2. When panel 3 is empty the model reconstruction algorithm is finished.

3.2.2 Process ordering

The second algorithm will try to simplify the model by grouping processes into bigger processes. So less content switching is required, resulting in less used resources when executing the model.

Like the previous algorithm, this algorithm also makes use of the state change information. This time only the state change 'finished' is used, since the process order of the chains should represent the order of the processes being finished. This way the chain shows the sequential relation between the processes. In the case that the 'started' state would be used the chain order would be messed up, because a process can be started and blocking multiple times before finishing once. Resulting in a chain which would block if it would be implemented.

The algorithm operates in two modi. The first mode is active when the current chain has not reached an endpoint yet. The second mode is for chains which have set their endpoint(s). A chain is ended if a process is added to the chain which already was added. In order to keep things simple, a chain is allowed to include a process only once.

The used notation will be described in the following section. The two modi are described in the next two sections.

Used notation

As described before, the algorithm tries to find static process order chains. When a series of processes is executed during the analysis they become a chain. The notation of the chains in the two sections explaining the algorithm are the same as used in the user interface of the runtime analyser. Figure 3.4 shows an example notation of these chains.

```
D->C->F->B-> (B, D*)
B->E->A->C-> (E**, D)
[start]->A->B->C->D-> (B)
```

Figure 3.4, Example notation of the chains

Each chain has a unique starting process followed by a series of processes and is ended by an endpoint. The endpoint is shown between the parentheses. It consists of either a single process or a group of processes. The endpoint indicates the chain(s) which might become active after the current chain is ended. Asterisks are used to indicate that chain is looped. One asterisk is added when an ending process refers to the beginning of its chain. Two asterisks are added when an ending process is pointing to a processes in the middle of its chain.

When looking at this example, the executable starts at its main starting point '[start]'. When the chain is ended the chain starting with **B** is activated. At the end of that chain two possibilities are available. The outcome depends on the internal mechanisms: either the current chain stays active and continues at process **E** or the chain starting with **D** becomes active. When chain **D** is ended two possibilities are also available: either chain **B** or **D** becomes active.

Rules for chains with no endpoint

The rules in this section are responsible for creating and extending new chains. The trace shown in Figure 3.5 is used to explain the rules in this section.

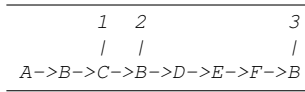


Figure 3.5, The used trace

A trace is a sequential order of processes fed to the algorithm. The numbers above the trace indicate the position for the explanation of the rules.

If the state of process p changes to 'finished' add p to the end of the active chain. **Rule 3.2**

Rule 3.2 is the most basic rule to create new chains. When position 1 is reached, it results in a new chain with processes **A**, **B** and **C** added, as shown in Figure 3.6.

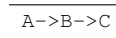


Figure 3.6, A new chain

At position 2 process **B** is finished for a second time. Since it is not allowed to add a process twice to a chain, Rules 3.3 and 3.4 are required.

If process p got added but it was already present in the chain, it will become an endpoint of the chain. **Rule 3.3**

If a chain gets ended by process p , the chain starting with p is looked up.
If the chain is not available a new chain starting with p will be created. **Rule 3.4**
The found or newly created chain will become the new active chain.

Rule 3.3 defines when a chain should be ended. When a chain that starts with process **B** is not found, Rule 3.4 states that a new chain should be added. The result is shown in Figure 3.7.

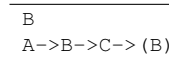


Figure 3.7, A Second chain is added

When position 3 is reached process **B** again applies to Rule 3.4. This time a chain starting with process **B** is found: the current active chain. No new chain needs to be added and the current chain is ended as shown in Figure 3.8.

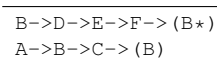


Figure 3.8, The complete trace converted to chains

These three rules are sufficient to add processes to new chains. New chains always have an unique starting process and do not have processes added twice to them.

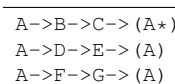


Figure 3.9, No unique starting processes

Figure 3.9 shows chains without unique starting processes. The first chain ends with **A***. No problem occurs here since the current chain is reactivated when it ends. The other two chains result in a problem, since it is unclear which chain is activated next. Only chain which is certainly not activated is the current chain.

Figure 3.10 shows the problem arising without Rule 3.3 being active. The end process refers back to the process **B** in the middle of the chain. Problem is the fact that two processes **B** are present in the middle of the chain and it is unclear which one is referred to.

A->B->C->B->D->E-> (B**)

Figure 3.10, No unique processes in chain

Rules for chains with endpoint(s)

When the active chain is ended already, the rules should check if the chain is correctly created. Therefore the position in the currently active chain should be remembered. To explain the rules in this section the trace is extended, as shown in Figure 3.11.

1	2	3	4	5
A->B->C->B->D->E->F->B->D->G->H				

Figure 3.11, The extended trace

First of all it should be checked if the active chain was at its end. Position 3 ends the chain starting with **B** shown in Figure 3.8. Rule 3.5 performs this check and makes the current chain active again.

If the finished process p is at the end of the chain, find the chain starting with process p and make it the active chain. **Rule 3.5**

When the chain is not ended, the current position in the chain should match the finished process resulting in Rule 3.6.

If the current position in the chain does not match the finished process p the chain must be split. **Rule 3.6**

At position 4, process **D** got ended. Previously process **B** was ended, so process **D** is expected next. No problems arise since the expected and the finished processes match. At position 5 process **G** finished, which is not the expected process after **D**. According to the rule, the active chain should be split.

Splitting chains is a complicated task, since the resulting chains still need to match the previous part of the trace. For the current trace the active chain should be split after process **D**, since that process was still expected. Rule 3.7 defines the steps to be taken in such a situation.

The active chain should be split at process e when process p is unexpected and a chain starting with process e is not yet present.

To split a chain:

End current chain before process e and put the processes after e in a new chain starting with process e **Rule 3.7**

Create a new chain starting with process p and make it the active chain.

It results in a new chain starting with **E** containing the remainder of the active chain. A new chain is created starting with process **G** and is made active. The resulting chains are shown in Figure 3.12.

G
E->F-> (B)
B->D-> (E, G)
A->B->C-> (B)

Figure 3.12, Chains after splitting chain **B**

When following the trace from the start till position 5, the chains in the figure are matching the given trace again. The active chain is the chain starting with process **G** and is not yet finished so the rules of the previous section apply to it.

If the trace is extended again according to Figure 3.13, a problem arises at position 7.

	1	2		3	4	5		6	7						
A->	B->	C->	B->	D->	E->	F->	B->	D->	G->	H->	B->	D->	G->	H->	I

Figure 3.13, The final trace

Till position 6 no problems occurred, Figure 3.14 shows the result till this position.

G->	H->	B->	D->	(G*)
E->	F->	(B)		
B->	D->	(E, G)		
A->	B->	C->	(B)	

Figure 3.14, The chains till position 6

At position 6 process **H** is finished and expected, so position 7 is reached. A problem occurs at position 7: Process **I** is finished but process **B** is expected, so the chain should be split. Problem is, a chain starting with process **B** is present already. Having two chains starting with the same process is not allowed. However, it is clear that the processes after process **H** match the chain starting with **B** exactly. Rule 3.8 provides a solution for these situations.

The active chain should be split at process e when process p is unexpected, but a chain starting with process e is present already.

Compare the remain processes after e with the chain starting with process e

If both parts are equal, remove the remaining processes in the active chain starting at process e . **Rule 3.8**

And create a new chain starting with p and make it the active chain.

If the comparison did not succeed, the static process order cannot be determined and the algorithm should stop.

Figure 3.15 shows the result after applying the rule.

I			
G->	H->	(B, I)	
E->	F->	(B)	
B->	D->	(E, G)	
A->	B->	C->	(B)

Figure 3.15, The chains after position 7

Process **I** is active and can be created using the rules of the previous section again.

If the comparison of the rule failed, it indicates that two unequal chains starting with the same process should be created. This situation is not supported by the algorithm, because it results in a badly construct model. The internal mechanisms of the CT library could not determine which of the two chains should be used in a certain situation.

A simple example which has a chance to fail the comparison would be a ‘one2any channel’: one process writes on the channel and multiple processes are candidates to read the value. If the reading order of these candidate processes is not deterministic, Rule 3.8 fails.

3.3 Design and implementation

3.3.1 Communication

In order to be able to follow the execution patterns of a program, the algorithm has to be fed with usable information. In order to get hold of this information some sort of communication is required. Two communication types are used when analysing the executable: command line output and TCP/IP communication. This is shown in Figure 3.1.

Command line communication

The used command line communication is a one way communication: only from the executable to the analyser. The executable sends output texts to the command line which are intercepted

by the analyser. The output texts are sent as two character streams, one for the regular output texts (stdout) and one as error output texts (stderr). Both streams are being watched by a thread in the analyser and the texts are displayed in the console tab of the user interface. The analysis algorithms do not make use of this information.

The command line communication allows for bidirectional communication. An input character stream (stdin) is available as well, but the analyser does not make use of it.

TCP/IP communication

The second communication type is much more sophisticated, since it makes use of an available protocol. The protocol is part of the animation functionality of gCSP and allows for influencing and monitoring the running processes.

After the executable is loaded it opens a TCP/IP port and listens to incoming connection requests. When a request is received and the connection is made, the executable starts initialising its processes. After the initialisation is done, the executable waits on commands to start or step the processes. During the creation and state changes of the processes information is sent over the communication channel to the analyser. An intermediate object receives and filters this information and sends it to the algorithms. More information about the usage of these commands can be found in Section 3.3.2.

3.3.2 Controlling the execution flow

In order to influence the flow of the execution of the processes the animation functionality is used. Commands are sent to the executable instructing to start, step or stop the execution.

The executable (or more precisely the CT library) sends status information to notify the listening application about state changes, the creation of new processes, breakpoints being placed, error information and logging information. This information is used to find out about the situation of the executable and the model it contains. So the analyser is able to reconstruct this model without having knowledge of the original gCSP file. How this information is used is described in Section 3.2.

3.4 Results

This section describes the results of the analyser, first a functional test will be performed. Next usability tests are performed to see whether the results of the analyser can be used when more realistic models are used.

3.4.1 Functional test

As a functional test a simple producer consumer model is created, as shown in Figure 3.16. Besides just sending data over a channel it also calculates the time passed since the executable was started. So it is possible to compare the model with an optimised version later on.

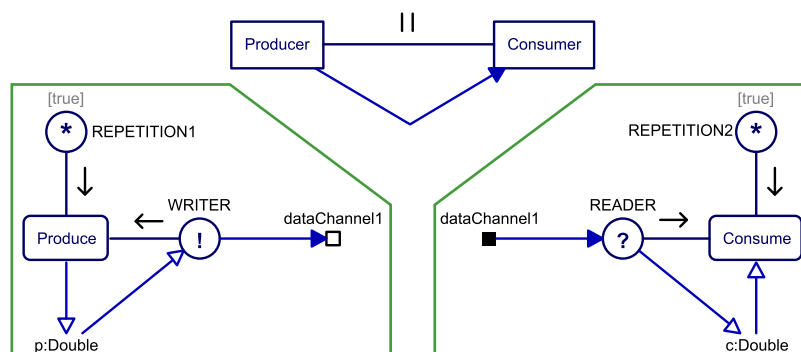


Figure 3.16. The ProducerConsumer model

The model is loaded in the analyser, which automatically generates code, compiles and creates an executable. Figure 3.17 shows the result of the analyser.

```

READER->Consume->WRITER->Produce-> (WRITER)
WRITER->Produce->READER->Consume-> (READER)
[start]->READER->Consume->WRITER->Produce-> (WRITER)

```

Figure 3.17, Three process chains for the ProducerConsumer model

Two chains are created which seems to be the same, except that they have a sort of ‘phase change’. A starting chain is shown as well, but it is exactly the same as the READER chain, so it can be neglected. Next the channel is changed to a buffered channel with a size of 1, see Figure 3.18. Only one chain is left which has a circular reference to itself, hence the asterisk at the end. The starting chain is the same again and can be neglected.

```

WRITER->Produce->READER->Consume-> (WRITER*)
[start]->WRITER->Produce->READER->Consume-> (WRITER)

```

Figure 3.18, Result for ProducerConsumer test with a buffered channel

When comparing both results their execution order looks the same on a first glance:

```
WRITER->Produce->READER->Consume
```

But the ending processes are different. In the first result it points to READER which results in:

```
WRITER->Produce->READER->Consume->READER->Consume->WRITER->...
```

And at the second result it points to WRITER resulting in:

```
WRITER->Produce->READER->Consume->WRITER->Produce->...
```

So depending on the channel type, different behaviour is found by the analyser.

Another difference is the amount of chains, the first result has two chains because references are only allowed to the start of the chain. So a new chain has to be added in order to reference to the start. The second result references to the start of itself which is allowed. The second result cannot directly be used to create an optimised model, since the channel needs two processes to be ready before sending data. It will not work with only one chain and thus one process.

Figure 3.19 shows the simplified version of the original model. It has only two processes which contains combined code from the original model in order to represent the two chains found. Also the repetitions are replaced by an infinite while-loop in the code. Resulting in a model which is as optimal as possible.

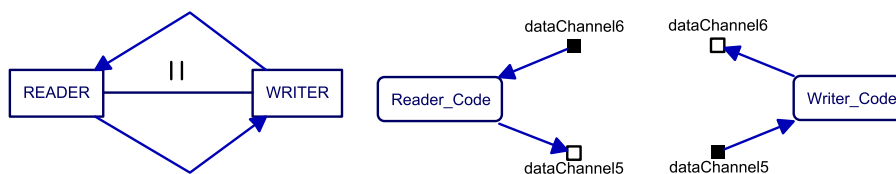


Figure 3.19, The simplified ProducerConsumer model

When the runtime analyser tries to analyse it will not give results. Because non of the processes will finish. Due to the added infinite while-loops, no finish state information is send. This is not a problem since it is more interesting to compare the speeds of both models.

Both models are build without animation functionality and the output from the consumer and producer. The test returns the consumed value at 60 equal intervals to get an average value for one run. This test is repeated 10 times to correct for starting issues, since the first run always seems slower than the following runs. The slower first run is caused by the one-time initialisation of the processes. Another cause is the scheduler of the CT library, which needs more time to schedule a process for the first time.

The original gCSP model needs quite some remodelling in order to comply to this test setup, as can be seen in Figure 3.20. The simplified model consists of two code blocks, so it is easy to implement the test setup.

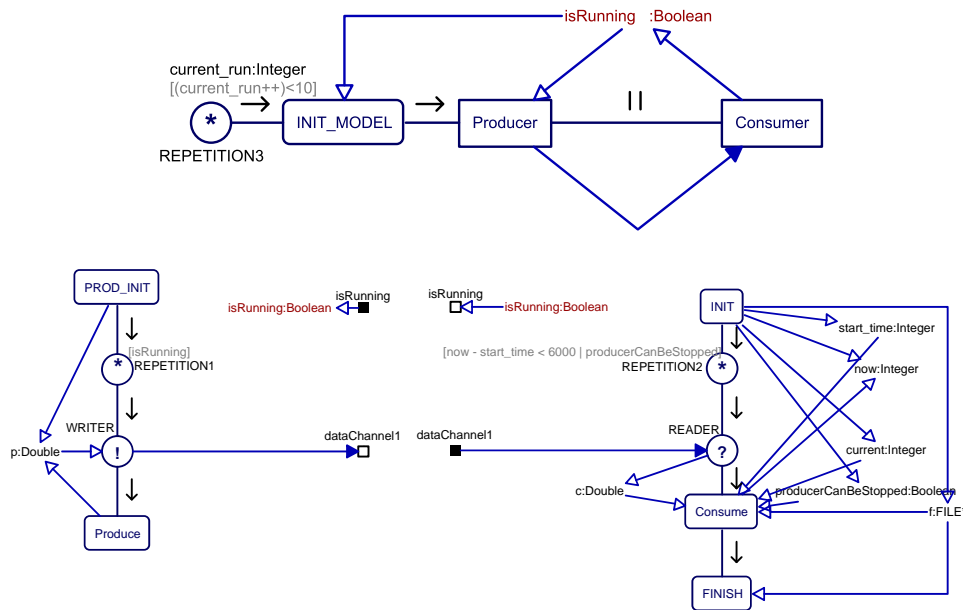


Figure 3.20, The ProducerConsumer model with measurement added

The figure shows that a lot of extra processes unfortunately are required in order to perform the measurements. This is the result of the lack of an `init` and `finish` code block possibility for processes. So the results will not be very accurate since extra context switched are introduced by the measurement system.

The found results are shown in Figure 3.21. The y-axis shows the amount of data produced and consumed for the corresponding interval. The more data is processed the better, since it indicates that the model is running faster.

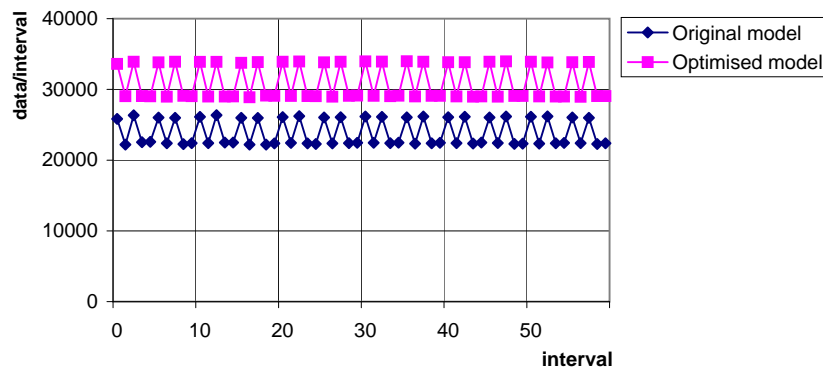


Figure 3.21, Measurement results

It is clear that the optimised model indeed has better results compared to the original model. The average factor between both models is 1.298, so the optimised model is about 30% faster. The minimum and maximum factors are 1.293 respectively 1.304. So doing multiple runs seems to result in a steady measurement result.

The only known limitation is the use of `One2Any` or `Any2Any` channels with three or more processes connected. Models which use these types of channels much, compared to other parts, might fail during analysis. The analyser will give a warning that it is not possible to continue analysing anymore. The main reason are chains which need to be split without being allowed. As described at the end of Section 3.2.2.

Caution should be taken when the analysed model is using recursions or alternative channels. This may lead to processes which are not used during the analysis. This becomes visible when panel 3 of Figure 3.2 still contains unused process, even though the analysis is finished. In such unexpected situations the user should recheck the original model for design errors. Otherwise one should remember to add the unused processes as well, when optimising the model using the results of the analyser.

3.4.2 Usability tests

The previous section showed that the analyser is working and that the optimisation indeed is better, compared to the original. This section describes the analysis results of several more realistic models to evaluate the usability of the tool.

Production cell

Figure 3.22 shows the used model of the production cell (van den Berg, 2006). This is the analysis model, which differs greatly from the model to actually control the production cell. The analysis model contains the blocks available on the production cell, but not the actual controlling parts.

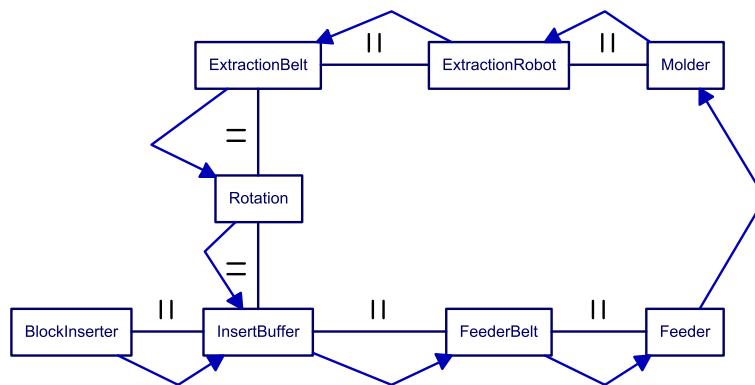


Figure 3.22, production cell

The processes define parts of the production cell which all are connected by transport mechanisms, like a belt or a gripper. In the model the processes are connected by channels to simulate these connections. Most of the blocks only read from their input channel and write to their output channel. The BlockInserter inserts a block into the InsertBuffer which will insert the block in the circular channels when the channel to FeederBelt is empty. After that the block will circle forever in the model. The used BlockInserter only inserts one block, but it would be possible to stress test a model by inserting multiple blocks. At a number of seven inserted blocks the model should deadlock, since all blocks will be waiting for each other.

