Communicating Process Architectures 2009
Peter Welch, et al. (Eds.)
IOS Press, 2009
© 2009 The authors and IOS Press. All rights reserved.
doi: 10.3233/978-1-60750-065-0-67

Analysing gCSP Models Using Runtime and Model Analysis Algorithms

M. M. BEZEMER, M. A. GROOTHUIS and J. F. BROENINK

Control Engineering, Faculty EEMCS, University of Twente,
P.O. Box 217 7500 AE Enschede, The Netherlands.

{M.M.Bezemer, M.A.Groothuis, J.F.Broenink} @utwente.nl

Abstract. This paper presents two algorithms for analysing gCSP models in order to improve their execution performance. Designers tend to create many small separate processes for each task, which results in many (resource intensive) context switches. The research challenge is to convert the model created from a design point of view to models which have better performance during execution, without limiting the designers in their ways of working. The first algorithm analyses the model during run-time execution in order to find static sequential execution traces that allow for optimisation. The second algorithm analyses the gCSP model for multi-core execution. It tries to find a resource-efficient placement on the available cores for the given target systems. Both algorithms are implemented in two tools and are tested. We conclude that both algorithms complement each other and the analysis results are suitable to create optimised models.

Keywords. CSP, embedded systems, gCSP, process scheduling, traces, process transformation

Introduction

Increasingly machines and consumer applications contain embedded systems to perform their main operations. Embedded systems are fed by signals from the outside world and use these signals to control the machine. Designing embedded systems becomes increasingly complex since the requirements grow. To aid developers with this complexity, Communicating Sequential Processes (CSP) [1, 2] can be used.

The Control Engineering (CE) group has created a tool to graphically design and debug CSP models, called gCSP [3, 4]. The generated code makes use of the Communicating Threads (CT) library [5], which provides a C++ framework for the CSP language. An overview is shown in Figure 1.

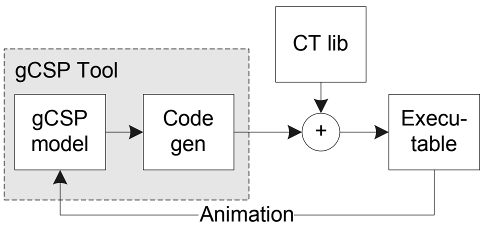


Figure 1. gCSP & CT framework

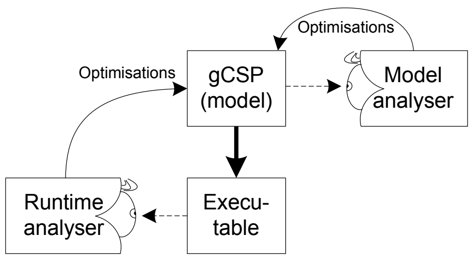


Figure 2. Locations of the analysers

gCSP allows designers to create models and submodels from their designer's point of view. When executing a model, it is desired to have a fast executing program that corresponds to the modelled behaviour. Transformations are required to flatten (submodels are removed) the model as much as possible, to represent the model in its simplest form, in order to be able to schedule the processes using the available resources without having too much overhead. This paper presents the results of the research on the first step of model transformations: it looks for ways to analyse models in order to retrieve information about ways to optimise the model. The transformations themselves are not included yet.

An example of such resources are multiprocessor chips as described in [6]. It is problematic to schedule all processes efficiently on one of these processors by hand, keeping in mind communication routes, communication costs and processor workloads. Researching the conversion of processes from a design point of view to a execution point of view is the topic of this paper.

Related Work

This part of the related work shortly describes work on algorithms to schedule processes on available resources. It finishes explaining which algorithm was chosen as a basis for this research.

Boillat and Kropf [7] designed a distributed algorithm to map processes to nodes of a network. It starts by placing the processes in a randomised way, next it measures the delays and calculates a quality factor. The processes which have a bad influence on the quality factor, are mapped again and the quality is determined again. The target system is a Transputer network of which the communication delays and available paths differ depending on the nodes which need to communicate. So the related processes are automatically mapped close to each other by this algorithm.

Magott [8] tried to solve the optimisation evaluation by using Petri nets. First a CSP model is converted to 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 according to his theorems.

Van Rijn describes in [9] and [10] an algorithm to parallelise model equations. He deduces a task dependency graph from the equations and used those to schedule the equations on a Transputer network. Equations with a lot of dependencies form heaps and should be kept together to minimise the communication costs. Furthermore, his scheduling algorithm tries to find a balanced load for the Transputers, while keeping the critical path as short as possible.