The results of one circling block can be viewed at Figure 3.23. The chains were too long to fit on a single line so they are separated by empty lines.

```

WRITER4->READER5->READER6->WRITER5->WRITER3->READER4->READER7->WRITER6->WRITER2->READER3
->READER8->WRITER7->READER1->WRITER8-> (WRITER4*)

WRITER5->READER6->WRITER4->READER5->WRITER3->READER4->READER7->WRITER6->WRITER2->READER3
->READER8->WRITER7->READER1->WRITER8-> (WRITER4)

WRITER6->READER7->WRITER5->READER6->WRITER4->READER5->READER4->WRITER3->WRITER2->READER3
->READER8->WRITER7->READER1->WRITER8-> (WRITER5)

[start]->WRITER3->READER4->WRITER2->READER3->WRITER7->READER8->WRITER6->READER7->WRITER5
->READER6->READER5->WRITER4->READER1->WRITER8-> (WRITER6)

```

Figure 3.23, Result of the analyser for the production cell model

Compared to the results of the ProducerConsumer example, it is much more complicated at a first glance. At closer examination the bottom three chains are one long chain and. Starting at

‘[start]’ and ending with chain ‘WRITER5’. After the starting phase is finished the model will circle in top chain ‘WRITER4’.

The actual result of the analyser consists of one chain, resulting in one process when implementing it. It might be clear that this will not work, since a channel should be connected between two processes. A solution, for a single core target, would be replacing the channels for variables. Now the sequential scheduled code blocks can write and read a variable without getting blocked.

Another problem is checking for the correctness of the result. The Reader and Writers are not nicely following up. So it becomes nearly impossible to check whether the correct order is found. Since the analyser is using the runtime information the order should be correct as is was with the ProducerConsumer model. All more realistic models are complex, as shown with this example. The results of the analysis are hard to be checked and without knowledge of the inner workings of the CT library it becomes virtually impossible.

This model showed that an alternative channel might result in unused processes. Panel 3 (Figure 3.2) keeps displaying ‘WRITER9’, ‘WRITER10’, ‘WRITER11’, ‘WRITER12’, ‘WRITER13’ and ‘READER2’. The model was build to check for deadlocks when seven block are inserted into the production cell. The analysis shows that only one block is inserted and keeps circling around. So it can be concluded that the analyser also can be used, to quickly check if a model is behaving as one should expect when using these channel types.

Dining Philosophers

A well-known concurrency problem is the dining philosophers dilemma. Concurrency is a situation with multiple processes running interleaved on a single core. Without proper implementation deadlocks might occur when two or more processes are waiting on each other. Figure 3.24 shows a gCSP model of this dilemma.

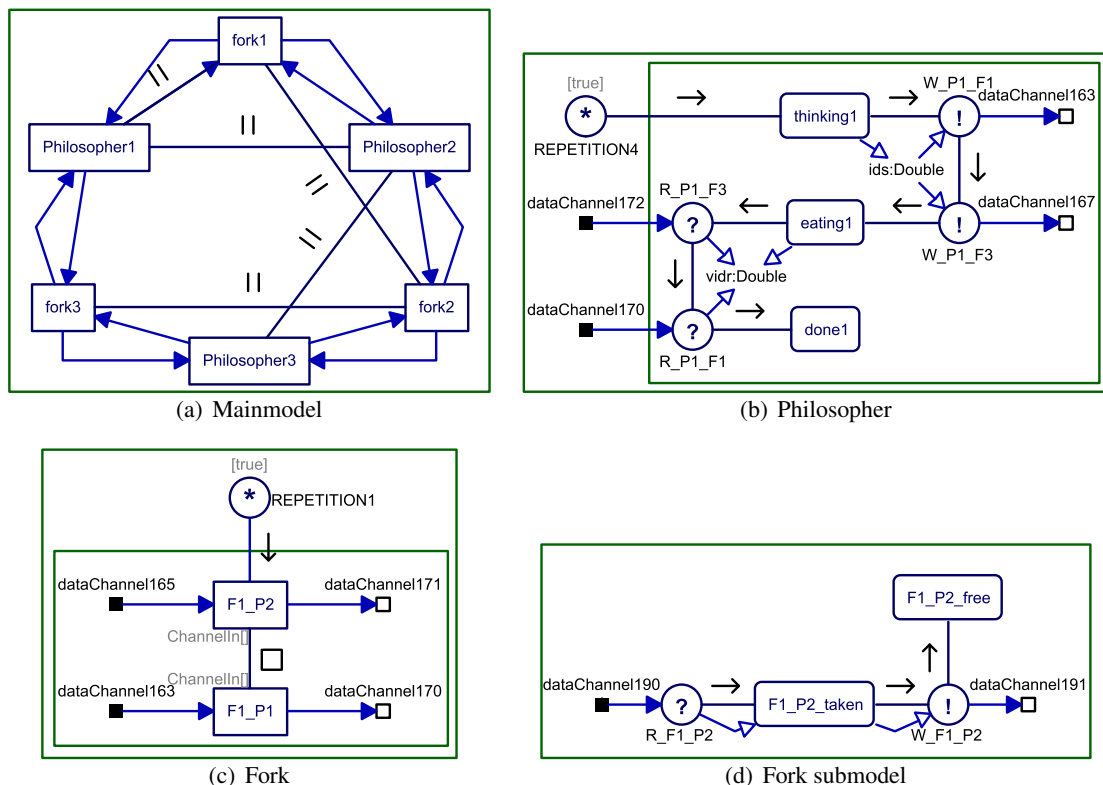


Figure 3.24, Dining philosophers

All philosophers are busy thinking initially, after a while they might get hungry and want to eat a bit. Since they all have proper manners they use forks, available at the left and right of each

philosopher. The only problem is the fact that the amount of forks is equal to the amount of philosophers. A hungry philosopher first grabs his left fork and next his right one, when both forks are obtained he starts to eat. When stuffed he drops both forks making them available for his neighbours, and start thinking again until he gets hungry.

A problem happens when a fork is taken by a neighbour philosopher, since the philosophers have their matters they patiently wait until the fork is released. The real dilemma occurs when all philosophers obtained one fork and are waiting for the other fork. They all will starve to death since a deadlock situation occurred.

The model consists of three philosophers with their three forks. Sub models of the philosophers and the forks are shown as well. The names of the model parts are abbreviated to keep the chains as short as possible. Examples of the abbreviations are: 'P1' for philosopher1 and 'F1' for fork1. The model is implemented correctly with respect to deadlocks, so it is possible to run using the runtime analyser. Its results can be found at Figure 3.25 and again the results are complex.

```

thinking2->W_F1_P2->F1_P2_free->R_F1_P2->F1_P2_taken->W_P2_F1->W_P2_F2->eating2
->R_F2_P2->F2_P2_taken->W_F2_P2->F2_P2_free->R_P2_F2->R_P2_F1->done2->(thinking2*)

[start]->thinking1->thinking2->thinking3->W_P2_F1->R_F1_P2->F1_P2_taken
->W_P2_F2->eating2->R_F2_P2->F2_P2_taken->W_F2_P2->F2_P2_free->R_P2_F2
->R_P2_F1->done2->(thinking2)

```

Figure 3.25, Result of the analyser for the dining philosophers model

The circular chain 'thinking2' is the only chain available, except for the starting chain. So the optimised model cannot be implemented directly using the results, like the production cell model.

When comparing the top chain with the graphical model, it becomes clear that not all processes get activated and finished during runtime execution. This behaviour is the result of the implementation of the parallel construct of the CT library. The processes are not really parallelly running, but get scheduled using the scheduler of the CT library. This scheduler prefers to restart the last finished process if possible, resulting in only one philosopher being able to eat and the other two suffering of starvation.

So the runtime analyser can be used to detect badly constructed models, especially since these processes also are visible in panel 3 of Figure 3.2.

A solution to the dining philosophers dilemma is to assign a butler. A hungry philosopher asks this butler for his forks. The butler grants permission when both forks are available, before granting permission the butler reserves the forks. Now the philosophers will not starve to death, assuming none of the philosophers is greedy and will not release his forks, since they always get both forks when permission is granted.

The corresponding model and the analyser results are shown in Figures 3.26 and 3.27. The philosopher and the fork sub models are slightly changed, since they have interaction with the butler as well, but not extensively and therefore are not shown.

The results show that the model is better constructed and all philosophers are thinking and eating as they should.

Plotter model

The previously analysed models are not real controllers, but demonstrate the different results of the analyser. The plotter model, shown in Figure 3.28, is an actual controller model able to control a plotter.

Normally the linkdrivers act as glue between the model and the hardware, but for the analyser they are implemented to always accept data from the channel or to always present data to the channel. So the model is able to run without real IO. Besides the change of linkdrivers everything is the same as is would be when controlling the actual plotter.

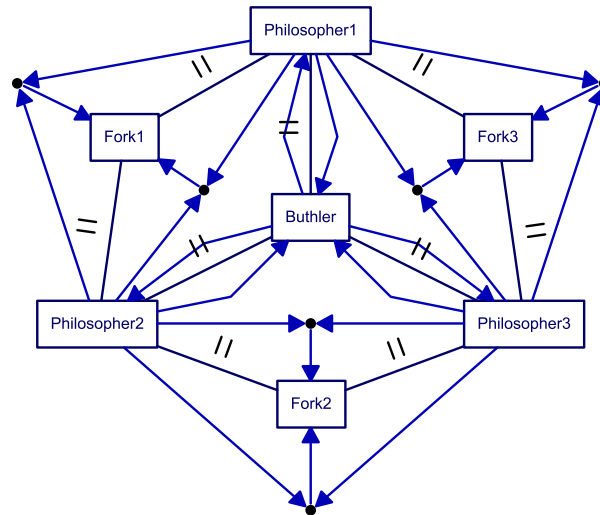


Figure 3.26, Dining philosophers with butler

```

R_F2_R->F2_released->W_P3_B->R_B_P3->W_B_P1->R_P1_B->W_P1_CF1->R_F1_C->F1_claimed
->W_P1_CF3->P1_eating->R_F3_C->F3_claimed->W_P1_RF1->R_F1_R->F1_released->W_P1_RF3
->P1_thinking->R_F3_R->F3_released->W_P1_B->R_B_P1->W_B_P2->R_P2_B->W_P2_CF2->R_F2_C
->F2_claimed->W_P2_CF1->P2_eating->(R_F1_C)

R_F1_C->F1_claimed->W_P2_RF2->R_F2_R->F2_released->W_P2_RF1->P2_thinking
->R_F1_R->F1_released->W_P2_B->R_B_P2->W_B_P3->R_P3_B->W_P3_CF3->R_F3_C
->F3_claimed->W_P3_CF2->P3_eating->R_F2_C->F2_claimed->W_P3_RF3->R_F3_R->F3_released
->W_P3_RF2->P3_thinking->(R_F2_R)

[start]->W_B_P1->R_P1_B->W_P1_CF1->R_F1_C->F1_claimed->W_P1_CF3->P1_eating->R_F3_C
->F3_claimed->W_P1_RF1->R_F1_R->F1_released->W_P1_RF3->P1_thinking->R_F3_R
->F3_released->W_P1_B->R_B_P1->W_B_P2->R_P2_B->W_P2_CF2->R_F2_C->F2_claimed
->W_P2_CF1->P2_eating->(R_F1_C)

```

Figure 3.27, Result of the analyser for the of dining philosophers with butler model

The motion sequencer uses a provided file to create a motion path for the pen. The motor controllers block contains a 20-sim model to control the X, Y and Z motor of the plotter. The safety block contains safety checks and is able to disable the X, Y or Z motor signal to make sure no unsafe situations occur. The last block, Scaling, scales the X, Y, Z and VCC signals within expected value ranges so the plotter received to correct signals.

After the runtime analyser finished the analysis, the results visible in Figure 3.29 are obtained.

Even such a big model has a defined running order, changing the input file does not change this result. When the complete input file is processed the executable exits and the analyser finishes the analysis. The duration depends on the amount and type of the drawing commands in the input file.

It is virtually impossible to validate the results. From ‘[start]’ to ‘WRITERZ’ the order seems reasonable. After that part the optimised channel effects start to play a role and ‘HPGLParser’ is finished for the second time, even before the rest of the model has been finished.

These so called optimised channel effects are the result of a channel optimisation in the CT library (described in (Hilderink, 2005) section 5.5.1). This optimisation only allows context switches when really needed. So for example after reading a result from the channel, the reading process stays active and could try to read a second time. This second time the channel will be empty and the reader blocks and the writing process will be activated again. However, before the reader tries to read the channel, other non-blocking processes could have been finished and put in the chains. This results in hard to explain analysis results.

Determining which chain is started when, is not possible from the analysis results themselves. Using the stepping option it is possible to determine that the order of chains is as shown in Figure

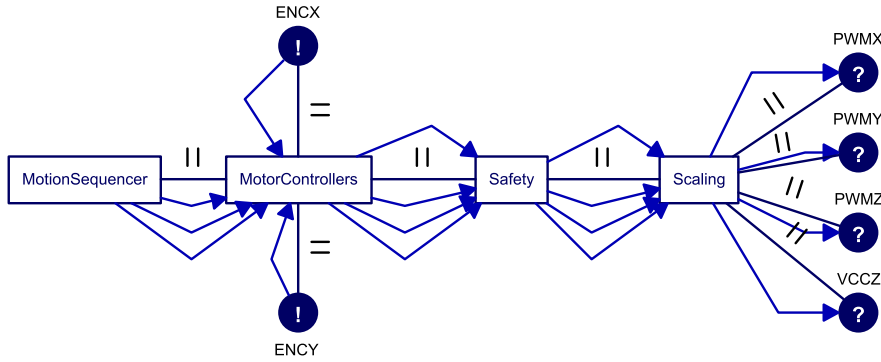


Figure 3.28. Plotter model

```

READER17->WRITER4->READER16->WRITER3->READER13->WRITER2->READER8->WRITER1->READER12
->READER11->READER10->READER9->WriteToTimer->Safety_X->READER14->READER15->Safety_Y
->Safety_Z->READER1->READER2->READER3->WRITERX->PWMY_Safe_WRITER->READER19
->PWMZ_Safe_WRITER->READER20->PWMX_Safe_WRITER->READER18->READER4->WRITERY
->DoubletoShortConversion->WRITER11->WRITER12->WRITER13->READER5
->LongtoDoubleConversion->Controller->WRITERZ->(READER21)

READER21->DoubletoBooleanConversion->WRITER14
->VCCZ_Safe_WRITER->(READER17, HPGLParser)

WRITER1->READER8->WRITER2->READER13->WRITER3->READER16->WRITER4->READER17
->WriteToTimer->READER12->READER11->READER10->READER9->READER1->READER2->Safety_X
->READER14->READER15->Safety_Y->Safety_Z->READER3->WRITERX->READER4
->WRITERY->PWMY_Safe_WRITER->READER19->PWMZ_Safe_WRITER->READER20->PWMX_Safe_WRITER
->READER18->READER5->LongtoDoubleConversion->Controller->WRITERZ
->DoubletoShortConversion->WRITER11->WRITER12->WRITER13->(HPGLParser)

HPGLParser->(WRITER1, READER21)

[start]->HPGLParser->WriteToTimer->READER1->READER2->READER3->WRITERX->READER4
->WRITERY->READER5->LongtoDoubleConversion->Controller->WRITERZ->(HPGLParser)

```

Figure 3.29. Result of the analyser for the plotter controller

3.30. The last ‘HPGLParser’ references back to the first one and the loop is complete. This complex order also is a result of the channel optimisations.

```

[start]->HPGLParser->WRITER1->HPGLParser->Reader21->READER17->READER21->(HPGLParser*)

```

Figure 3.30. Execution order of the chains

3.5 Conclusions

First of all the runtime analyser seems to work as expected. It is able to analyse most (pre-compiled) models, only rare situations are known for which the analyser will stop prematurely.

The sets of rules are defined by using relevant tests containing most common situations. The defined rules are sufficient for deterministic models and for simple non-deterministic models as well. However, the rules are not proved to be complete. When new (non-deterministic) situations are analysed it might be possible that new rules are required.

Currently, the results of the analysis are based on a single thread scheduler. When multiple-cores are used and the scheduler is able to schedule for multiple threads the analysis results are undefined. Processes will be running really parallel and received state changes are originating from unknown threads. To solve this problem the received state changes should be accompanied by thread information. The analysis algorithms can construct the chains for each thread separately using this extra information.

Another usage possibility of the the current runtime analyser tool would be post-executing analysis. The executable is executed on a target system first and the trace is stored in a log. A small tool could be created to replay the logged trace by using the animation protocol to connect to the

runtime analyser tool. The tools would not be able to notice the difference and it becomes possible to do analysis for models which runs on their target system.

Even though the results are likely to become very complex, usable information can be obtained as shown by the usability tests. A simple result to obtain is whether all processes are used at runtime. Processes which are unused will not become part of the model tree. Of course these processes might become active depending on external inputs, but this information can be used to take a closer look at least.

A more complex usage of the results is to actually implement them. Using the chains it becomes possible to create big processes having the processes of each chain in them. By replacing channels, which stay in the same process, with variables the processes will not block and also become even simpler. These steps result in an optimised model which requires less system resources and runs faster.

The analyser also can be used to check a model against its expected behaviour using the unused processes panel. When processes are unused this might indicate that an alternative channel is behaving unexpectedly. Or that a recursion is infinitely looping which might not be unwanted. Of course when using the analyser this way, the unused processes also might indicate that not all situations are tested. Like a safety check might incorporate processes to run when unsafe situations occur. This situation might never occur when just analysing a model.

4 Choice of the model input mechanism

In order to read the gCSP2 models a suitable input mechanism should be provided. The current gCSP files are stored using the Extensible Markup Language (XML) (World Wide Web Consortium, 2008) as stated before in Section 2.3. This is a well-known and a well-defined format and is a good basis to store the gCSP2 models in. A lot of tools are available to read and parse these files, Section 4.1 describes four possibilities to read XML files using these tools.

It is also possible to create dedicated gCSP files and accompanying readers using a scanner-parser combination. A scanner, or tokeniser, is a program which recognises lexical patterns in a stream of characters. A parser converts the tokens, generated by a scanner, into a data structure. Scanners and parsers are quite complex and therefore generators are available. The two best known scanner-parser generators are described in Section 4.2.

The final section presents the conclusion and selected input method used in the rest of the project.

4.1 XML readers

A XML reader is a piece of software which is designed to read XML files. Mostly they are part of a XML library which is also able to write XML files and to present the XML files in a data structure with all kind of usage support methods. A general disadvantage for XML readers is that they are bloated, they provide lots of functionality of which only a part will actually be used.

This section describes three software libraries which are able to read XML files. The first type consists of the generic XML libraries. These generic XML libraries can be found on the Internet in a lot of varieties. Next XMLSpy is described. This is a complete software environment to create XML files and files related to them. Finally the Eclipse Modeling Framework is described, this is a plug-in for the Eclipse environment.

4.1.1 Generic XML library

Many generic XML libraries can be found on the Internet and many flavours are available each with their own advantages and disadvantages. One of the most well-known libraries is MSXML, which is provided within the Windows operating system. The latest version is MSXML 6.0 (Microsoft Corporation, 2008). Other well-known XML libraries is Libxml2 (Veillard, 2008) and TinyXML (Thomason, 2008), which are available on many platforms and quite popular in the open source scene.

Depending on which flavour is picked, extensive documentation might be available, which could result in an easy to use library. As the term ‘generic’ already implied it is not possible to use a meta-model to fit the model data in. Any XML tag is allowed, which obviously is a disadvantage.

4.1.2 gCSP

The current version of gCSP also stores its model using XML files, so it could be used to read the XML files as well. The main reason for writing a new meta-model is because of the complexity and messy implementation of the gCSP files. The gCSP code is not very clean, since a lot of patches are added over the years. It is based on the Java XML serialiser and thus has the same advantages and disadvantages as the generic XML libraries.

4.1.3 XMLSpy

XMLSpy (Altova, 2008) is a XML development environment, it allows developers to create, modify and transform XML related files. Using XMLSpy it is possible to create, by graphical means,

a XSD (XML Schema Definition), which describes the syntax of the XML file. This XSD file can be used to generate Java, C++ or C# code.

The generated code is mainly a data storage structure, with a provided interface to a generic XML library to load and store its contents. Some of the supported libraries are MSXML, Java API for XML Processing (JAXP), and Microsoft System.XML. The used library is mainly dependent on the used programming language. So the only real advantage of using XMLSpy over a generic XML library seems to be the graphical way of defining the syntax or so called meta-model of a XML file. The generated code makes use of the generic XML libraries. So the advantages and disadvantages of generic XML libraries are also valid for a XMLSpy solution.

4.1.4 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) (The Eclipse Foundation, 2008) is a plug-in, of the Eclipse environment, which is an open source development environment. EMF provides support for modelling models, so it can be used to create meta-models. After a meta-model is created, the EMF is able to generate code. This code defines the meta-model and provides methods to store, read and modify the model. An available (de)serialiser provided with EMF, supports XML file formats. A major advantage is the fact that it is possible to modify the code without losing the modifications when the meta-model code is regenerated.

A small disadvantage of Eclipse and EMF is the complexity of the environment and all of its features. Before one is able to fully use it quite some experience is required, fortunately lots of documentation and tutorials are available.

4.2 Scanner-Parser generators

As stated before, it is also possible to read the XML files using a scanner-parser combination. Both are pieces of software, which can be combined into a single application. The combination of both is able to read data and convert it into another set of data. In this case from a gCSP file to a data structure in the memory. Actually the XML readers described above also contain some sort of scanner-parser combination to read the files, and to write the contents into the memory. However, using a custom created combination has some advantages. Building these would be a lot of work, thus automated generators are available.

These generators are fed with a language definition and generate the required code to read a stream of characters of this specified language. First the stream of characters is converted into a stream of tokens by the scanner. These tokens consist of characters which together are a building block of the language. This stage is called the scanning, or lexing, stage and is followed by one or more parsing stages. During this stage the tokens are combined into new tokens and finally the obtained information is stored. Without the scanning stage the process would be too complicated, hence the two stages.

As a result the obtained software is only able to read the models and put it into a dedicated structure. It results in less bloated software in comparison to the generic XML libraries. Since these libraries are able to perform all kinds of operations on the XML specific data structures. Another major advantage is the possibility to have multiple parsing stages, so it will be possible to implement the required algorithms in the parser stages. This will seamlessly link with reading the documents.

A disadvantage of using a scanner-parser combination is the fact that it only is able to load a model. After modifying a model or generating a new one it has to be stored again, which would require another piece of software specially made for this task.

Different scanners and parser use different algorithms to parse the character streams. Some may be very good in finding the lexical patterns, while others are sufficient for simple languages only, some might be able to backtrack when a dead-end in the pattern is found, while others will generate

an error or warning. All these kind of differences influences the complexity and speed of the scanners and parsers.

4.2.1 Flex and Bison

The most well-known parser generator combination probably is the Flex (The Flex Project, 2008) and Bison (Free Software Foundation, 2008) combination. Both are open source programs and around for many years.

Bison is a parser generator that converts an annotated context-free grammar into an LALR(1) or GLR parser for that grammar. LALR is short for **L**ook **A**head **L**eft-to-right **R**ightmost and is a specialized form of the LR parser. This is a type of parser that reads input from **L**eft to right and produces a **R**ightmost derivation. The rightmost derivation, contrasted by leftmost derivation, is a way how the tokens and sub-tokens are linked to each other. Look ahead parsers take more of the parser context into account and are therefore more specific compared to non-look ahead parsers. The number in the parentheses tells how many unconsumed tokens, the look ahead property, are allowed. The higher this number the more complex the parser will become, since more combinations will apply.

GLR is short for **G**eneralized **L**eft-to-right **R**ightmost and thus also a LR parser. Generalised means that it allows for ambiguous grammars in contrast with the LALR parsers. Ambiguous grammars have constructs which can be interpreted in multiple ways. An example is the following (partial) line of C: `x * y;`. This can be read as 'x multiplied by y' or as 'a declaration of y which is a pointer of the type x'. Depending on x being a variable or a type declaration, the first respectively the second should be chosen by the parser. It might be clear that ambiguous languages are very complex to parse.

Because of their familiarity a lot of documentation and examples can be used. So it is very easy to start using them. A disadvantage is the lack of a (graphical) interface since they are command line tools. The output of this combination is specialized on C/C++ and converting this to Java incorporates a loss of code quality of course.

4.2.2 ANTLR

ANTLR (Parr, 2008), which is short for **A**nother **T**ool for **L**anguage **R**ecognition, is able to generate a scanner-parser combination. Having scanners and parsers generated from the same input has a nice advantage that strings being used in the parser parts automatically are created as tokens in the generated scanner. ANTLR makes use of a so-called StringTemplate, created by the same author, which enables the possibility to generate code for a variety of target languages. It is even possible to add custom or unavailable languages.

The generated parser is a LL(*) type parser. Where LL is short for **L**eft-to-right **L**eftmost, which is the opposite of the generated bison parser. The * indicates that it can use a variable amount tokens to look ahead depending on the current language rule being processed. Because of this look ahead type it is not possible to use lookup tables and additional code is generated to replace them. In order to allow even more complex language definitions ANTLR also supports backtracking. This is useful when sub-rules are used and the parser has to fit the available tokens to the processed rule and to the used sub-rules. It can try several combinations without needing huge look ahead combinatorics, since it will just take a step back and try another possibility when the current one does not fit.

Except for the modern parser generator possibility, a variable look ahead parser generator is quite unique, the error reporting abilities of ANTLR is quite good as well. Certainly compared to other generators, the readability is much higher and it will be easier to find the, mostly complicated, errors for the user. Because the generated parsers use code instead of lookup tables, the generated parser is much better to debug as well. Because the debuggers are created to follow code and

not table entries. Is it also possible to create the ANTLR rules using a graphical editor called ANTLRWorks. It shows the rules as flow diagrams. It also is able to interpret input and use a simple debugger to show the steps taken to produce the parsed output.

4.3 Conclusion

Table 4.1 shows an overview of the advantages and disadvantages of the readers, most columns are clear but some vague columns might need a short description: The 'Java' column indicates that the reader (and meta-model if applicable) can be created in Java, the 'meta-model' column that the reader has a meta-model to check the files with and 'complexity' indicates how complex or difficult the creation of the reader is.

	Java	Code cleanliness	Meta-model	Editor / Environment	Open source / Free	Complexity
Generic XML library	+	-	-	¹	+	+
gCSP	+	-	+/-	¹	+	-
XMLSpy	+	-	+	+/-	-	+
Eclipse EMF	+	-	+	+	+	+/-
Flex & Bison	-	+	+	-	+	-
ANTLR	+	+	+	+/-	+	-

¹ Since no meta-model is available, an editor is not required

Table 4.1. Requirements for the readers

The desired language to read the gCSP2 models is Java, because the code should be reusable by the future version of the gCSP editor. Since the new editor is going to read the models, a simple reader is required as well so it will be easy to update the reader when the model gets extended. These two restrictions drop the scanner-parser generators, especially since the complexity requirement is weighted heavily.

Eclipse also is rated as mediocre complex, but it is expected that it is much easier to learn to use eclipse than a scanner-parser generator. So the +/- rating would lean more to the '+ side'.

It also is nice to have an editor or environment to create the gCSP meta-model, especially if the meta-model editor could be part of the code development environment. So the generic XML libraries will be dropped from the optimal choice, XMLSpy has a +/- rating because the environment only generates code which needs to be used in another environment. This leaves Eclipse EMF being favourable compared to XMLSpy.

XMLSpy is a very nice and easy to use editor, it is very user friendly and attractive. Maybe the user-friendliness is a little over the top. Another drawback is the fact that it is commercial software, a free 30-day trial can be obtained, but afterwards the user is expected to pay. These arguments combined with the badly rated environment leaves the choice to Eclipse EMF.

Eclipse is nice and easy to use environment, especially when one gets used to it. A major advantage is the wide range of available plugins. A combination of Java, EMF and a version control feature creates a powerful environment for code creation of the meta-model and the analysers. So Eclipse in combination with EMF is chosen to create the meta-model in.

5 Meta-model creation

As mentioned before a meta-model is a model to describe or create models with. Chapter 4 concluded that EMF would be a good choice to create the meta-model with. During the creation of the meta-model, the fact that it will be used by the model analyser only at the moment is kept in mind: The implemented model information storage only consists of the basic and most used objects. For example, guards cannot be stored yet, because they are complex and not used in most models.

The first section in this chapter contains information about general EMF projects. Next the new gCSP meta-model is described. Followed by the description of the implemented unit testing suite and a converter tool to convert old gCSP models into the new gCSP2 models. The chapter ends with conclusions about the meta-model.

5.1 EMF projects

The meta-model is created as an EMF project, which requires some features to be installed. During the project the following features are installed, besides the ones enabling Java language support:

- Eclipse Plug-in Development Environment (v3.4.0)
- EMF SDK - Eclipse Modeling Framework SDK (v2.4.0 and v2.4.1)
- Graphical Modeling Framework SDK (v2.1.2)

The Graphical Modeling Framework SDK is only required to create a UML diagram from the meta-model.

An Eclipse project can be build by extending other projects, an EMF project is such an extended project. It is based on an Eclipse plug-in project with two extra files stored in a model folder. The Eclipse plug-in project itself is again an extension, which is based on a Java project. The two extra files are

- An ecore file, containing the meta-model
- A genmodel file, containing the information to generate code for the ecore file

More information about the available files in the EMF project can be found in Appendix A.

5.2 Meta-model requirements

The current gCSP model is not really described in a meta-model resulting in complex files. The new gCSP meta-model, called gCSP2, should be less complex and well structured.

One of the reasons for the complexity of the current files is caused by a lot of double objects. This undesired situation is the result of using the standard Java XML writer, which is unable to write references. For example, Figure 5.1 shows a simple model containing two processes: Writer and Reader which are connected by a data channel. This channel contains references to the Writer and the Reader process, since it should remember which processes are connected to it.

The current gCSP file format would have Reader and Writer included twice, once as children of the parts list and once as children of the data channel. A more correct situation would include both Reader and Writer in parts, since they are parts of the model. It would create references to the processes in the data channel object. The gCSP2 meta-model should implement this behaviour, to reduce the complexity of the files greatly.

Another requirement of the gCSP2 meta-model is that it should be able to contain all necessary data for the Model Analyser. All data which is not used by the analysis will not become a part of the meta-model, because it is pointless at this stage to define structures for it. For example, at the start of the chapter guards are mentioned and will not be stored. Graphical information, like coordinates, user-draw areas and object colours, will not be stored as well.

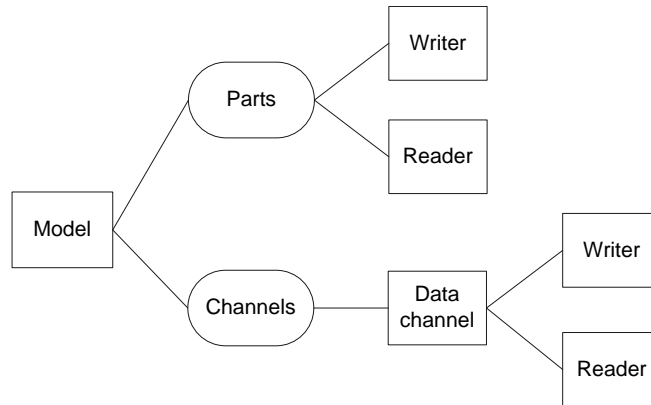


Figure 5.1, Simple model with references

5.3 The gCSP2 meta-model

Figure 5.2 shows the created meta-model, which is suitable to be used in the Model Analyser. An UML diagram of the meta-model can be found in Appendix B.

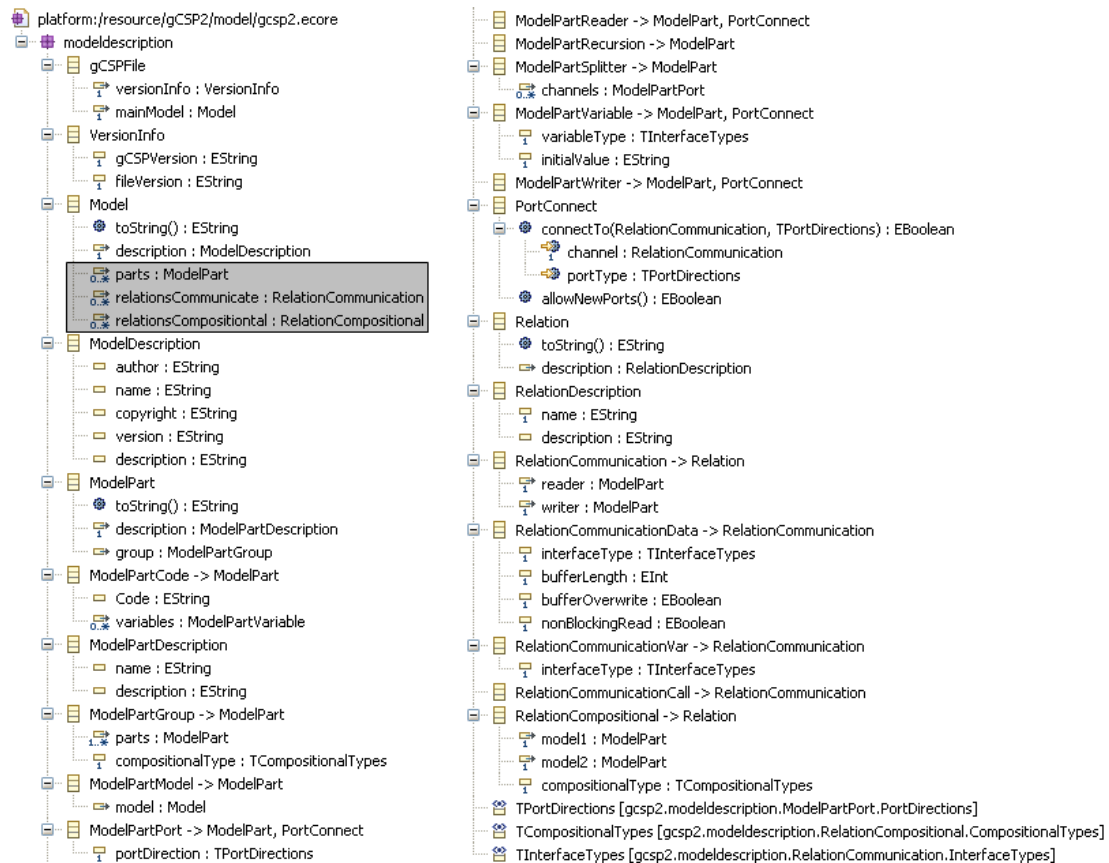


Figure 5.2, The created gCSP2 meta-model

It can be seen that a *Model* has three important members, shown in the transparent rectangle:

- Parts (like *Readers*, *Writers*, *Groups*, (other) *Models*)
- Communication relations (*Data* and *Variable* channels)
- Compositional relations (like *Parallel* or *Sequence* constructs)

The *Model* and all other meta-model classes, have their own description class to store basic information of the object like the objects name, description and author. Even though each type of object has its own dedicated description class, it is tried to keep the fields of these classes similar

in order to keep things as simple as possible. The generated model code keeps track whether fields are changed or not and during serialisation of the model only the changed values are used and the files are kept as small as possible.

As stated before a *Model* contains parts, described in the *ModelPart* class. A part is an abstract object, which incorporates that it is unknown what the type of the part is, but the general part information is available. This is done to make all available object to be placed in a *Model* of the same type, which reduces complexness. In the current situation only the description class is available, but it could be extended with other general classes. Like a class to store the location and dimensions of the model part on the graphical model view, since all parts have a location and dimension. The different types of parts, like code blocks, readers and writers, have their own classes. All based on the *ModelPart*, but with added information containing the type specific details. The *ModelPartCode* class, for instance, has additional fields to store the variables and the source code. In contrast to the *ModelPartGroup* class, which has extra fields to store the grouped parts.

Like the original gCSP meta-model, it is possible to create, so called, sub-models, which are also based on the part class and behave similar. For the model it is yet another part to describe its functionality. However, the model part class contains a link to a model class with its own parts and relations available. This model class is able to behave as an independent model like any other model and could be stored like any other model. This implementation might become the basis of a multiple instance sub-model system, which is not yet present in gCSP. Having such a system allows a sub-model being used multiple times, which saved work to maintain all sub-model separately. It also enables the use a libraries containing complex sub-models which can be easily reused.

Most likely the parts have relationships to each other, for example a reader is connected to a writer by a channel. Therefore a relation class is created, which also is an abstract class and the basis of the different types of relations available. The *RelationCommunication* and the *RelationCompositional* classes are specifications of the relation class defining two available types of relations in CSP.

The communication relation class is also abstract, since a communication relation is not specific enough. Since each communication relation (a channel) has two ends, the abstract class adds two fields: a reader and a writer field. Two communication relations are implemented, the data channel and the variable channel, both extend the abstract communication class with fields to store the channel specific properties.

5.4 Testing of the model

In order to test the model, for example to check if it can be stored and loaded without problems, a unit testing environment is used. A unit tester is a tool which is able to perform defined tests on the software very quickly. After modifying the software, the tests can be performed to check whether the modifications did not break the desired behaviour of other parts. The Eclipse interface of junit, such a unit testing suite, is shown in Figure 5.3.

Methods can be created to perform the tests, these methods will be executed when junit is started. A test method typically creates the object to test and performs simple tests, like 'is the value of variable x equal to the value of variable y' or 'is the reference to another object correctly set'. If such a test fails, it will send a message to the IDE, shown in the figure. The developer sees amount of failures and the corresponding messages and locations of the failed test.

The unit testing environment got used during the creation of the meta-model and provided useful feedback while removing bugs.

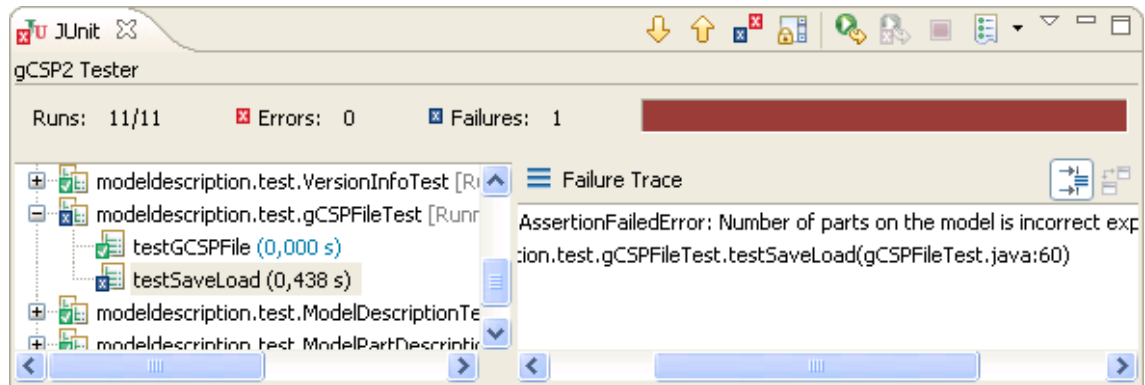


Figure 5.3, jUnit integrated in Eclipse showing the results of a test run

5.5 Converting gCSP to gCSP2

It is possible to create gCSP2 files using the gCSP2 meta-model by hand, but this is a lot of work. Besides this, it might be too hard at all to manually convert complex models without adding errors. So a converter is created, which is able to load gCSP files and write them back in the gCSP2 format. Internally it reads the old data structure and places the information pieces into a new model. Only the data required for the Model Analyser is converted. Things like guards, call channels, linkdrivers or 20-sim code blocks are either ignored or converted into something similar but more simple.

5.6 Conclusions

The created model is able to contain all model specific information, parts and relations are available to describe the functionality of a model. The model part objects can be used to create sub-models so hierarchy can be obtained as well. Advanced parts and relations, like link drivers or call channels, are not available yet. Not all graphical information can be stored in the meta-model, the description classes contains textual information. Information about locations and sizes of the parts and relations can not be stored yet.

The created meta-model will be suitable for the model analyser, since enough model information is present to do analysis on. The missing advanced parts are not yet used in the current model analyser and their lack in the meta-model is no problem.

A minor drawback of using EMF is the (de)serialisation of user defined types. Standard Ecore data types do not provide a Point or Color data type, like Java does. In order to still be able to use them, Ecore allows the definition of new data types based on Java data types. Three of these definitions can be found on the bottom right of Figure 5.2. For serialisation of the values of these data types the `toString()` method is used, which is always present for Java classes. The problem occurs when deserialisation is required to load the value, which uses the generated method `createPointFromString()` in the case of the Point type. This method uses Java reflection to find a constructor which has a single string parameter. If none are present, it continues searching for a method `valueOf()`. First of all it fails and secondly using reflection is considered unsafe and slow. So it is required to overwrite the generated methods `convertPointToString()` and `createPointFromString()` so the serialisation and deserialisation of the values uses the same format.

In order to use the meta-model in gCSP2 the missing information needs to be added to the meta-model. A critical view is required to keep the meta-model simple. It is recommended to split the model specific information and the graphical information, so the model easily can be used by tools because to graphical data can be ignored.

6 Model analyser

This chapter describes the developed model analyser tool. First a short introduction is given, followed by an explanation of the algorithm parts and the testing and results of the same models reviewed by the runtime analyser in Chapter 3. The chapter ends with conclusions about the model analyser and its usability.

6.1 Introduction

The runtime analyser consists of several algorithm blocks connected to each other, shown in Figure 6.1. For each block the corresponding section describing it is shown within the parentheses.