In this work, van Rijn's algorithms are used as a basis for the model analysis algorithms, mainly because the requirements of both sets of algorithms are quite similar. The CSP processes can be compared to the model equations, both entities depend on predecessors and their results are required for other entities. The target systems both consist of (a network) of nodes having communication costs and or similar properties. However, van Rijn's algorithm is not perfectly suitable for this work, so it will be extended into a more CSP dedicated algorithm.

The results of the runtime analyser can be presented in different ways. Brown and Smith [11] describe three kinds of traces: CSP, VCR and structural traces, but they indicate that even more possibilities of making traces visible are available. Because the runtime analyser tries to find a sequential order in the execution of a gCSP model, the results of the tool use the sequential CSP symbol '->' between the processes. This way of showing the sequential process order is quite intuitive, but for bigger models it gets (too) complex, a solution is presented in the recommendations section.

Goals of the Research

Designing and implementing algorithms to analyse gCSP models is the main goal of this research. It must be possible to see from the algorithm results how the performance of the

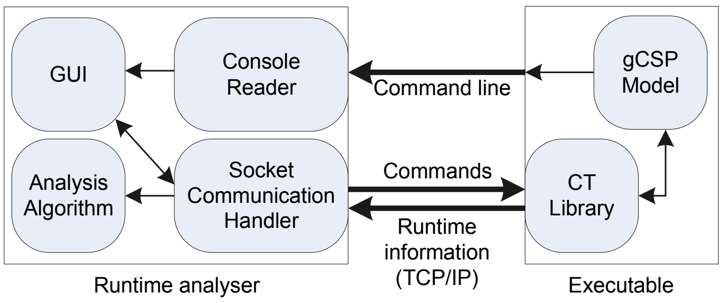


Figure 3. Architecture of the analyser and the executable

execution of the model can be optimised in order to save resources. Two methods of analysis are implemented as shown in Figure 2.

- The *runtime analyser* looks at the executing model and its results can be used to determine a static order of running processes.
- The *model analyser* looks at the models at design time and schedules the processes for a given target system.

Both analysers return different results due to their different points of view. These results can be combined and used to make an optimised version of the model, which is currently still manual labour.

Outline

First the algorithms and results of the runtime analyser are described in Section 1. Section 2 contains the algorithms and results of the Model Analyser. The paper ends (Section 3) with conclusion about both analysers and some recommendations.

1. Runtime Analyser

This section describes the algorithms and the implementation of the 'gCSP Runtime Analyser', called runtime analyser from now on. The algorithm tries to find execution patterns by analysing a compiled gCSP program.

Hilderink writes in [12] (section 3.9) about design freedom. He implies that the designer does not need to specify the process order of the complete mode, but leave it to the underlying system. The runtime analyser sits between the model and the underlying system, the CT library. It suggests static solutions for (some of) the process orders, so the underlying system gets (partly) relieved from this task, making it more efficient.

1.1. Introduction

Figure 3 shows the runtime analyser architecture with respect to the model to be analysed. In order to execute a model, it is compiled together with the CT library into an executable. This can be done by the tool or manually if compiling requires some extra steps. The main part of the analyser consists of the implementation of the used algorithm. The executable gets started after compilation and communication between the tool and the executable is established. After the executable is started it initialises all processes and waits for the user to either run or step through the executable.

Communication is required to feed the algorithms with runtime information. Therefore two types of communication are used:

- **TCP/IP** communication, which is send over a TCP/IP channel between the CT library and the Communication Handler. It is originally used for animation purposes

[4, 13], but the runtime analyser reuses it to send commands to the executable, for example to start the execution and to receive information about the activated processes and their states.

- **command line** communication, consisting of texts normally visible on the command line. It is used for debugging only, the texts will be shown to the user in a log view.

The processes in a gCSP model have several states, which are used by the algorithms: *new*, *ready*, *running*, *blocked* and *finished*. Upon the creation of the process it starts in the *new* state. When the scheduler decides that a process is ready to be started it is put in the ready queue and its state is changed to the *ready* state. After the scheduler decides to start that process, it enters the *running* state. From the running state a process is able to enter the *finished* or the *blocked* state, depending on whether the process was finished or was blocked. Blocking mostly occurs when rendezvous communication is required and the other site is not ready to communicate yet. From the finished state a process can be put in the ready state again via a restart or via the next execution of a loop. When the cause for the blocking state is removed, the process is able to resume its behaviour when it is put in the running state again.