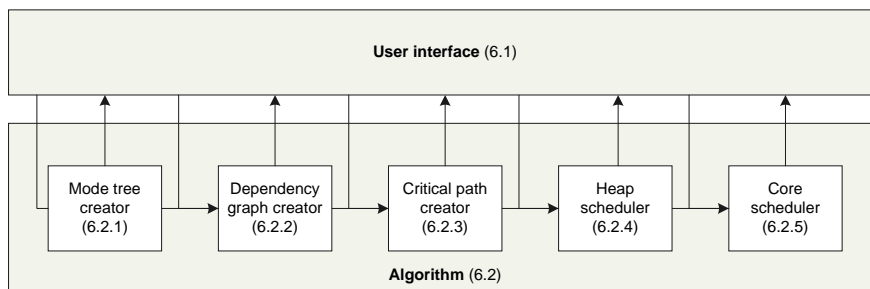


Figure 6.1, Algorithm steps

During each step of the algorithm the user interface is updated to reflect the current situation. This allows for visual stepwise execution of the algorithm steps.

The user interface of the model analyser is shown in Figure 6.2. A full description of the user interface can be found in Appendix H.

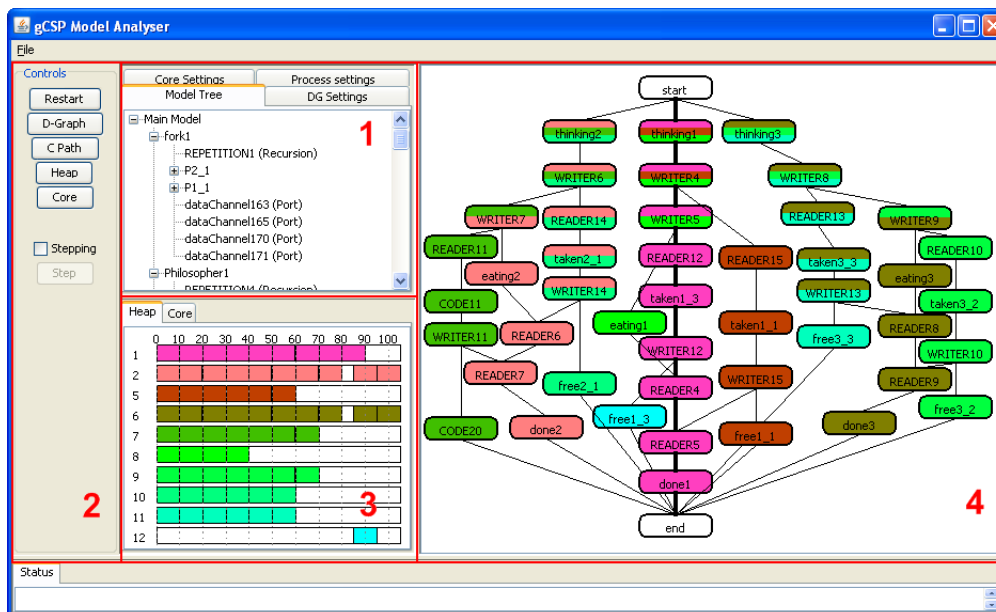


Figure 6.2, User interface showing the results of an analysed model

Panel 1 is a model structure tree showing all parts of the model and its sub-models with their model parts. Below panel 3 is visible, containing timing information of the scheduled heaps and cores and at the right in panel 4 the dependency graph with its critical path is visible. Panel 2 contains a couple of buttons to restart the analyser after settings got modified. These settings are available in panel 1 at the various tabs.

The buttons below the restart button in panel 2 can be used to skip steps of the algorithm. When a setting is changed only a part of the algorithm needs to be rerun depending on the changed setting. This may save rearranging the dependency graph when that step is skipped. On slower computers or when using a big model, skipping algorithm steps might save valuable time.

The User Interface also supports rearranging the dependency graph by selecting and dragging (groups of) processes. It is able to store and reload the rearranged processes in the dependency graph. It is also able to export the created views in PNG and EPS format and to change settings of the available cores and processes in order to influence the algorithms.

6.2 Algorithms

The algorithm created by van Rijn (1990) will be used. It is the most suitable and clearly explained algorithm found. The important parts are based on a set of rules that can be extended, in order to create an even more sophisticated algorithm. The original algorithm was created to be used for model equations run on transputers. The extended algorithm is usable for scheduling CSP processes on available processor cores or distributed system nodes.

The algorithm blocks, shown in Figure 6.1, are independent blocks chained together. So it is fairly easy to extend the blocks or to add a new one in between without the need to rewrite following blocks. Assuming that the data sent to the existing blocks stays compatible. This data is also used by the user interface to update its views after each step. This section will describe each separate step or block of the algorithm.

Unless stated otherwise, the model of Figure 6.3 is used to create the shown results in the following sections.

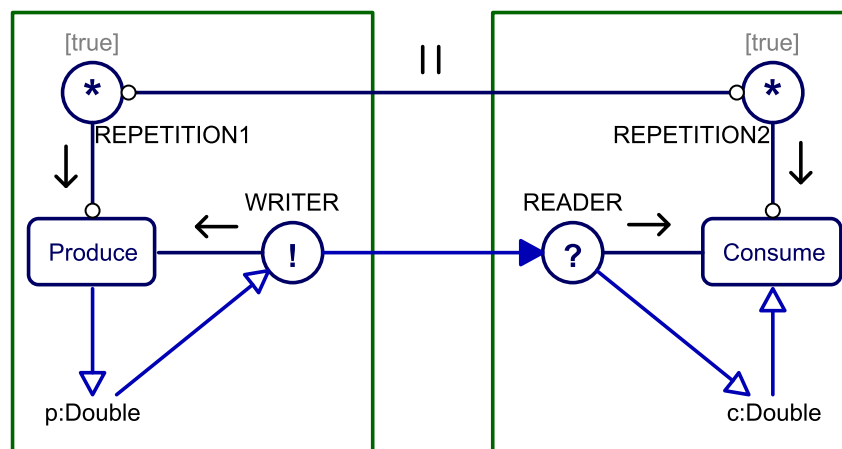


Figure 6.3, Exploded view of the ProducerConsumer model

6.2.1 Creation of the structure tree

The structure tree, shown in panel 1 of Figure 6.2, is only for informational use. The algorithm does not use it. This algorithm step is simple:

- It loads the selected model
- Builds the tree by recursively walking through all sub-models to add the available parts
- And sends the loaded model to the dependency graph step

6.2.2 Dependency Graph creation

The dependency graph is extensively used by the algorithm blocks to find and make use of the dependencies of the processes. Thus creating a good and complete dependency graph is very important.

The dependency graph block also recursively walks through the model and all of its sub-models. First to add all vertices which represent the available processes and a second time to add edges representing the dependencies. This results in a usable graph for finding predecessors and successors for each vertex, so the other algorithm blocks can find all dependencies when a vertex is processed. The graphical representation also uses the predecessors and the successors to place the vertices in a slightly ordered way.

The first step, adding vertices, is done by finding all processes and examining whether it is a vertex or not. The only suitable and supported processes are code blocks, readers and writers. The linkdrivers are converted by gCSPTogCSP2 (Section 5.5) as readers and writers, the 20-sim blocks become code blocks. The regular processes will be removed, since those only act as containers to group parts of the model. So by only using the three mentioned processes sufficient information is available to create the vertices.

After the first step two extra vertices are added: a start and an end vertex. These vertices are used to connect the beginning and ending processes to.

The second step, adding edges, is a lot more complicated. Since one part of those edges are derived from channels, which can be running through several processes and intermediate parts before getting to the actual process. The other part of the edges is derived from the sequential relations, which always stay in the same process. The created groups have to be taken into account to determine which and how many edges are derived from a single sequential relation.

Figure 6.4 shows an example dependency graph. All processes are taken from the Producer-Consumer model except for the two recursion processes. These processes are not real execution processes and therefore not required in the dependency graph. The vertices are created using the sequential relationships, except for the dependency between 'WRITER' and 'READER', which is derived from the channel.

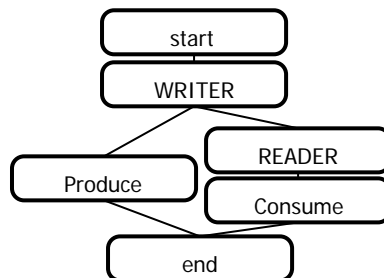


Figure 6.4, Dependency graph of the ProducerConsumer model

6.2.3 Critical path determination

The heap scheduling algorithm needs to know which path in the dependency graph is the critical path.

Each process has a weight which is a relative indication of the amount of execution time required by that process. Starting with the start vertex, the algorithm walks down through all paths to the end vertex. Each passed vertex gets the maximum weight of its predecessor added to its own weight.

When all vertices have their weight value assigned, the algorithm walks back to start using the preceding vertex with the highest weight value available. Each passed vertex is marked being part of the critical path.

Figure 6.5 shows the calculated weight values in each vertex and the critical path is indicated with the thick line. Each process has a weight value of 10 but these values can be modified if desired. It also is visible that the end vertex has to choose which predecessor to use and picked 'Consume'.

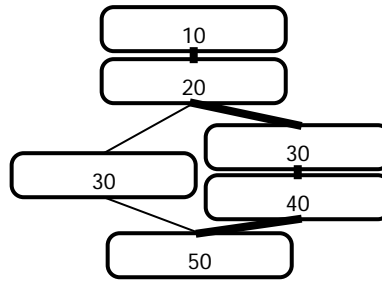


Figure 6.5. Critical path of the ProducerConsumer model

6.2.4 Heap scheduler

Heap blocks are required by the core scheduler to be able to schedule the processes onto cores. A heap can be seen as a group of processes closely related to each other. This definition allows for processes to be scheduled onto multiple heaps. The processes on the heaps have a heap timing set and can also be seen as a sequentially scheduled group of processes.

Since the heap scheduling algorithm is complex and comprehensive, the complete algorithm and its rules are explained in Appendix C. In general the algorithm puts the vertices of the critical path in heap1. The remaining vertices are grouped in sequential chains. Depending on communication costs, the length of these chains varies. Because copying a vertex and its predecessors onto multiple heaps might be cheaper than communicating the result to from one heap to another. The heap timing value assigned to each vertex placed on the heap is calculated by using the end time of the heap and the weight value of the vertex.

The result can be found in Figure 6.6: at the left is the dependency graph visible with the vertices coloured depending on which heap(s) they got placed. At the right bars are visible which represent the heaps. The vertices are placed on these bars in the determined chronological order.

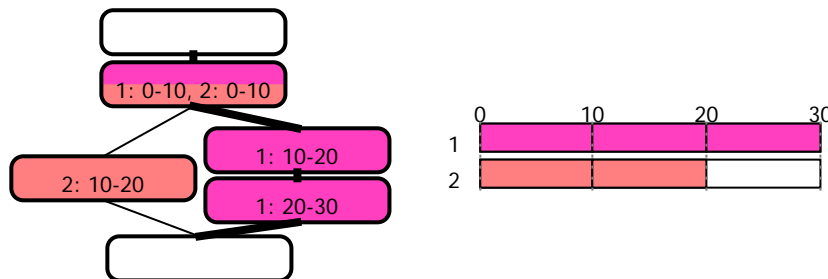


Figure 6.6. Results of heap scheduler

The texts inside the vertices have the ‘x: y-z’ format, with:

- x being the heap number the vertex is placed on
- y being the start time of the vertex on heap x
- z being the end time of the vertex on heap x

The start and end vertices are not scheduled, since they represent no real processes.

The figure also shows that ‘WRITER’ is scheduled onto both heaps. This is understandable since the communication weight is default set to 25 and the vertex has a weight of 10, so putting it on both heaps saves a weight value of 15, resulting in less workload for the cores.

6.2.5 Core scheduler

The last algorithm block is the core scheduler. This block schedules the heap blocks onto a number of available cores. A core can be seen as an independent unit, like a real core of a processor or a node in a distributed system. Depending on the amount of available cores and their relative speeds the scheduler tries to find an optimum.

Like the heap scheduler, the core scheduler is also very complex and comprehensive so the complete algorithm and its rules can be found in Appendix D. The algorithm is divided in two parts, an index block creator and an index block scheduler part. The heaps might be too complex to get scheduled properly. For example if two heaps are dependent on each other they cannot be scheduled, since all predecessors need to be scheduled first. The index blocks are defined in such a way that this problem will not occur anymore.

The actual scheduler consists of many rules to come to a single optimal scheduling result for each index block. A big distinction is made between heaps which are part of the critical path and the rest, to make sure that the critical path is scheduled optimally and the end time of this path will not become unnecessary long. The overall rules are designed to keep the scheduled length and thus the ending time as short as possible.

Figure 6.7 shows the results of the core scheduler in a same way as the heap results are shown. The dependency graph view shows the core schedule results and the view with the bars represents them in a chronological order.

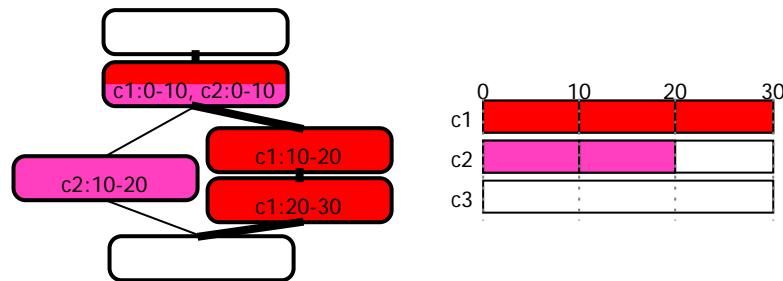


Figure 6.7, Results of core scheduler

Three cores, named c1, c2 and c3, are available in the figure. Their core names are visible in the dependency graph with the vertex timing behind it. The bars view also shows the core names with the vertexes placed on chronological order on the bar.

The figure shows that core three is unused, which is no surprise since only two heaps are available and in the end a complete heap gets scheduled on the same core.

6.3 Design and implementation

6.3.1 Algorithm

The algorithm is implemented by dividing the steps from Figure 6.1 into separate Java classes, to make sure it can be extended or reused easily. The processes have their own class to store their information. This class is filled by the first three algorithm steps and results in the dependency graph.

Because a process might be available on multiple heaps and cores, the heap and the core schedulers have their own class to store heap and core specific information in addition to the process specific information.

The algorithm classes should be able to run independently of the user interface. So it is easy to reuse them in another tool or integrate them with a designer application like gCSP.

6.3.2 User interface

The user interface is also build modular. All panels are acting highly undependable. As input they get the list of object they are interested in. A dependency graph panel gets the list of processes, the heap bar view the list of heaps and so on.

Events are used to communicate between the different panels. When the user clicks in a heap or core bar, an event is issued and received by the dependency graph so it is able to select the process

itself. The same mechanism is used when the dependency graph has changed its selection: an event makes sure that the process setting panel is also changed to the selected process.

The bar panels are simple extensions of a super class taking care of maintaining and painting the bars. The super class also takes care of exporting its panel to EPS or PNG format, so the actual implementation do not have to support it by them selves. So adding a new view again is an easy task. Same goes for the settings panel, which also are derived of a super setting class.

The algorithm might take some time before finishing when a large model is analysed. In order to keep the tool responsive the user interface and the algorithm are running in separate threads. During the calculations of the algorithm the user interface is updated with intermediate results so the user can start rearranging the dependency graph for example.

The threaded implementation also makes stepping possible. When an algorithm step is finished it is possible to stop its thread to see the result of the step. The user interface will continue the algorithm thread when the user clicks the step button.

6.4 Results

This section describes the results of the analyser. The setup is the same as the results of the runtime analyser described in Section 3.4. First a functional test is performed and next usability tests to see whether the results of the analyser can be used when a more realistic model is used. Since the same models will be used, no figures and explanations will be repeated in the following sections.

6.4.1 Functional test

As a functional test a simple producer consumer model is created, as shown in figure 3.16. This model was already used to describe the algorithm.

This section will describe the reaction of the algorithm on changed settings. First the amount of cores will be reduced to one, of which the result can be seen in Figure 6.8.

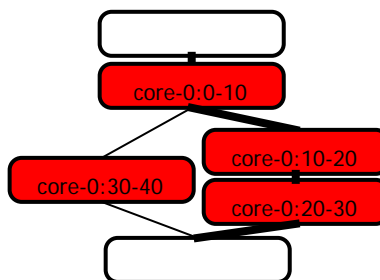


Figure 6.8. Results of the core scheduler with one available core

Both heaps now are scheduled on the same core, a couple of things should be correct now:

- ‘WRITER’ is available in both heaps, but of course should be scheduled only once.
- The timing of all processes should be correct. It is not allowed to have two processes with overlapping timing estimates.
- The dependencies of the processes should be honoured. A process should be scheduled after its predecessors even when those predecessors were scheduled on different heaps.

All of these constraints are met when looking at the figure. In fact these constraints are always met when the available settings are varied even more, like changing the amount of cores or the weight of processes, which is not showed here. So it can be concluded that the analyser is functional for the ProducerConsumer model.

6.4.2 Usability tests

The functional test showed that the analyser is functioning for simple models as expected. Now the more complex models are tested. The analyser should still produce functional results, as given in the list in the previous section. Since it is a lot of work to check all the possible situations, only a few interesting situations are described. Besides being functional, the results also have to be usable. So the complexity and the optimal solution of the results should be checked. Preferably for all possible situations, which of course is not realisable. So a selection of potential interesting results will be discussed.

Production cell

The first usability test is performed with the production cell with four equal cores available to schedule the processes onto. The result can be found in Figure 6.9.

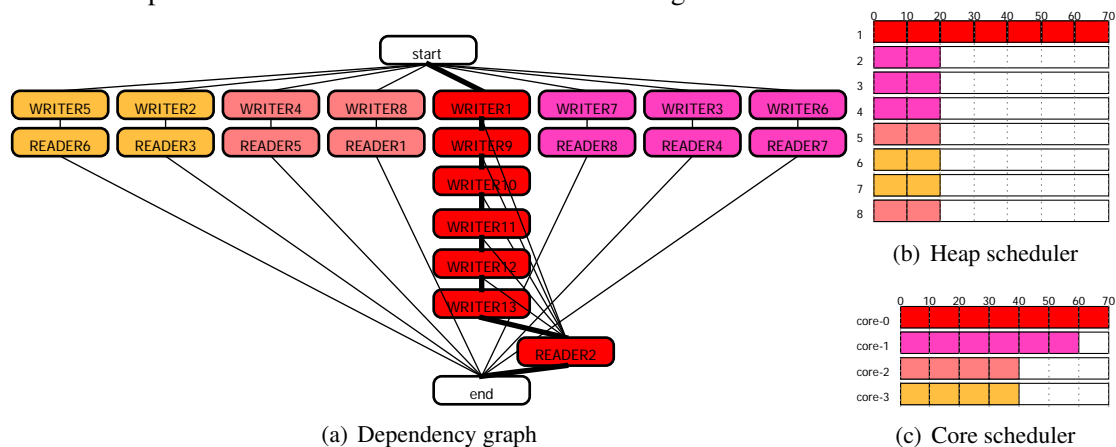


Figure 6.9, production cell results scheduled on four cores

The heap representing the critical path contains the processes of the BlockInserter combined with 'Reader2' of the InsertBuffer. All other heaps have a writer from one process and the reader of the following process. The reason for this layout of the dependency graph is the fact that all processes are placed in parallel constructs so only the data channels generate dependencies, resulting in writer-reader pairs.

The heap bars show the index block being separated by the thick vertical lines. It is also possible to colour the processes, of the dependency graph and the heap bars, depending on the index block(s) they belong to.

The core bars show that the critical path is scheduled on core-0 and the other heaps are equally divided onto the other cores, as far as possible. Resulting in the most optimal schedule. When one of the cores is made faster, the critical path is moved to that core, since it will result in a shorter schedule length and the other processes again are equally divided onto the other cores.

Dining Philosophers

To show the results of different settings, the examination of the dining philosophers model is done with some changed settings. Three cores are available, with core-0 twice as fast as the other two cores. The thinking processes have a weight of 40, compared to the weight of 10 of the other processes. The results of the dining philosophers model with the changes can be found at Figure 6.10.

Three parts can be distinguished. Each part consists of a philosopher and a fork. Shown in the figure by the dashed boxes. This seems weird because each philosopher interacts with two forks. When examining the model showed in Figure 3.24(c), an alternative compositional relation is visible. Alternative relations are not implemented yet, so basically the dependency graph is not complete.

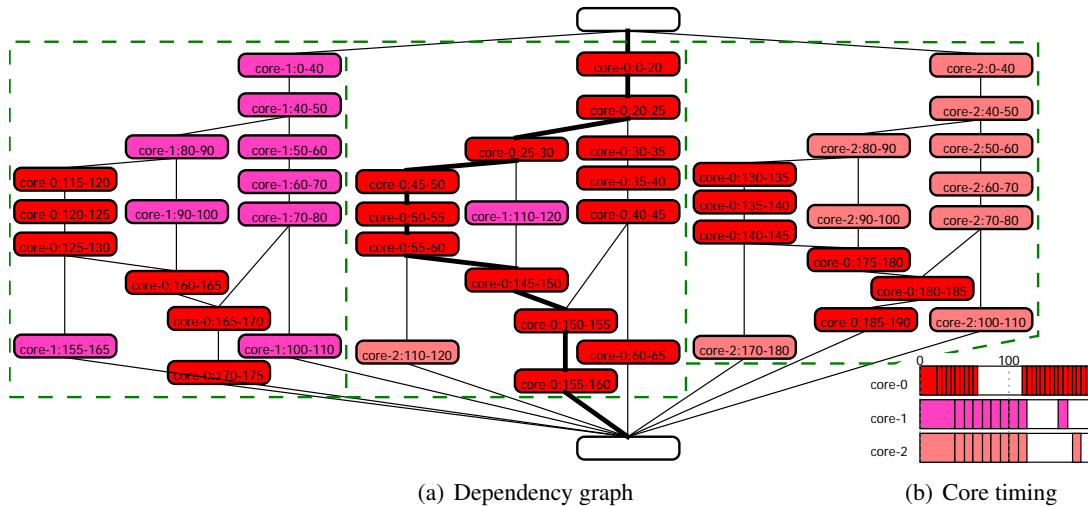


Figure 6.10, Dining philosophers results scheduled on three cores

Alternative relations are different compared to parallel and sequential relations. Parallel relations explicitly do not generate dependencies and sequential relations explicitly do generate dependencies. Alternative relations could be read as ‘one of the preceding processes should have been finished’. This is a sort of OR-relation instead of the AND-relation of the sequential processes. Currently, the algorithm cannot handle OR-relations and alternative relations are ignored for the time being.

Figure 6.10 is the result of higher weights of the thinking processes. Without these higher weights the thinking processes will be copied to multiple cores. This is a valid action since that is cheaper than communicating the results to the other cores. The higher weights prevent this behaviour, but also might cause a less optimal scheduler result as well.

The sub-optimal result can be noticed in the core timing bars, which have white gaps in the schedule of all cores. This indicates that the core will be idle for that period of time. The gap in the bar of core-0 is enlarging the schedule length, although the gaps in the other cores are also almost enlarging the schedule length. The gap in core-0 is caused by the left most process ‘core-0: 115-120’ which needs results from ‘core-1: 80-90’. The communication time of 25 makes the process start at 115 resulting in the gap, since the previous process ended at 60. Lowering the thinking weights will remove the gap, but processes get scheduled onto multiple cores now and the schedule length barely changes. Figure 6.11(a) shows the dependency graph with the processes being scheduled on multiple cores.

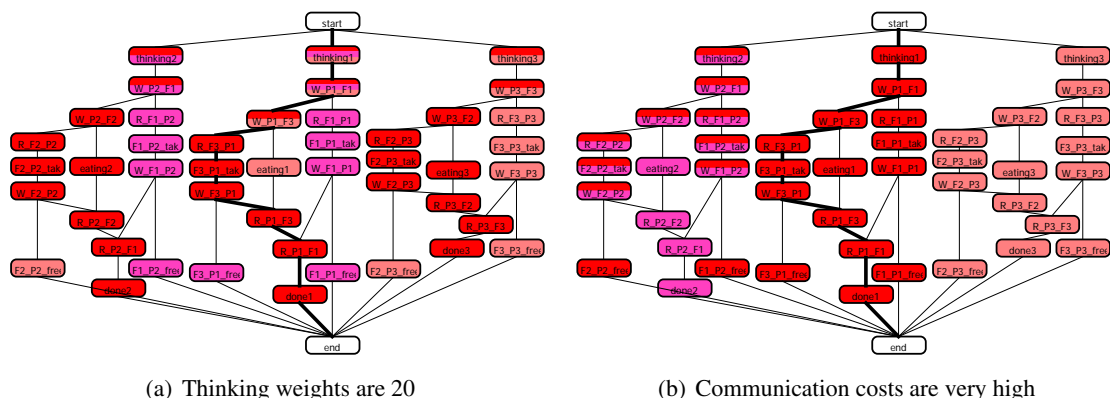


Figure 6.11, Other Dining philosophers results with changed settings

Another available option (whether it is better is questionable) is to make the communication costs higher. Indicating that the cores should be less dependent on the other cores. A disadvantage is that the scheduler is restrained in its possibilities. However, the result is more as one should expect: one core for each philosopher. Figure 6.11(b) shows this situation. The right and middle philosophers have their own core, the left one is scheduled on two cores, mainly because core-0 is twice as fast and has enough free time left. To execute two processes on core-0 nine additional processes have been copied to the core. The reason for copying is the high communication costs and immediately shows why this solution is questionable. When the core speeds all are equal, all philosophers will get their own core.

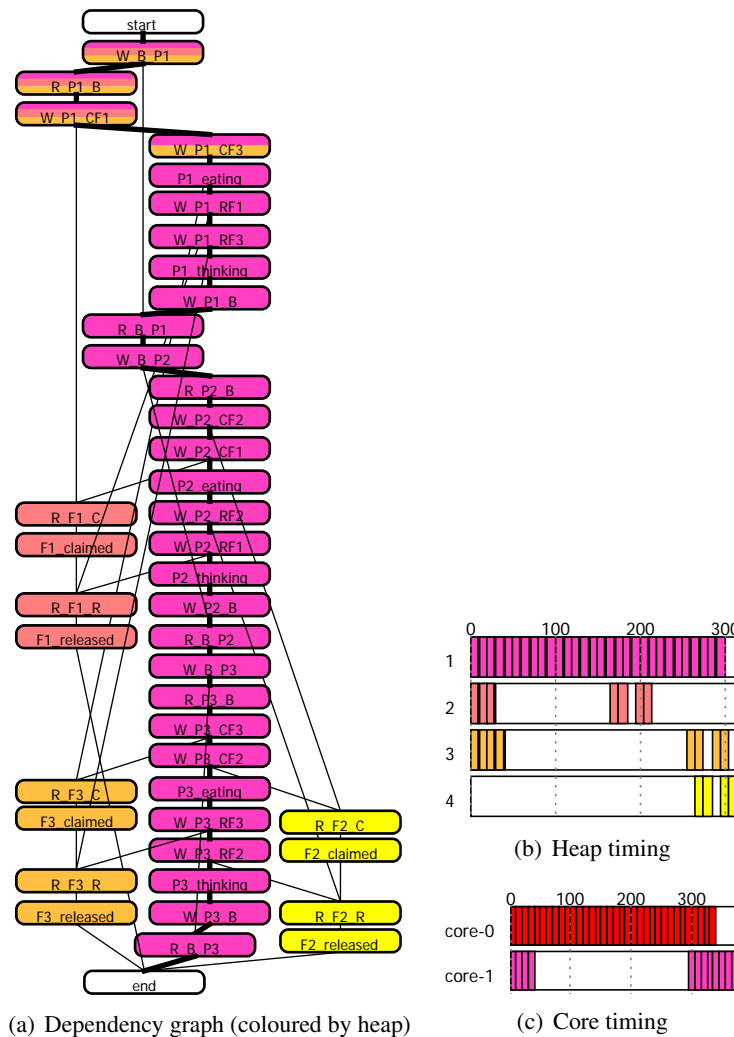


Figure 6.12, Philosophers with a butler

The results for the dining philosopher with butler model can be found in Figure 6.12. A very long critical path is visible and the individual philosophers cannot be seen as easily as before. Main reason is an implementation of the model without alternative relations and interaction with all philosophers and forks with the butler. Such a long critical path compared with other paths is unsuitable for the scheduler. The heaps are scheduled completely on one core, so the critical path gets scheduled on core-0 and all other heaps on the other core(s).

A nicer solution would be splitting the long heap and put it on multiple cores. The schedule length will not change, in fact it gets longer by the additional communication times, but a sort of pipeline could be used. When the first part of the critical path is finished, it could be started again while the second part is running on another core.

Plotter model

Finally the plotter model is analysed and its results are shown in Figure 6.13. Like the previous model, the plotter also has a long sequential path. This is as expected since the model was designed to be sequential: first sensor information is read, it gets processed, checked for safely and motors are controlled.

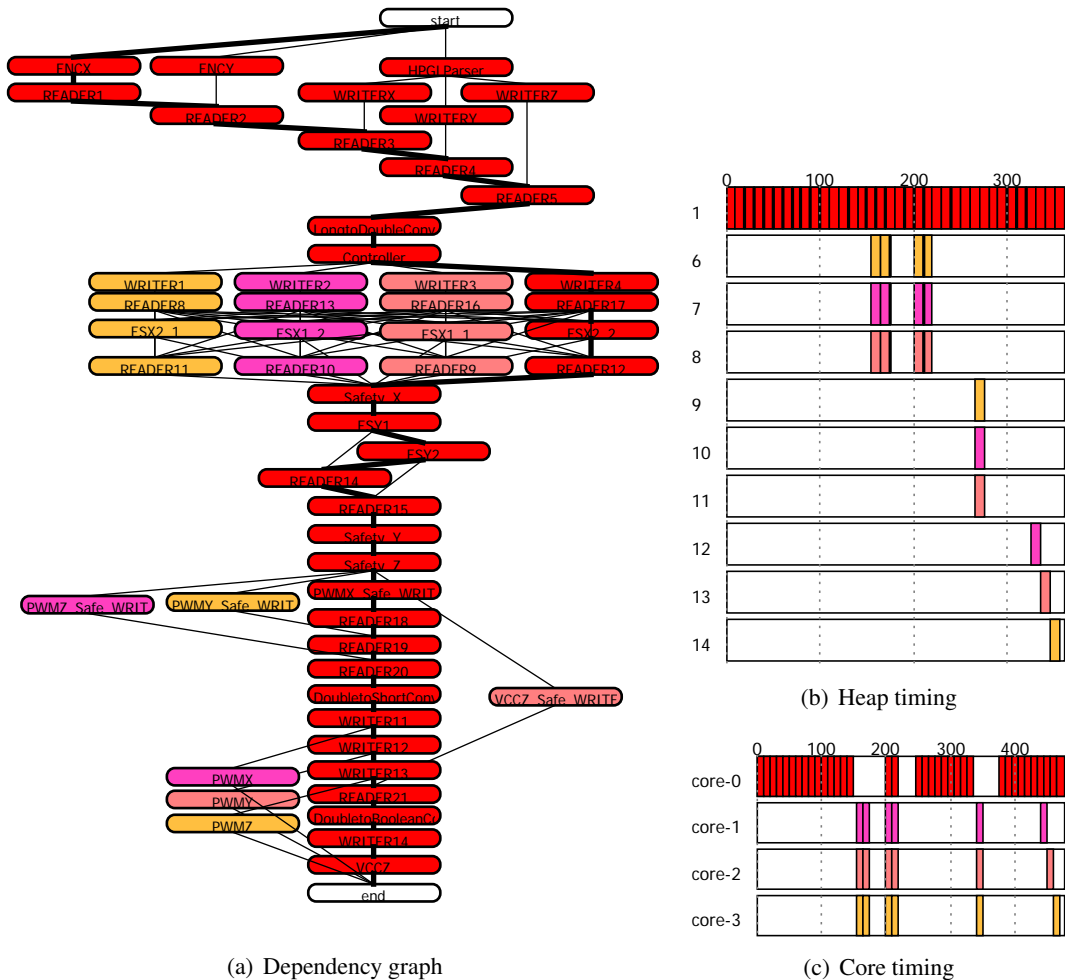


Figure 6.13, Plotter results

As mentioned for the previous model as well, such a long critical path is very suitable to be placed on multiple cores, at maximum one core for each of the four steps. Again a pipeline will be created, although it will be more imaginable since the model already is split into these four sequential parts.

The current cores are scheduled as optimal as possible. The gaps smaller than the weights processes scheduled on the other core, so the algorithm made a good choice. This model is not very suitable to be optimised as it is and it can be concluded that the model should be made more parallel when running on a multi-core target. If the pipeline is not feasible, it might be a solution to parallelise the X, Y and Z paths.

6.5 Discussion

As seen in the result section the analyser is very suitable to be used to analyse model for different situations and reasons. To see whether it is created optimal or to see how timing is estimated and which processes might become bottlenecks depending on the target system.

Even though, a couple of things to improve are known already and should be kept in mind when the analyser is used.

- In the current algorithm the implementation of the communication delays are too simple. A delay between the predecessor and the process is added, but in a more realistic situation this is not sufficient.
- The communication delays are also made too generic (as defined in the algorithm of van Rijn (1990)). When defining the heaps it is not known whether communication is going to take place or not. So the default communication costs should be removed. The core scheduler also adds default communication costs to heaps even when the costs of the heap are calculated for a core without communications.
- Communication between cores always has the same costs. In real situations cores might be on the same processor or be on different nodes in a distributed system. So the communication costs should be able to be different depending which cores are communicating.
- The current situation might become better when the heap scheduler is improved. The current heaps tend to be very long, mainly due the fact that the communication costs are too dominant. Big heaps are hard to schedule optimal since they are placed completely on just one core.
- The possibility to adjust the speed of the cores is not implemented very well yet, as shown by the setup of the dining philosopher model. When the differences in core speeds become too big, the core scheduler leaves big gaps in the core schedules. The larger the speed differences become the larger the gaps become.
- Due to the core speeds the weight of a process is reduced depending on the core the process gets scheduled on. This leads to rounding errors, since the internal data is stored as integer, which obviously leads to errors in the scheduler.
- Some processes might be node dependent, for example process which depends on external I/O. This I/O probably is wired to one node and the process should be present on that particular node. So a possibility to force a process to be scheduled on a particular node would be handy. This node can consist of multiple cores of which one should be chosen. More information about process-core affinity is described by Sunter (1994) section 3.3.

6.6 Conclusions

The results section shows that the analyser is usable in its current stage. It provides enough information for discussions to improve an analysed model. It also indicates the weak points of a model. For example the long critical paths indicates that a model is build too sequential, resulting in bad results on a multi-core target system.

Improvements are described in the discussion section to make the analyser algorithms even better. Depending on the situation the analysis algorithms have to deal with, the priority of implementing these improvements can be determined. Besides these suggestions, the algorithm can be modified with additional functionality. A lot of results are presented for which new applications can be developed.

One of these applications is the implementation of a deadlock analyser, which needs information about dependencies of processes to find situations which might become a problem. One of these problematic situations is caused by circular paths in the dependency graph. These paths also have bad influences on the algorithm and are handled internally already. Adding some extra rules to the internal problem handling code could be enough to detect deadlocks during analysis. When build in the model designer application (gCSP) the deadlock checks could be performed during the design of a model.

7 Conclusions and Recommendations

7.1 Runtime analyser

The runtime analyser is working correctly following the design specifications. Although the results tend to get too complex to interpret, it is quite possible to extract usable information from them as described in Section 3.5. Limitations of the analysis algorithm should be kept in mind in order to use the results without problems.

Directly implementing the results into a new and optimised model is a step too far at the moment. First the results should be tidied even better, so rules can be defined how to create a more optimised model using them. As stated before the results can give insights of the model, which was the goal for this analyser.

7.2 The meta-model

The choice for using Eclipse with EMF to create the meta-model with, worked out great. It is flexible and works well when looking at code generation even when the code has been modified. After the meta-model is created one does not have to look back and redesign the implementation, since it directly works as expected.

The current implementation of the gCSP2 meta-model is usable for the model analyser. All gCSP model elements can be retrieved quite easily and can be found on a logical place. Adding extra functionality is possible without disturbing the model. For example implementing process groups after the models were created already, was easy and did not break any of their original behaviour. So it can be concluded that the meta-model works like a charm.

7.3 Model analyser

Using van Rijns algorithm as a basis was a good choice. It provided a solid basis to implement the heap and core schedulers for the current situation. Even though his implementation did not provide enough rules to obtain an optimal choice from a variety of available choices, these rules still form the basis. The additional rules developed in this project make the algorithm even better, but still not yet perfect. Especially when using different core speeds some optimisations are required. Also the lengths or complexities of some heaps might be sub-optimal and hard to schedule later by the core scheduler.

The user interface is comprehensive and showing a lot of information using graphs, bars and colours. The functionalities of storing settings, exporting views and changing parameters for the algorithm makes it a usable and fun to explore. Therefore the goals of this analyser as described in Section 1.2.2 are met, although a lot of enhancements can be added as stated in Section 7.4.3.

7.4 Recommendations

During the project quite a few optimisations and additional purposes for both tools are discussed. These improvements will be described in the following sections, so future work can be done to create an even better and more usable set of tools.

7.4.1 Runtime analyser

In order to simplify the results of the runtime analyser it would be nice to extend the algorithm by finding the execution order of the chains as well. This can be done the same way as the process order is found. Section 3.4.2 (the plotter part) shows such a chain order example. Even though it is created by hand, the idea is the same.

7.4.2 The meta-model

In order to use the meta-model in the gCSP2 application, the meta-model needs some additions. The most obvious being graphical information, which could be implemented next to the model data. The description classes describe the model data, like author, version and so on, a graphical description class next to it could describe graphical information like colour, coordinates and dimensions.

When using a super class for these descriptions, a generic painting, action performing and code generating interface can be created. This results in an application which is able to interact with the model parts without the need to actually know what the parts are standing for. So the meta-model can be enhanced with new model parts without the need of extensively upgrading the gCSP2 application or reinventing the wheel over and over.

The meta-model also needs to be extended by implementing the currently missing model parts, like guards, linkdrivers, 20-sim code blocks, call channels and so on. Maybe some of the available elements could be modified to become more usable. Like adding code blocks to a recursion which are called when the recursion evaluation returns true or false. The 20-sim code blocks could be extended by creating a link to the actual 20-sim model and interaction between 20-sim and gCSP2 could be established.

Last but not least, it would be a big enhancement if the meta-model allows for so called master models, which could function as a blueprint for other models. These other models could refer to the master model when its functionality is needed instead of re-implementing its behaviour. With such an option libraries could be created containing these master models, or library blocks in this case, and other users could re-use the models without knowing their internal workings.

7.4.3 Model analyser

The main issue of the current implementation of the model analyser algorithm is the handling of communication costs. It just adds the communication costs to the end time of a preceding process. In a more realistic situation, both cores should reserve schedule time after or before the process to setup a communication channel. Implementing this is hard because whether a result has to be communicated is depending on whether two processes are scheduled on different cores or not. In order to save resources a core should be able to quickly setup the channel and other hardware handles the communication itself. This way, the channel setup time becomes minimised and the communication time becomes even zero. It can be compared DMA transfers in computers, which are handled outside the processor by other hardware as well.

This information is not available when the preceding process is scheduled. Just adding extra time after the process when its succeeding processes are scheduled might be tricky. Especially when the core has more processes scheduled after the process. The algorithm might not be able to find good results anymore, since the timing changes after the processes got scheduled. So these processes might need to get rescheduled again, which could lead to infinite loops. Hence the current model does not have this implemented yet.

More heap and core scheduling optimisations can be found in Sections C.4 and D.4. These recommendations are referring directly to algorithm parts which are not explained in the main report and are not discussed here as well.

Another point of optimisation is to implement different dependency relations, like the OR-relation described in 6.4.2 (the dining philosophers part). The scheduler could use its calculated timing information to determine whether one or more of the preceding processes of the OR-relation group has been scheduled already. If this is the case it can ignore the other members of such an OR-group. Also it can be shown in the dependency graph, what processes are used to get the result, by colouring the lines. Implementing this would result in better handling of alternative relations.

Other dependency relations could be implemented as well in order to use guard information or call channels.

As stated in Section 6.6 the tool could be extended by adding a deadlock checker. The dependency graph shows the relations between processes and this information could be used to find deadlocks, although some additional information might need to be included. Like the set of rules which determines a deadlock and the knowledge when certain processes block or not. Combining such functionality with the gCSP2 application could result in a design-time deadlock checker, which warns the designer when he designs a possible deadlock.

A EMF project files

The meta-model described in Chapter 5 is created in Eclipse EMF. This appendix will fully describe the EMF projects and its files.

A.1 Available project files

An Eclipse project can be build by extending other projects, an EMF project is such an extended project. It is based on an Eclipse plug-in project with two extra files stored in a model folder. The Eclipse plug-in project itself is again an extension, which is based on a Java project.

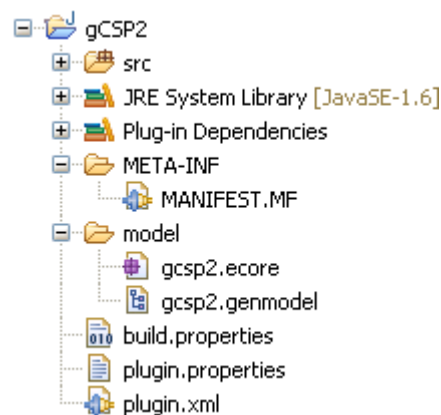


Figure A.1, Overview of an EMF project

Figure A.1 shows an EMF project, as mentioned before it consists of three projects mixed into a single one:

- A Java project: The src directory is the default location for the Java project to store the source code, the ‘JRE System Library’ is to manage the Java library dependencies and the build.properties contains information for Eclipse how to build the Java project.
- A plug-in project: MANIFEST.MF, plugin.properties and plugin.xml contain the plug-in properties. Together those files contain the information to hook into the Eclipse plug-in API in order to act as an actual Eclipse plug-in. An editor is provided to edit these files simultaneously, since lots of properties are available spread over these three files.
- An EMF project: The last two files, gmsp2.Ecore and gmsp2.genmodel are used to define the model and to generate model code. Like the plug-in files, both files can be edited by using the provided editor or by hand. A more detailed explanation of these files can be found in the next part.