1.2. Algorithms

The runtime analyser uses two algorithms: one to reconstruct the model tree and the other to determine the order of the execution of the processes. Both algorithms run in parallel and both use the information from changes in the process states. Before the algorithms can be started the initialisation information of the processes is required to let the algorithms know which processes are available. This initialisation information is sent upon the creation of all processes, when they enter their new state. This occurs before the execution of the model is started.

1.3. Model Tree Construction Algorithm

Model tree (re)construction is required because the executable, which is analysed, does not contain a model tree anymore. Using the model tree from the gCSP model is difficult, because the current version of gCSP has unusable, complex internal data structures. This will be improved in the new version (gCSP2), but till then the model tree construction algorithm is required. An advantage of reconstructing the tree is the possibility to be sure that the used gCSP model tree corresponds with the model tree hidden in the code of the executable. If both model trees are not equal an error has occurred. For example the code generation might have generated incorrect code or communication between the executable and the analyser is corrupted.

Figure 4 shows the reconstructed model-tree of the dual ProducerConsumer model shown in Figure 15. The numbers between parenthesis indicate the number of times a process got finished. This is shown for simple profiling purposes, so the user is able to see how active all processes are.

Processes are added to the tree when they are started for the first time. All grouped processes, indicated with the dashed rectangles in Figure 15, appear on the same branch. In Figure 4 the label **B** shows a branch, which is made black for this example. The two processes on this branch, Consumer1 and Consumer2, are grouped and belong to Seq_C. The grouped processes indicate a scheduling order, in this example they have a sequential relationship as shown by the CSP notation right of the Seq_C tree node.

The rule for the model tree creation, but also for process ordering, are based on the internal algorithms, which are used by the CT library to activate and execute the processes. When a processes becomes active it creates its underlying processes and adds them to the ready queue of the scheduler. Therefore one rule is sufficient to reconstruct the model tree, as shown in Rule 1

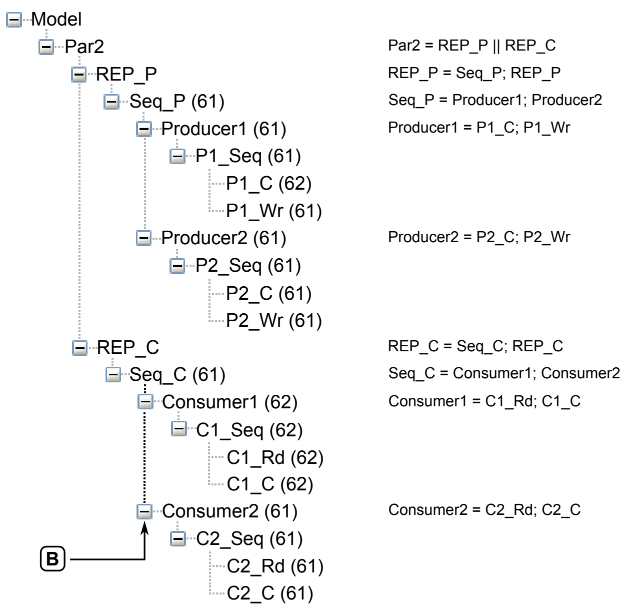


Figure 4. A reconstructed model tree of the dual ProducerConsumer model, with the CSP notation at the right

If a process enters its ready or running state for the first time, make it a sibling of the last started process, which is still running. **Rule** 1

In order to make the rule more clear the creation of the shown model tree, based on the behaviour of the scheduler build in the CT library, will be discussed. First 'Model' is added as the root of the tree and marked as last started process even though it is not a real process. On the first step Par2 enters its running state and thus added to 'Model'. Since Par2 is a parallel construct, both REP_P and REP_C enter their ready state and both are added to Par2, since this was the last process entering its running state. In this example REP_P enters its running state, since it is added already nothing happens. Now Seq_P enters its running state, so it is added to REP_P and so on, until P1_C enters its finished state and P1_Wr enters its running state. P1_Wr is not added to P1_C because that process is not running anymore, instead the last running running process is P1_Seq. After P1_Wr blocks, no rendezvous is possible yet, REP_C enters its running state, since it already as added to the tree nothing happen. Seq_C enters its running state and is added to REP_C since that is last running process.

This keeps on going until all processes are added to the tree and are removed of the list of unused processes. When this list is empty, the reconstruction algorithm is finished. Otherwise, it may be an indication of (process) starvation behaviour or a modelling error.

1.4. Process Ordering

Like the model tree construction algorithm, the process ordering algorithm also uses the changes in the process states. This time, only the state change to 'finished' is used, since the required process order is the finished order of the processes and not the starting order.