A.2 Explanation of the EMF related files

The first file is the Ecore file, this file stores all information of the meta-model being created. Ecore is a meta-model itself, since it can be used to describe other models. The Ecore meta-model is described using itself, so it could be called a meta-meta-model.

The Ecore project file, called gmsp2.ecore in the figure, contains the meta-model information of the gCSP2 models.

Figure A.2 shows a part of the gCSP2 meta-model in the available Ecore editor. Three classes are shown: gCSPFile, VersionInfo and Model. The GCSPFile class has two references: one to the VersionInfo class and one to the Model class. A reference to another class tells the model that an object of this other class is contained in the current class. The VersionInfo class has two string objects, all standard and custom made Java classes can be included as fields of an Ecore class.

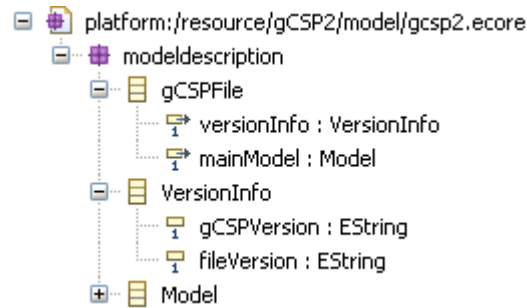


Figure A.2, Example of an Ecore file showed in its editor

The genmodel file contains all settings for code generation, there are many settings available so it is impractical to explain them all. The settings vary from package names to model optimisation properties. Using the genmodel editor, it is possible to change these settings.

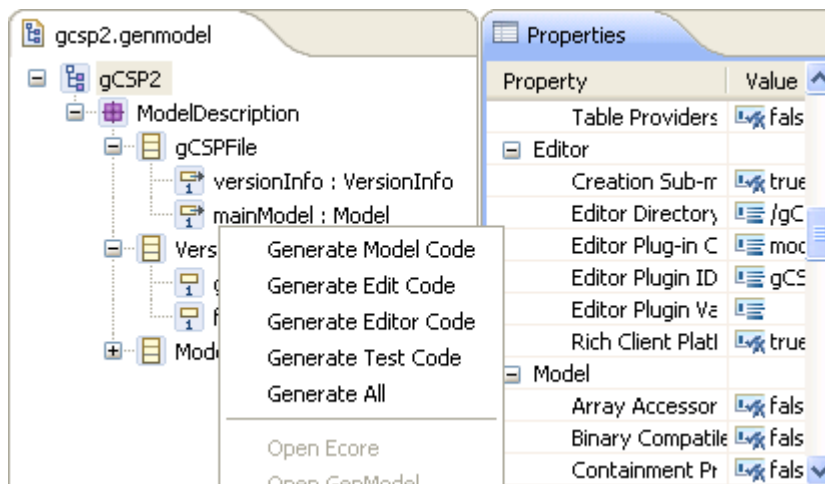


Figure A.3, Example of a genfile showed in its editor

An example of the genmodel editor is shown in Figure A.3, the classes are visible again, but in this editor they cannot be edited. Some of the available properties are shown on the right. In the shown menu the code generation commands can be found.

The generate model code option is the most important generation option, since it contains the code to store all information described by the meta-model. Classes are generated containing the given relations and fields and accompanying code is added to retrieve the related objects and to modify the field values. The Ecore code generator allows the developer to modify generated code parts without losing these changes when the model is generated again. So after a basic model is created, supporting methods can be added or filled in.

As the menu shows, it is also possible to generate edit and editor code, which is used to implement an Eclipse based editor, like the Ecore and genmodel editors. The code provides a simple, non-graphical editor plug-in, which fits seamlessly into the Eclipse environment.

The test code generation option can be used to create a test suite of the model. It generates an empty junit test class, to which methods should be added containing the test code. The code generation is very simple and it might be more convenient to manually create a test project containing the test classes.

B UML diagram of the meta-model

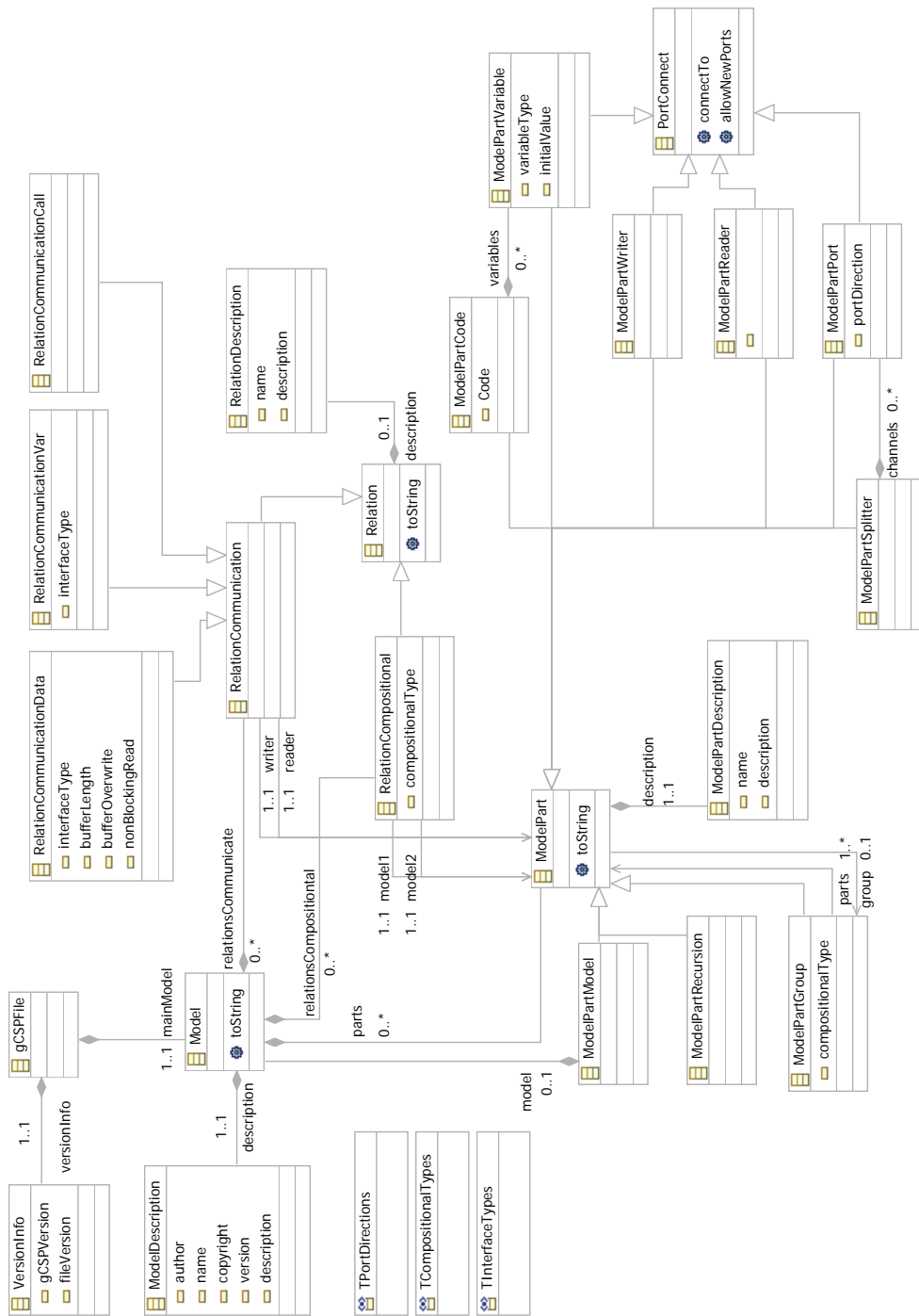


Figure B.1, UML diagram of the meta-model

C Heap scheduler

This appendix contains an extensive description of the heap scheduler. The heap scheduler takes distinction in processes which are part of the critical path and regular processes. This is done to keep the critical path as short as possible, to get an optimal schedule during the core scheduling algorithm step.

The algorithm works in three phases. The first phase adds all processes of the critical path to heap 1. Heap 1 is a special heap which is only used by these processes and their predecessors when the communication costs would become too high. The second phase basically covers the complete algorithm and will be explained in the following sections. The third phase removes redundant heaps by determining whether all processes on a certain heap are also part of another heap. These three phases are clearly shown in the calculate() method of the CoreScheduler class.

The following three sections will cover the rules embedded into phase two. The most important aspect of this phase is the distinction between critical path processes and the regular ones. When a critical path process is scheduled the subrules of ‘Decision rule 1’ are active and the regular processes use the subrules of ‘Decision rule 2’.

A queue is used to determine which processes are ready to be scheduled. In order to become part of the queue a process must have all of its predecessors scheduled already. Initially the queue is filled with the successors of the fictional start process. When the queue is emptied all processes should have been scheduled and phase two is finished.

First a section will describe the notation used to explain the rules. Next, two sections describe the situations of decision rules 1 and 2.

C.1 Heap scheduler notations

The following notation is used to describe the Heap scheduler rules:

- A is short for ‘process A ’
- A_p is short for ‘a predecessor of process A ’
- A_{p1} is short for ‘predecessor number 1 of process A ’
- $A_{p,g}$ is short for ‘a group processes preceding A_p ’
- $A_{p1,g}$ is short for ‘a group processes preceding A_{p1} ’
- W_A is short for ‘the weight of process A ’
- $t_{s,A}$ is short for ‘the start time of process A ’
- $t_{e,A}$ is short for ‘the end time of process A ’
- t_c is the time to setup a channel and communicating the result from one core to another.

Combinations of the used notations like t_{e,A_p} are possible as well. The notation in use to define a heap and its contents is shown in Figure C.1.

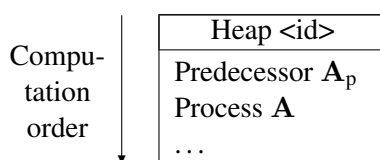


Figure C.1, Example notation for a heap

The defined heap has a identifier ‘<id>’. The heap containing the critical path has ‘1’ as identifier, all other heaps have a letter as identifier. The vertical arrow indicates the schedule order. So A_p is scheduled before A , as already defined earlier.

If a situation requires special conditions, for example timing conditions, they are displayed underneath the heap figure. Otherwise any conditions are valid for that particular situation.

C.2 Decision rule 1

This rule is called when the current process needs to be scheduled as a critical path process. Since phase 1 already added all critical path processes on heap 1, the current process does not require to get scheduled anymore. Decision rule 1 therefore is used to check the predecessors of the process. Each predecessor is processed separately and according to the following situations.

Situation 1

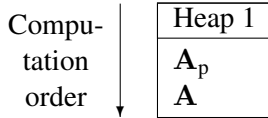


Figure C.2, Situation 1 before applying the decision rule 1

Figure C.2 shows that A_p is placed on the heap 1 already. So nothing has to be done.

Situation 2

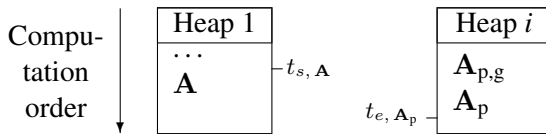


Figure C.3, Situation 2 before applying the decision rule 1

Condition: $t_{e, A_p} = W_{A_{p,g}} + W_{A_p} \leq t_c$

In the situation shown in Figure C.3 the time to communicate a result to another core is larger than the end time of A_p . So it is more efficient to copy A_p and $A_{p,g}$ to heap 1. In order to keep the critical path heap schedule as short as possible. The result is shown in Figure C.4.

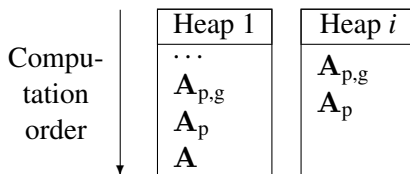


Figure C.4, Situation 2 after applying the decision rule 1

Situation 3

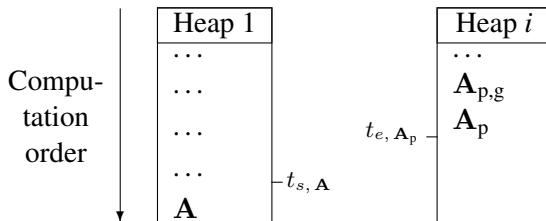


Figure C.5, Situation 3 before applying the decision rule 1

Condition: $W_{A_{p,g}} + W_{A_p} > t_c$
 $t_{e, A_p} + t_c < t_{s, A}$

Figure C.5 shows the situation for the given condition. Communicating the result of A_p from heap i to heap 1 is quick enough to have the result ready before A starts. Furthermore the total weight of A_p and $A_{p,g}$ is larger than the communication costs.

So communicating the result is more efficient and keeps the schedule of heap 1 as short as possible. Figure C.6 shows the fictional result after applying the decision rule 1. The current implementation

of the algorithm does nothing when this situation occurs. When communication timing gets added to the algorithm the figure is more accurate.

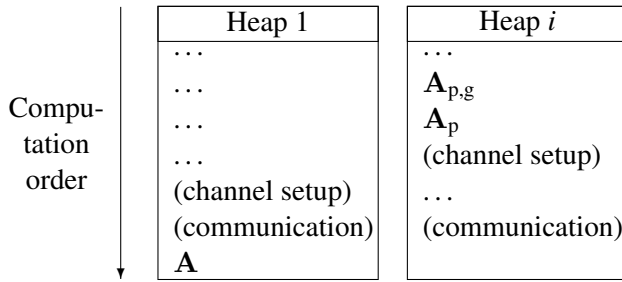


Figure C.6, Situation 3 after applying the decision rule 1 (for an algorithm which includes communication timing)

Situation 3b

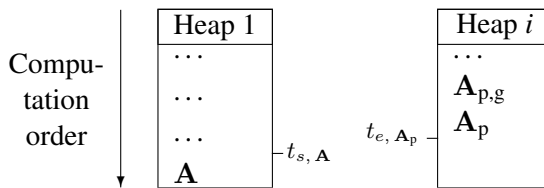


Figure C.7, Situation 3b before applying the decision rule 1

Condition: $W_{A_{p,g}} + W_{A_p} > t_c$
 $t_{e, A_p} + t_c > t_{s, A}$
 $W_{A_{p,g}} + W_{A_p} + t_c < t_{s, A}$

Figure C.7 shows an additional situation (compared to the algorithm of van Rijn), closely related to situation 3. It checks for another communication possibility in order to prevent copying A_p and $A_{p,g}$ to heap 1. The condition suggests that communicating is possible if A_p and $A_{p,g}$ are placed on a new heap, shown in Figure C.8.

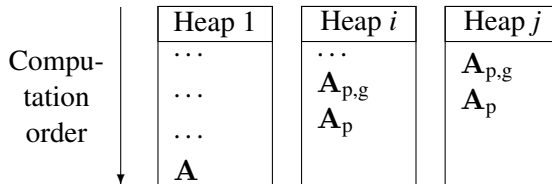


Figure C.8, Situation 3b after applying the decision rule 1

Situation 4

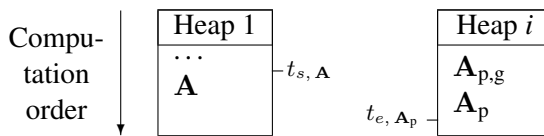


Figure C.9, Situation 2 before applying the decision rule 1

Condition: $W_{A_{p,g}} + W_{A_p} > t_c$
 $t_{e, A_p} + t_c > t_{s, A}$
 $W_{A_{p,g}} + W_{A_p} + t_c > t_{s, A}$

Figure C.9 shows the situation where nothing can be efficiently done. An idle time will be introduced to heap 1 or it will grow excessively. If

$$t_{e, A_p} + t_c - t_{s, A} < W_{A_{p,g}} + W_{A_p}$$

is true then it is more efficient to place A_p and $A_{p,g}$ on heap 1 (situation 2). Else it is more efficient to create a new heap to place A_p and $A_{p,g}$ on (situation 3b).

Situation 5

This situation is not a real situation, but defines two types of predecessors which should not be scheduled:

- the start process, even it is fictional it might be a predecessor of **A** according to the dependency graph.
- a preceding process which is dependent on **A**. Or in other words: a process which has a circular reference with **A**. Circular processes are not supported in this version of the algorithm.

C.3 Decision rule 2

Decision rule 2 schedules a process which is not a part of the critical path. Since the process has yet to be scheduled, it is not shown in the figures. In the following situations it is assumed that **A** needs to be scheduled.

Situation 1

No predecessors are available, so schedule **A** onto a new heap. This is shown in Figure C.10.

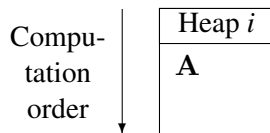


Figure C.10, Situation 1 after applying the decision rule 2

Situation 2

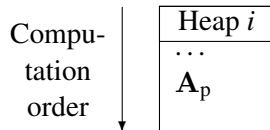


Figure C.11, Situation 2 before applying the decision rule 2

Only one predecessor is present and it is situated at the end of the heap, as shown in Figure C.11. **A** should be added after **A_p**, as shown in Figure C.12. If multiple heaps have such a situation, only the first heap is picked to place **A**.

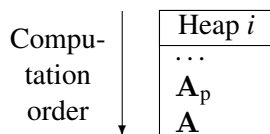


Figure C.12, Situation 2 after applying the decision rule 2

Situation 3

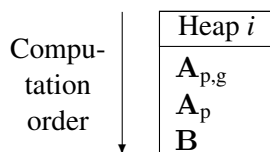


Figure C.13, Situation 3 before applying the decision rule 2

Again only one predecessor is present, but it now it is not situated at the end of a heap, as shown in Figure C.13. So a new heap needs to be created. A decision has to be made whether the new heap should contain **A_p** and **A_{p,g}** as well.

Condition: $W_{A_p} + W_{A_{p,g}} \leq 2 * t_c$

If the condition is true, it is most efficient to place A_p and $A_{p,g}$ on the new heap as well. As shown in Figure C.14. Otherwise only A is placed on the heap and the result of A_p needs to be communicated. This results in a similar situation as shown in Figure C.8.

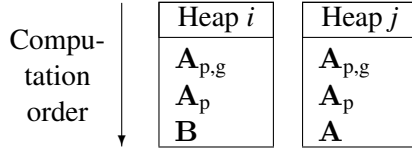


Figure C.14, Situation 3 after applying the decision rule 2 (when the condition is true)

Situation 4

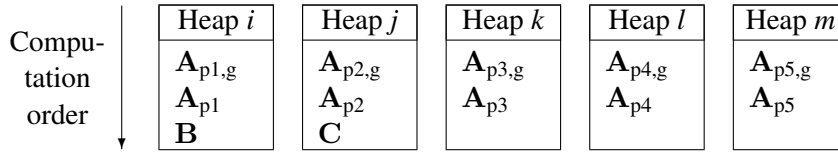


Figure C.15, Situation 4 before applying the decision rule 2

Previous situations all had just one heap available with preceding processes. Now multiple heaps have preceding processes (shown in Figure C.15) and the best heap should be picked to place A .

Heaps i and j are not suitable to place A on, for the same reason as described for situation 3. Therefore heaps k , l and m are suitable heaps for placement in this example. If no suitable heaps remain, a new one should be created.

From the suitable heaps the one with the largest schedule length is chosen. Since all other heaps need to communicate their result to this heap it is most efficient to send it to this heap. For all other heaps situation 3 can be used to choose whether the preceding processes needs to be copied or the result is going to be communicated.

An example situation is:

$$\begin{aligned}
 W_{A_{p1,g}} + W_{A_{p1}} &\leq t_c \\
 W_{A_{p2,g}} + W_{A_{p2}} &> t_c \\
 W_{A_{p4,g}} + W_{A_{p4}} &> t_c \\
 W_{A_{p5,g}} + W_{A_{p5}} &\leq t_c \\
 W_{A_{p3,g}} + W_{A_{p3}} &> W_{A_{p4,g}} + W_{A_{p4}} \\
 W_{A_{p3,g}} + W_{A_{p3}} &> W_{A_{p5,g}} + W_{A_{p5}}
 \end{aligned}$$

The last two equations show that heap k is the optimal heap. The first and the fourth equations show that heap i and m are best to be placed on heap k as well. Since that is more efficient than communicating according to situation 3. The other two heaps will not be copied, since those are too big. Because heap k has the longest schedule length, especially with the copies preceding processes, communicating is not wasting too many resources. The result for the example according to situation 4 is shown in Figure C.16.

C.4 Recommendations

The current two decisions rules are used in the model analyser to create the heaps. A couple of things could be made better in order to improve the heap scheduler.