The result of the algorithm is a set of chains of processes showing the execution order of the executable. A chain is a sequential order of a part of the processes, it ends with one or more cross-references to other chains. The algorithm operates in two modes: one for chains which are not complete and therefore not have set cross-references to other chains and one for chains which are finished and have cross-references. A chain gets finished when the same process is added for a second time to this chain, after it is finished no processes can be added to it. The reason to finish chains is that they are not allowed to have the same process add to it twice, as this would result in a single chain which gets endlessly long, instead of having a set of multiple chains showing the static running order of the processes. Therefore, two sets of rules are required one for each mode.

The combination of the created chains should be equal of the execution trace of the model. So basically the set of chains is another notation of the corresponding trace, so by following the chains and their cross-references the trace can be reconstructed.

1.4.1. Used Notation

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

Figure 5. Example notation of a set of chains

The notation of the chains is shown in Figure 5. Each chain has a unique starting process followed by a series of processes and is ended by one or more cross-references. The cross-references indicate which chain(s) which might become active after the current chain is ended and are shown at the end of a chain between parentheses. Asterisks behind a cross-reference 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 process in the middle of its chain. During the explanation of the algorithm there always is an active chain, indicating that the the current execution is somewhere in that chain. This chain is shown in bold face and starts with an asterisk, in this example notation the chain starting with **B** in the example is the active one.

The execution order starts at starting point '[start]'. In the example of the used notation, the chain starting with **B** is activated after the '[start]' chain is finished, since the cross-reference points to this chain. At the end of chain **B** two cross-references are available and the outcome depends on the underlying execution engine. Either the current chain stays active and continues at process **E** or the chain starting with **D** becomes active and so on.

1.4.2. Rules for Chains with no Cross-References

These rules are responsible for creating and extending new chains. The trace shown in Figure 6 is used for the explanation, the numbers above the trace indicate a position, which is referred to in the explanation.

```
      1  2              3
      |  |              |
A->B->C->B->D->E->F->B
```

Figure 6. The used trace

```
*[start]->A->B->C
```

Figure 7. A new chain

*If the state of process **p** changes to 'finished' add **p** to the end of the active chain.* **Rule** 2

Rule 2 is the most basic rule to create new chains. When position 1 is reached, the result is a chain with processes **A**, **B** and **C** added, as shown in Figure 7. 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 and 4 are required.

*If process **p** is finished, but is already present in the active chain, it will become an cross-reference of this chain.* **Rule** 3

*If process **p** becomes the cross-reference of a chain, the chain starting with **p** is looked up.*
*If the chain is not available a new chain starting with **p** will be created.*
The existing or newly created chain will become the new active chain. **Rule** 4

```
*B
 [start]->A->B->C->(B)
```

Figure 8. A second chain is added

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

Figure 9. The complete trace converted to chains

Rule 3 defines when a chain should be ended. When a chain that starts with process **B** is not found, Rule 4 states that a new chain should be added. The result is shown in Figure 8.

When position 3 is reached, process **B** again applies to Rule 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 9.

1.4.3. Rules for Chains with Cross-References

When the active chain is finished and thus has one or more cross-references, the algorithm should check if this chain is correctly created. A chain is correctly created if it is valid for the complete execution of the model, when this is not the case the chain needs modification. In order to check whether a chain is correct or not, the position in the currently active chain is required. To explain these rules the trace is extended, as shown in Figure 10.

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

Figure 10. The extended trace

First of all it should be checked if the active process is at the end of the chain. Position 3 ends the chain starting with **B** shown in Figure 9. Rule 5 performs this check and find the referenced chain, in this case it is the same chain.

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** 5

When the chain is not ended by the finished process, the process should match the next process in the chain, this results in Rule 6. At position 4, process **D** finishes. Previously process **B** was finished, so process **D** is expected next. No problems arise since the expected and the finished processes match.

If the active process does not match the finished process ***p*** *the chain must be split.* **Rule** 6

At position 5 process **G** finished, instead of the expected process (**E**), which comes after the previous process (**D**) in the chain. 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 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 the current chain before process ***e***

Put the processes after ***e*** *and the cross-references in a new chain starting with process* ***e*** **Rule** 7

Add ***e*** *and* ***p*** *as new cross-references*

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 split chain. A new chain is created starting with process **G** and is made active. The resulting chains are shown in Figure 11. When following the trace from the start till position 5, the set of chains in the figure match 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.

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

Figure 11. Chains after splitting chain **B**

1.4.4. Rule for splitting chains when a matching chain already exists

The steps of Rule 7 can only be used if a chain starting with process **e** is not present. If the trace is extended again according to Figure 12, a problem arises at position 7. Until position 6 no problems occurred, Figure 13 shows the resulting set of chains until this position.