The communication delays mentioned in the several situations are considered for these situations. When the decision is made that a result needs to communicated, it should be stored in the two concerned heaps. When determining the schedule length these communication setup times can be included as well. This could result in better heap scheduling, which reflects on the core scheduling as well.

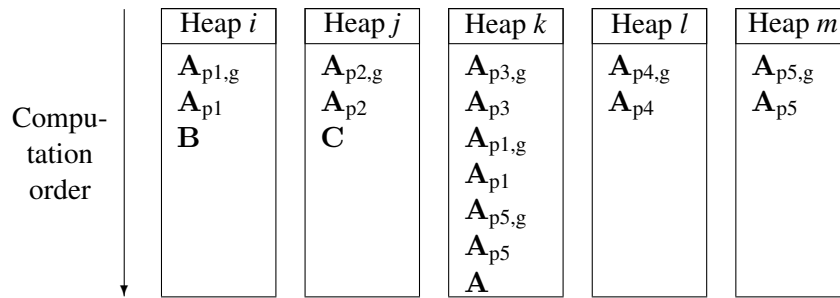


Figure C.16, The example of situation 4 after applying the decision rule 2

Another weak point of the current scheduler is removing the redundant heaps. A heap is thought to be redundant when all of its processes are found on another heap. Redundant heaps are removed, since it is not required for the core scheduling algorithm. This saves analysis time because the heap is not processed anymore. A better criterion would be checking if the processes also have equal or larger starting times. For example decision rule 1 situation 3 assumes that heap *i* is present. So its result can be communicated to heap 1. If heap *i* is removed the result has to be retrieved from another heap which might not have the result ready on time and unnecessary idle time(s) are introduced.

D Core scheduler

This appendix contains an extensive description of the core scheduler. The core scheduler also consists of three phases. First index blocks are created on the heaps. Next these index blocks get placed on the available set of cores. After an index block is placed the ready queue is updated.

At the start of the core schedule, the current schedule length is set to zero. This length indicates the length of the schedules of the cores. During the scheduling process the length will grow in order to be able to fit the processes on the cores.

In this appendix the same notation is used, as described in Section C.1.

D.1 Index blocks

First phase of the algorithm is to create index blocks for the heaps. Figure D.1 shows two heaps requiring index blocks.

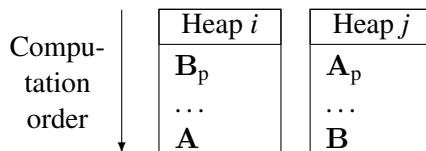


Figure D.1, Example heaps requiring index blocks

A Heap is scheduled on a core when all of its preceding processes are scheduled already. So the example is a problem, since heap i is dependent on heap j and vice versa. To solve this problem, index blocks are introduced. Both heaps are split into two index blocks as shown in Figure D.2.

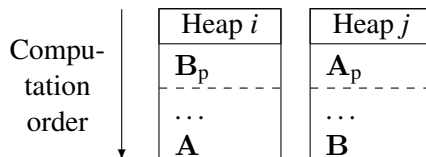


Figure D.2, Example heaps with the added index blocks

Now the scheduler first can schedule the first index block of both heaps. After that the second two index blocks are allowed to be scheduled, because all predecessors are scheduled already.

The example shows the first rule for adding heap blocks. The second rule occurs when a process on the heap is predecessor for two other processes on the heap. Or in other words when the path of processes on the heap splits. Figure D.3 shows this situation.

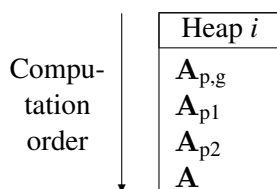


Figure D.3, Example heap with a split path

When the split path joins again another index block is required. This results in the split part being on a separate index block. The core scheduler seemed to get a better results when using this second rule. Probably because the complexity of such heaps prevent it scheduled in a early stage. The result for the given example is shown in Figure D.4.

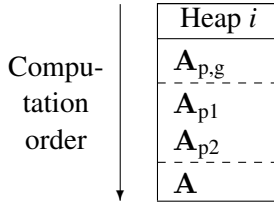


Figure D.4, Example heap with a split path contained in its own index block

D.2 Heap placement

The heap placement is split into several rules. The first rule checks if the heap already is partially scheduled on a core. If this is the case, the current index block is scheduled on the same core on a free spot after the previous index block of that heap. If not free spots are present, it is scheduled at the end of the core.

When the first index block of a heap is scheduled, first a number of potential cores is listed. A core has potential when the processes of the complete heap fit on the core with the current schedule length in mind. Whether the heap is split and the processes are placed onto free spots or the heap is placed completely at the end of the core does not matter. In order to know if a heap fits, an estimated schedule length is calculated:

$$t_{est} = W_{virt} + t_{idle}$$

W_{virt} is the weight of all processes scheduled on the core added to the processes placed on the heap. The weight of a process is included only once. The introduced idle time t_{idle} is not used in the current implementation of the algorithm. If the estimated time is smaller than the current schedule length the core is put on the potential cores list.

If the list of potential cores is empty, the current schedule length is too small. So it needs to be enlarged to the minimum core schedule length, found during the testing of what heaps would fit or not. The list will be created again and now it will contain at least one potential core.

Using the list of potential cores, the optimal core will be determined. First the best placement time for the index block is determined. This time should depend on the free slots on the core, but currently the core schedule length is used. This is the worst-case scenario and thus a valid result, but using free slots would be nicer. If the found placement time is larger than the starting time of the index block, the core is removed from the potential cores list.

For the remaining cores, if any, a value will be calculated according to:

$$coreStat = W_{overlap} - t_c - t_{idle}$$

A higher coreStat value the more optimal that core is. Overlapping process is a positive thing, since it saves resources when a process is not being executed multiple times. The communication time (t_c) is not calculated, since communication times are not fully implemented yet. Same goes for idle time. So basically only the overlap determines the optimal core.

If two cores have the same coreStat value, the core with the lowest scheduled weight is considered even more optimal.

If no optimal core is found, the index block starting time needs to be delayed. So finding a core with shortest estimated schedule length seems a good idea. The estimated schedule length is the virtual weight value (W_{virt}) of the core as described earlier. The core with the lowest estimated length, after the heap is added, is the most optimal core.

If two cores have the same estimated schedule length, the overlap of the heap and the two cores is determined. The core with the biggest overlap becomes the optimal core.

Even after this test two cores still might be optimal, so another test is performed. The more predecessors reside on the core the more optimal the core becomes. Having predecessors on the same core saves communication resources.

More conditions and test can be added if required. Although these three conditions seem enough to determine the optimal core when the potential cores list was emptied.

D.3 Ready queue

When an index block is scheduled on a core, new index blocks might become ready to schedule. So this phase will check the successors of the index block and add them to the ready queue when all of their predecessors are scheduled. This seems easy to achieve, but determining whether all predecessors are scheduled is tricky. Especially when the predecessors are placed on multiple heaps.

When the algorithm starts the ready queue is filled with the succeeding processes of the fictional start process. When an index block got added, the last process of that index block is used.

For each succeeding process all index blocks (on multiple heaps) containing it are tested. The processes on the succeeding index block must have their predecessors scheduled. Whether a predecessor is scheduled depends on several situations:

- the predecessor is placed on the same heap and scheduled
- the predecessor is not placed on the same heap, but **all** index blocks containing the predecessor are scheduled

The first rule is active when the result does not have to be communicated, so that heap is our real preceding heap. The second rule is to make sure to have an optimal choice. All preceding heaps are scheduled and the core scheduler can pick the optimal core when scheduling the index block.

When all processes of the index block have their predecessors scheduled according to the rules. The index block can be added to the ready queue. When the index blocks are not perfectly formed or the ready queue rules fail, the core scheduler is likely to ‘forget’ to schedule a series of processes. The ready queue is a very delicate system.

Also the order of index blocks to be scheduled determines how optimal the result of the core scheduler becomes. So when an index block should be added to the queue another set of rules is used.

Firstly the index block is added in such a way that the queue stays sorted on the starting time of the index blocks. An index block with the lowest starting time should be scheduled first, since it should be scheduled in front of the other index blocks.

Secondly when multiple index blocks have the same starting time, the total weight of the processes in the index block is used. The index block with the highest weight should be scheduled before the other index blocks. When two index blocks have the same weight another nice rule could be added. Currently this situation is ignored and a random choice is made.

D.4 Recommendations

As stated in Section D.1 the second index block creation rule takes care of the split paths on a heap. A better way for splitting the heap might be to isolate the complete split paths as shown in Figure D.5. The predecessors of each path are influencing the scheduling of their path only, not of the other path as they do now.

Enabling the ignored values, described in the previous sections, would make the core scheduler better. Values like t_{idle} and t_c (different for each core) are explained in the report written by van Rijn. Using free spots when possible in estimating values will give a better choice which core is potential or optimal. When a heap is tested on a core, the fact whether communication is needed or not can be determined for the first time. Enabling or disabling the communication time, depending

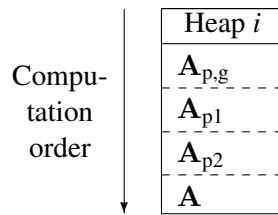


Figure D.5. Better solution for split paths

on which core is tested, influences the choices also. So a lot of possibilities are left to implement in order to optimise the core scheduler algorithm. The Java code contains a lot of ‘FIXME’s, ‘TODO’s and comments to describe what needs to be done.

When the cores have different speeds, internally the process weights are adjusted. Resulting in different start and end times of the scheduled processes. The algorithm is not (fully) aware of this possibility and the bigger the core speed difference become, the larger the idle times also become. This is mainly due the fact that core speeds got added at a late stage of the project and the algorithms of van Rijn also did not include it. It is recommended to fully investigate what parts are (also) influenced by the core speeds and make those parts aware. So the large idle times will be gone eventually.

E Runtime analyser code description

All classes, methods and fields are described by JavaDoc, so use a JavaDoc aware editor or generate the JavaDoc documentation to find information about these specific parts. This appendix describes the structure of the classes in a chronological order to clarify which is found where.

Three packages are available:

- `gjsp.analyse`, containing all algorithms and supporting classes. They (almost) should be able to function without the user interface.
- `gjsp.analyse.ui`, containing all user interface classes which defines the (graphical) user interface
- `gml.debug`, containing a copies class in order to prevent `gCSP` added completely to the jar file.

The first two packages will be described in the following sections.

E.1 The analyse package

As stated the analysis classes should be UI independent. This is not completely the case, since the analysis classes uses the `MainWindow` class to output informational lines to the status, console or debug panel. When using the algorithm in another UI it is fairly simple to add the required methods to the new `MainWindow` class.

In order to start an analysis the `ToolSetup` class needs to be used. It initialises all required classes. Like the `AnalyserProcess` class, which represents the (running) executable. And the `Analyser` class, which contains the actual algorithms. The `ToolSetup` class is a threading class which finishes when the analyser finishes.

The `AnalyserProcess` class runs the executable and attaches the `ConsoleReader` class to it. So the console output (`stdout` and `stderr`) can be read and is transferred automatically to the UI.

The `Analyser` class initiates the TCP/IP connection with the running executable by using a `SocketCommunicationHandler` object. This object handles all low level communication and provides methods to read or send information from or to the executable. The `Analyser` also creates a `ControlAnalyser` object, which is part of the UI and handles controlling the executable.

The `Process`, `ProcessChain`, `ProcessChainList` and `ProcessList` classes are used by the `Analyser` to store the chains. Before these classes were created, the lists were implemented as real lists and it was hard to keep track of the items on the lists. These classes provide methods to use and modify the lists at a higher level.

E.2 The UI package

The user interface packages provide the user interface with the `MainWindow` class. In order to keep the user interface running when the algorithms are running a thread wrapper class is added, called `UIThread`. This class is used in the `main()` method to show the `MainWindow` and start the application. When the `MainWindow` closes the `UIThread` class finishes running and the tools exists, all independent whether the `Analyser` thread still is running or not.

The `ModelBuilder` class is a threading class which will create an executable form a given `gCSP` model. Assuming that the path settings are correct. When the class finishes the executable is created or an error occurred.

The `LogPanel` provides a tab, visible at the bottom of the `MainWindow` which contains text lines to inform the user. Since the debugging tab has two extra checkboxes the `DebugPanel` is created which uses the `LogPanel` class and adds memory storage and filtering methods.

Finally the ControlAnalyser acts as glue between the Analyser thread and the UI thread. This class is required since the Analyser thread need to periodically check whether a button in the control panel is clicked or not. This might be at a later time than the button actually got clicked.

F User manual runtime analyser

This appendix contains the user manual for the runtime analyser. All functionalities of this tool are completely described. Figure F.1 shows a screenshot of the runtime analyser, as shown in Section 3.1. It already described the contents of the panels. This contents will be explained further in the following sections.

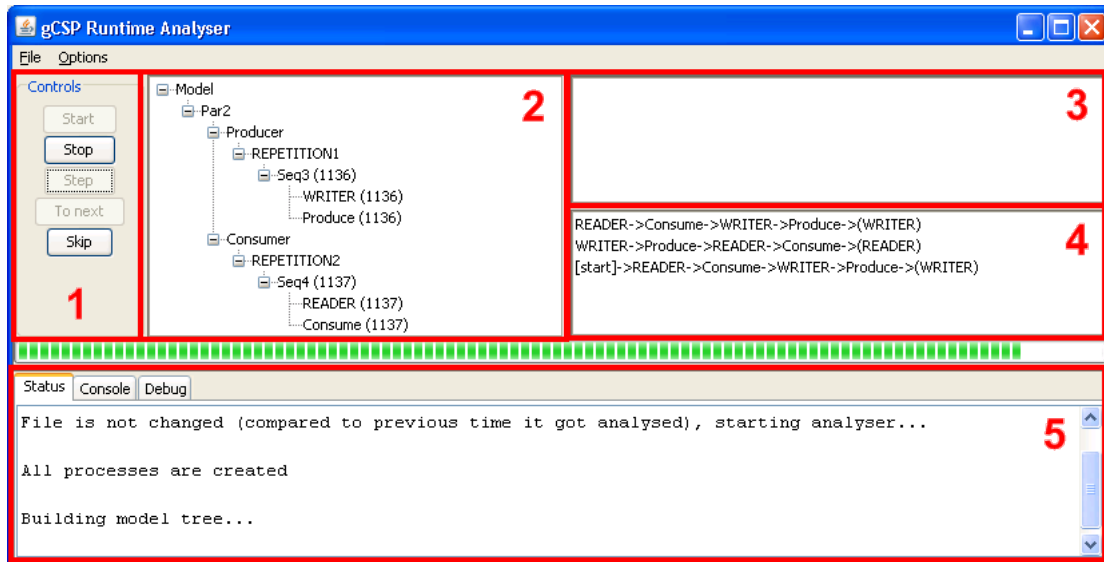


Figure F.1, UI of the runtime analyser, same as Figure 3.2

F.1 Controlling the analyser

The top left part, numbered 1, contains buttons to control the algorithm after a model is started. The ‘Start’, ‘Stop’ and ‘Step’ buttons are quite self-explanatory. The ‘To next’ button can be used to run the current algorithm step and automatically stops when the step is finished. Currently two algorithms steps are available, see Sections 3.2.1 and 3.2.2. When the model tree creation is not yet finished, the button will run the analyser until the model tree is finished. For models which have processes which are not used the result is an infinite running analyser. When the button is clicked after the model tree creation is finished, the analyser will run until the process order algorithm is finished. Since this currently never happens, the analyser will keep running. The usability of this button might become better in the future, when new algorithms get added and these algorithms are finished at a certain point.

The ‘Skip’ button skips the current algorithm, again the functionality is quite limited. When the model tree is created, this algorithm can be skipped. leaving the process order algorithm running. When the process algorithm is only running, clicking the button results in a finished analysis since not other algorithms are present at the moment.

F.2 The model tree

Panel number 2 is the output of the tree recreation algorithm described in Section 3.2.1. It shows the tree as it was recreated by the algorithm. It should be the same when comparing to the original model tree in gCSP.

The numbers shown behind the process names in parenthesis, indicate the number of times the process is finished. In this example the numbers are all the same, but models which have nested recursion or alternative channels included, might have different finished counts. The numbers can be used as an indication whether the models is functioning as expected. For example an inner

recursion runs ten times before it is finished and the outer recursion runs one time. The inner loop processes should have their counters about ten times higher than the outer loop processes. This is clearly visible when a commstime model is analysed, shown in Figure F.2. Changing numbers show that the executable still is running and the analyser is correctly connected to it.

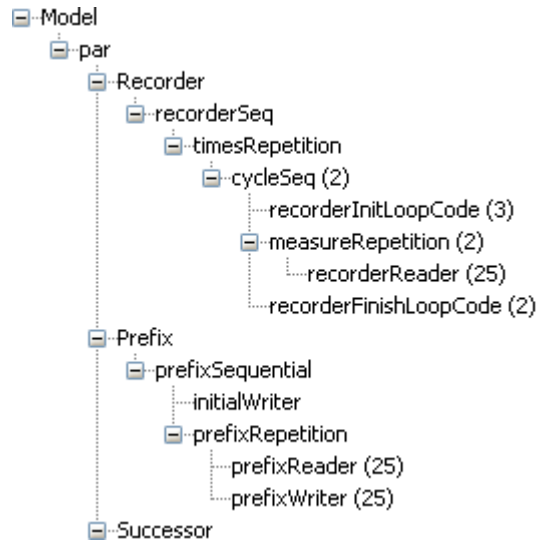


Figure F.2. A commstime tree, showing the varying finished counts

F.3 Unordered processes list

The unordered processes are shown in panel 3. The panel is empty in the figure since all processes are placed. However, in the initial phase of the analysis this panel shows the available processes. During the analysis the number of visible processes will decrease.

When processes will not get removed from this panel, it might indicate that not all processes are reached during analysis. For example when a model is dependent on IO and some of these IO signals never occur the corresponding processes will not be reached. It also might indicate that the model has a design flaw and the processes are redundant or have assigned wrong relations to the other processes.

F.4 The chained processes list

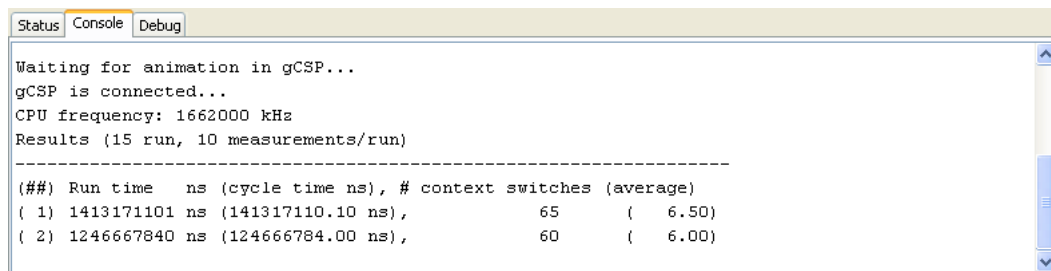
The results of the process order algorithm, Section 3.2.2, are visible in panel 4. Each chain has a unique starting process to make sure the results can be used in a optimised model. The '[start]' chain contains the processes which are finished when the model got started. This chain might be the same as another chain. If this is the case the start chain can be ignored.

The ending processes of the chains are put between the parentheses. These processes are only informative and should not be included in the optimised model. They allow for the possibility to follow the flow of the analyser through the chains. Multiple processes might end a chain. In this case, the chain is followed by any of these processes depending on the current execution steps of the model. The plotter model, discussed in Section 3.4.2, shows multiple ending processes on some of its chains.

These ending processes might be marked with one or two asterisks, these asterisks indicate that the end process is referring back to the current chain. One asterisk indicates that it refers back to the start of the chain. Two asterisks that it refers back to a process somewhere in the middle of the chain. Again this is only an indication to help the user how the chains should be constructed in order to get an optimised model.

F.5 Informative panels

Finally panel 5 shows the status of the analyser. It shows the steps when a model is compiled into an executable or which analysis steps are finished and running.



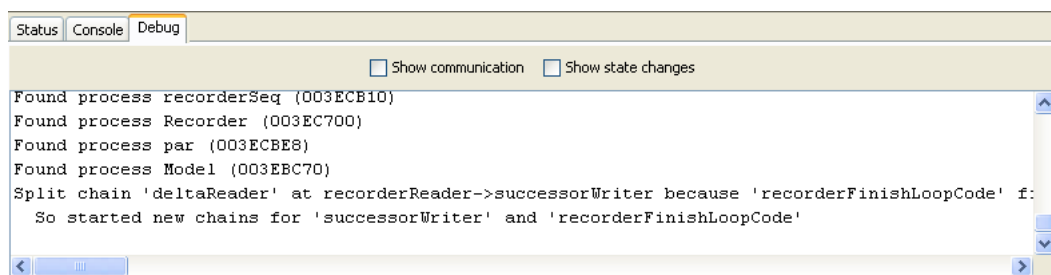
```

Status Console Debug
Waiting for animation in gCSP...
gCSP is connected...
CPU frequency: 1662000 kHz
Results (15 run, 10 measurements/run)
-----
(##) Run time   ns (cycle time ns), # context switches (average)
( 1) 1413171101 ns (141317110.10 ns),          65      ( 6.50)
( 2) 1246667840 ns (124666784.00 ns),          60      ( 6.00)

```

Figure F.3. The console tab

Figure F.3 shows the console tab of the panel. It contains the ‘stdout’ and ‘stderr’ output of the executable during analysis. This is not used by the analyser algorithm and only to inform the user about the executable status. When a model is compiled, the compiler warning and errors will be shown on this tab as well. The Status tab only shows a warning that errors occurred during compilation.



```

Status Console Debug
 Show communication  Show state changes
Found process recorderSeq (003ECB10)
Found process Recorder (003EC700)
Found process par (003ECBE8)
Found process Model (003EBC70)
Split chain 'deltaReader' at recorderReader->successorWriter because 'recorderFinishLoopCode' f:
  So started new chains for 'successorWriter' and 'recorderFinishLoopCode'

```

Figure F.4. The console tab

The last tab is shown in Figure F.4. It shows the debug information of the analyser algorithms. For example the last two lines of the figure explain why a chain is split and how. Because a **lot** of information is shown in this tab, two checkboxes are present to filter on types of information to be shown.

‘Show communication’ can be selected to show raw unformatted protocol data send and received by the runtime analyser. The ‘Show state changed’ checkbox can be selected when the formatted information about state changed should be visible. Whether the information of the two checkboxes is actually shown also depends on the settings described in the following section.

F.6 Runtime analyser settings

The runtime analyser has a dropdown menu containing a couple of extra settings. The first three settings can be used to determine whether the analyser should store the different information types should be stored in memory or not. When running the analyser for a longer period of time, the amount of information might become excessively large. The analysis might stop because the Java Virtual Machine (JVM) is out of memory. Unticking these options result in less memory usage and the analysis will be able to run for a longer period of time.

The following three settings can be used to set the paths required to be able to compile a model into an executable. If a path is incorrect and the required program or file cannot be found the console output will show an error.

All settings are stored and automatically set when the analyser is run.

G Model analyser code description

All classes, methods and fields are described by JavaDoc, so use a JavaDoc aware editor or generate the JavaDoc documentation to find information about these specific parts. This appendix describes the structure of the classes in a chronological order to clarify which is found where.

Three packages are available:

- `gjsp.analyse`, containing all algorithms and supporting classes. They (almost) should be able to function without the user interface.
- `gjsp.analyse.ui`, containing all user interface classes which defines the (graphical) user interface
- `gjsp.analyse.ui.bars`, containing all bar panels (currently the heap and the core schedule bars) and the base class providing the basic bar functionalities.
- `gjsp.analyse.ui.settings`, containing all settings panels (currently dependency graph, core and process settings) and the base class providing the basic setting functionalities.
- `org.jibble.epsgraphics`, containing the EPS exporting methods

Only the first two packages have enough classes to describe their relation. These descriptions can be found in the following sections.

G.1 The analyse package

The `ModelAnalyser` class is the main analysis class, it is responsible for the different algorithms steps, shown in Figure 6.1 and acts as glue between them. This class is a threading class, so the algorithm runs on a different thread than the user interface.

The `DependencyGraph`, `CriticalPath`, `HeapScheduler` and `CoreScheduler` classes are the algorithm steps currently available. The model tree is build by the UI since it is not used by the algorithms. All algorithm classes are static and are called using their required parameters, like a processes list or heaps list. Exceptions are used when the algorithm step fails, so the `ModelAnalyser` class knows that the analysis should be stopped.

The other classes in the packages are used to store and transport the results of the algorithm steps. They contain fields to store the information and methods to interact with the information. The names of these classes are explaining what kind of information is stored inside.

G.2 The UI package

The UI package contains a `UIThread`, `MainWindow` and a `LogPanel` class, like the runtime analyser. The `UIThread` and the `LogPanel` classes are even exactly the same. The `ExportDialog` and `StoreSettingsDialog` classes provide the two available dialogs of the UI. The `DependencyGraphPanel` class shows the dependency graph and contains all required methods for selecting, changing, storing and loading the dependency graph.

More information about this package can be found in Section E.2, since both tools have the same layout.

H User manual model analyser

This appendix contains the user manual for the model analyser. It describes all functionalities of the tool. Figure H.1 shows a screenshot of the model analyser. The numbered panels are briefly described in Section 6.1 already and more information is provided in the following sections.

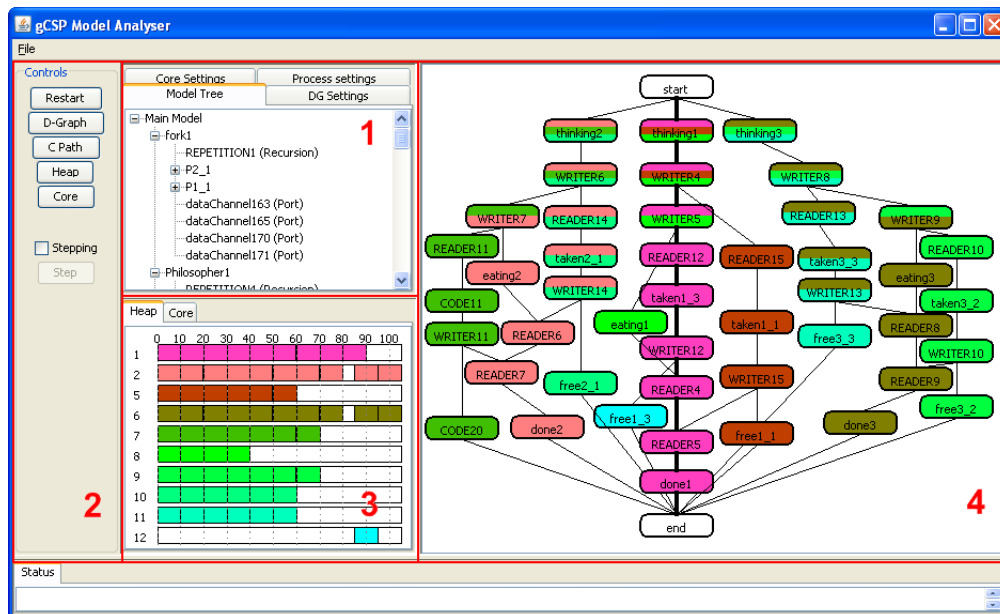


Figure H.1, UI of the model analyser, same as Figure 6.2

H.1 Model tree

Panel 1 shows the model tree. It shows the same structure, compared to the gCSP model. After the name of a process its the type is given.

The model tree is not used by the algorithms and functions as a check whether the gCSP2 model is converted correctly from a gCSP model. This check is required since a gCSP2 editor is not available yet.

The other tabs available in panel 1 contain settings to influence the algorithms. These settings are explained in Section H.5.

H.2 Controlling the analyser

The buttons available in panel 2 can be used to control the analyser. When a model is loaded the algorithm automatically starts and the UI looks like the figure. By default the dependency graph does not show any colours. Section H.3 describes the possibility to select a colouring method to colour the processes. After modifying settings to influence the algorithms, they have to rerun.

The top five buttons rerun the algorithm(s) from a specified point. Depending on settings got changed one or more steps needs to be redone. Especially on slower computers, it might be profitable to run only these algorithms again. Rerunning the 'dependency graph creation' algorithm results in the graph being recreated and the processes are unordered again. So starting the algorithms after this step is also timesaving, since the dependency graph stays intact.

When the checkbox is selected, the heap and the core scheduler algorithms perform one step and waits. By clicking on the step button the next step is performed. So it enables the user so see what the order of scheduling is and makes it possible to determine why a certain process is placed onto

a certain heap or core. If the checkbox is deselected and the step button is clicked the algorithm will finish its analysis completely.

Note: Too fast clicking on the Step button might result in a button which stays gray. This bug is not yet found and when it happens the complete analyser needs to be restarted. Hopefully it gets addressed properly in the future.

H.3 Schedule information

Panel number 3 contains two tabs with the heap and core schedule information. Both informational views are build-up the same. They contain bars representing the heaps or cores. The bar length is a sort of timing axis, although no real times are used. The objects placed on the bars are the processes, ordered on their relative times. Unscheduled parts, white gaps, indicate that a heap or core is idle. When hovering over a scheduled process, it shows the process name, the process gets selected when it is clicked.

The heap view has a popup menu where the colouring type can be selected. The resulting process colours are also visible in the dependency graph view. Several types are available:

- None, will removes the colour information
- Heaps, gives a unique colour to the processes scheduled on each heap
- Index Blocks, gives a unique colour to the processes being part of each index block
- Cores, gives a unique colour to the processes scheduled on each core

At the moment 40 different colours are available. When more than 40 colours are required, the colours are reused. The current set of colours is believed to be easily distinctable. Other colours can be added in the code quite easily when required.

H.4 Dependency graph

The dependency graph is visible in panel 4. It contains the processes and their dependencies. The additional start and end processes are shown as well. Normally the start process is on top and the end process at the bottom. In this situation the graph should be read from top to bottom, i.e. a process connected to a lower process is the predecessor of that lower process. When a process only has the start process as predecessor, it does not have dependencies. If the process only have the end process as successor, it does not generate dependencies to other processes. A combination of both also is possible for a completely isolated process. The thick line from the start process to the end process shows the calculated critical path, which depends on the amount of processes and their weight values.

The processes can be clicked to select them. Multiple processes can be selected by clicking them while pressing the control key. By clicking them, with the control key press, again the process gets deselected. When the selection is clicked and the mouse button is hold, it is possible to drag the selection by moving the mouse. In order to switch two processes, selected one and click the other process while holding the shift key. Hovering over a process for which the text is too large, it shows the complete text.

This view also has a popup menu which can be used to specify what text should be available in the processes:

- Process names, shows the name of the process.
- Critical path values, shows the values used to calculate the critical path
- Scheduled heaps, shows the heaps on which the process is scheduled
- Heap timing, shows the timing of process for the given heaps. The format is 'x: y-z'. Where x stands for the heap number, y for the relative start time and z for the relative end time.
- Scheduled cores, shows the core on which the process is scheduled
- Core timing, shows the timing of the process for the given cores. The format is 'x: y-z'. Where x stands for the core name, y for the relative start time and z for the relative end time.

- Core internal timing, same as core timing but without the core speed correction. So it shows the timing inside the core classes, which can be used for debugging purposes.

H.5 Algorithm settings

The algorithms can be influenced by a number of setting panels. All panels have a ‘Change’ button, which should be clicked in order to activate the made settings. After clicking the button, the algorithm has to be rerun manually and depending on the setting some algorithm steps can be skipped. Figure H.2 shows the core settings.

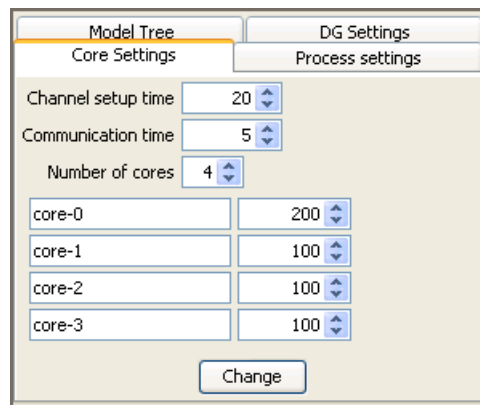


Figure H.2, Core settings

The channel setup time and the communication time can be specified here. Currently these two times represent the delay when a process is depending on a process scheduled on another core. In the future, the setup time is the time it takes to setup the channel. Whether the channel needs to be setup every time or not, depends on the future algorithms and available settings. The communication time is the time it takes to communicate a (default) result when both processes have the channel setup. These two settings require that the algorithm is rerun from the heap scheduler.

The number of cores is used to add or remove available cores. Each core can have its name and speed set separately. The figure shows that ‘core-0’ has a speed which is twice the speed of the other cores. The core names can be used to distinguish a multiple node system. The core specific settings are only influencing the core scheduler algorithm.

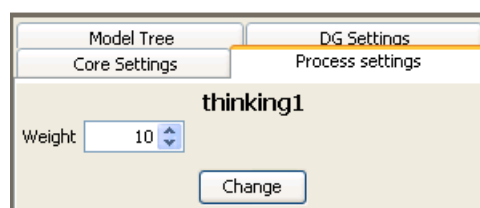


Figure H.3, Process settings

Figure H.3 shows the process settings panel. Currently only the weight of a process can be changed. The algorithms should be rerun from the critical path calculations, since the critical path is depending on the process weights.

In the future this panel might contain settings to force a process onto a specified core. Future versions also might offer the possibility to change the settings for a group of processes selected.

Figure H.3 shows the dependency graph settings. These settings can be used to influence the minimum sizes of the processes and the spacing between them. None of these settings influence the algorithms, so no reruns are required.

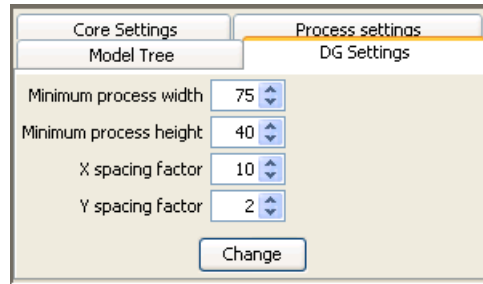


Figure H.4. Dependency graph settings

The settings described in this section can be stored, so they can be loaded easily again. Using the stored settings multiple setup for target systems can be analysed quickly. The ‘Save settings’ option in the ‘File’ menu opens the window, shown in Figure H.5.

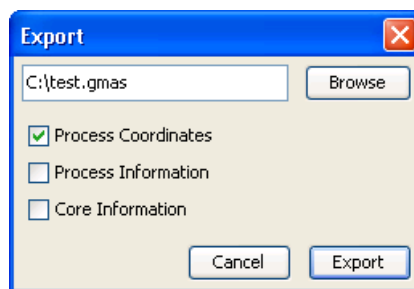


Figure H.5, Export settings

The filename can be specified in textfield. The browse button opens a dialog window which helps to select a location. The gmas extension is short for ‘gCSP2 Model Analyser Settings’. The three checkboxes below the textfield allows the user to select what needs to be stored. For example it is possible to store the dependency coordinates and the process weights only. So other setting files can be used to store multiple target systems, which then are easily accessible. The default location to store the settings files is in the settings folder at the same location as the model analyser jar file is present. If this directory is not present it will be automatically created.

Loading a settings file can be done, by using the ‘Load settings’ option in the ‘File’ menu. An open dialog will be shown, from which the settings file can be opened.

H.6 Exporting the results

It also is possible to export the results of the algorithms, which extensively has been done for this report. The ‘Export’ option in the ‘File’ menu will open the export window, as shown in Figure H.6.

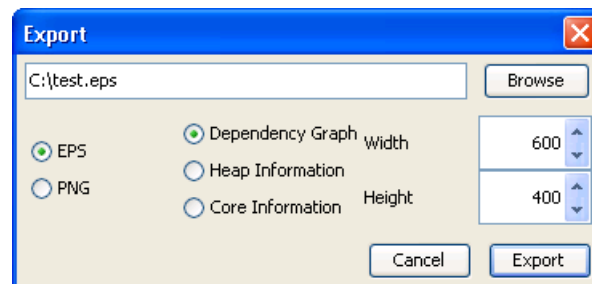


Figure H.6, Export results

The textfield and the ‘Browse’ button behaves similar compared to those in the store settings windows. The two supported export formats are shown at the left of the figure. The ESP format

should be used when a vector drawing should be created and the PNG format for a bitmap export. A vector export results in a better and scalable result, but is not supported in all applications.

In the middle, the algorithm results panels are present in order to choose one for the export. The exported file looks exactly like the panel itself, so the panel settings for colours, positioning and sizes should be set on beforehand. The only different setting is the size of the exported drawing, which can be controlled with the right two fields. Depending on these sizes texts might be fit or not in the process boxes. So it might be required to play around a little to get a optimal export result.

In the future a preview might become available in order to immediately see the result of the sizes and speeding up export process. Settings to change the text sizes might become available as well, so the sizes of the process boxes can become smaller when a smaller text size is chosen.

Bibliography

- 3TU (2008), Dutch Robotics.
<http://www.dutchrobotics.net/>
- Altova (2008), XML editor for modeling, editing, transforming, and debugging XML technologies.
http://www.altova.com/products/xmlspy/xml_editor.html
- van den Berg, B. (2006), *Design of a Production Cell Setup*, Master's thesis, University of Twente.
http://www.ce.utwente.nl/rtweb/publications/MSc2006/pdf-files/016CE2006_vdBerg.pdf
- Boillat, J. E. and P. G. Kropf (1990), A fast distributed mapping algorithm, in *CONPAR 90: Proceedings of the joint international conference on Vector and parallel processing*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 405–416, ISBN 0-387-53065-7.
- Broenink, J. F., M. A. Groothuis, P. M. Visser and B. Orlic (2007), A Model-Driven Approach to Embedded Control System Implementation, in *Proceedings of the 2007 Western Multiconference on Computer Simulation WMC 2007, San Diego*, Eds. J. Anderson and R. Huntsinger, SCS, San Diego, San Diego, pp. 137–144, ISBN 1-56555-311-X.
http://eprints.eemcs.utwente.nl/9351/01/102CE2007_WMC07_pdf.pdf
- Controllab Products (2008), 20-sim.
<http://www.20-sim.com/>
- Free Software Foundation (2008), Bison - GNU parser generator.
<http://www.gnu.org/software/bison/>
- Hilderink, G. (2005), *Managing complexity of control software through concurrency*, Ph.D. thesis, University of Twente.
http://doc.utwente.nl/50746/1/thesis_Hilderink.pdf
- Hilderink, G., J. Broenink and A. Bakkers (1997), Communicating Java Threads, in *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50, IOS Press, Netherlands, University of Twente, Netherlands, volume 50, pp. 48–76.
http://www.ce.utwente.nl/rtweb/publications/1999/pdf-files/014_R99.pdf
- Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice Hall International.
<http://www.usingcsp.com/cspbook.pdf>
- Jovanovic, D. S., B. Orlic, G. K. Liet and J. F. Broenink (2004), Graphical Tool for Designing CSP Systems, in *Communicating Process Architectures 2004*, pp. 233–252, ISBN 1-58603-458-8.
<http://doc.utwente.nl/49238/1/jovanovic04gcsp.pdf>
- Magott, J. (1992), Performance evaluation of communicating sequential processes (CSP) using Petri nets, in *IEE PROCEEDINGS-E*, pp. 237–241.
- Microsoft Corporation (2008), List of Microsoft XML Parser (MSXML) versions.
<http://support.microsoft.com/kb/269238/>
- Molanus, J. (2008), *Feasibility analysis of QNX Neutrino for CSP based embedded control systems*, Master's thesis, University of Twente.
http://www.ce.utwente.nl/rtweb/publications/MSc2008/pdf-files/032CE2008_Molanus.pdf
- Parr, T. (2008), ANTLR Parser Generator.
<http://www.antlr.org/>
- van Rijn, L. (1990), *Parallelization of model equations for the modeling and simulation package CAMAS*, Master's thesis, University of Twente.
- Roscoe, A. (1997), *The Theory and Practice of Concurrency*, Prentice Hall.
<http://web.comlab.ox.ac.uk/people/Bill.Roscoe/publications/68b.pdf>

- Spath, B., O. Faust and A. R. Allen (2006), Portable CSP Based Design for Embedded Multi-Core Systems, in *Communicating Process Architectures 2006*, Eds. F. R. M. Barnes, J. M. Kerridge and P. H. Welch, pp. 123–134, ISBN 1-58603-671-8.
http://www.wotug.org/paperdb/send_file.php?num=164
- van der Steen, T. (2008), *Design of animation and debug facilities for gCSP*, Master's thesis, University of Twente.
http://www.ce.utwente.nl/rtweb/publications/MSc2008/pdf-files/020CE2008_vdSteen.pdf
- Sunter, J. (1994), *Allocation, Scheduling & Interfacing*, Ph.D. thesis, University of Twente.
- The Eclipse Foundation (2008), The Eclipse Modeling Framework (EMF).
<http://www.eclipse.org/modeling/emf>
- The Flex Project (2008), flex: The Fast Lexical Analyzer.
<http://flex.sourceforge.net/>
- Thomason, L. (2008), TinyXML.
<http://www.grinninglizard.com/tinyxml/>
- Veillard, D. (2008), The XML C parser and toolkit of Gnome.
<http://www.xmlsoft.org/>
- Veldhuijzen, B. (2008), Redesign of the CSP execution engine, to be published (report 036CE2008).
- World Wide Web Consortium (2008), Extensible Markup Language (XML).
<http://www.w3.org/XML/>