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

Figure 12. The final trace

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

Figure 13. 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 according to Rule 6 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, since it would not be deterministic which chain should be activated when a cross-reference to one of these chains is reached. However, it is clear that the processes after process **H** match the chain starting with **B** exactly. Rule 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 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**. Add the cross-references to the chain starting with **e** if they are not present at this chain. Create a new chain starting with **p** and make it the active chain.* **Rule** 8

If both parts are unequal the comparison did not succeed. The static process order cannot be determined and the algorithm should stop.

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

Figure 14. The chains after position 7

Figure 14 shows the result after applying the rule. Process **I** is active and can be added to the set of chains using the described rules.

If the comparison of the rule fails, it indicates that two unequal chains starting with the same process should be created. This situation is not supported by the algorithm, because the executing engine of the CT library in combination with this gCSP model does not result in a static execution order and the model is non deterministic. 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 8 fails.

1.5. Results

This section describes the results of the analyser, first a functional test is performed. Next, a scalability test is performed to see whether the analyser still functions when a bigger real-life model is used.

1.5.1. Functional Test

As a functional test a simple producer consumer model is created, as shown in Figure 15. The Producer and Consumer submodels are shown below the main model.

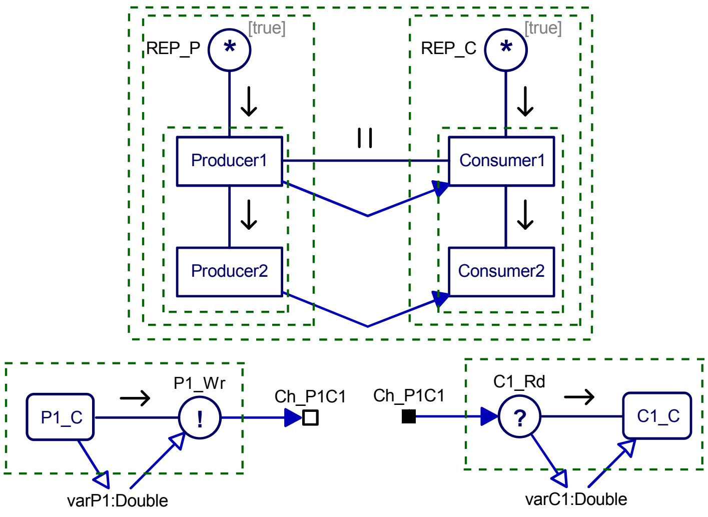


Figure 15. The dual ProducerConsumer model

The model is loaded in the analyser, which automatically generates code using gCSP, compiles and creates the executable. Figure 16 shows the result of the analyser.

```
P1_C->C1_Rd->C1_C->P1_Wr->P2_C->P2_Wr->C2_Rd->C2_C->(P1_C*)
[start]->(P1_C)
```

Figure 16. The process chains for the dual ProducerConsumer model

One big chain is created and the `[start]`-chain is empty indicating that no startup behaviour is present. While keeping in mind that it shows the finished order of the processes, it can be verified that it corresponds with the model. First `P1_C` is finished. Next `P1_Wr` blocks since no reader is active yet, so `C1_Rd` becomes active and reads the value from the channel. After `C1_C` finishes `C2_Rd` blocks since `P2_Wr` is not active yet. `P1_Wr` finally can finish as well, this can be continued for the other Producer-Consumer pair.

The result of the analysis of the model results in a single chain, in order to compare the results of the optimised model against the results of the original model at least two chains are required. Therefore the model of Figure 15 is reduced to a single Producer-Consumer pair, by removing `P2` and `C2`. The results of the analysis of this new model are shown in Figure 17

```
P1_Wr->P1_C->C1_Rd->C1_C->(C1_Rd)
C1_Rd->C1_C->P1_Wr->P1_C->(P1_Wr)
[start]->(C1_Rd)
```

Figure 17. The process chains for the single ProducerConsumer model

The chains are about half the size of the original model, which is as expected, but the extra chain is new. The difference between both chains, is that one chain first writes to the channel and then reads, while the other chain first reads and then writes. When the chain of the original model is closely examined this behaviour is also visible, it is less noticeable since two ProducerConsumer pairs are available this behaviour.

The analysis result now has two chains so a simplified version can be created, which is shown in Figure 18. It has two processes, one for each chain, both containing a code block representing a chain. The repetitions are replaced by infinite while-loops in the code blocks, resulting in a model which is as optimal as possible.

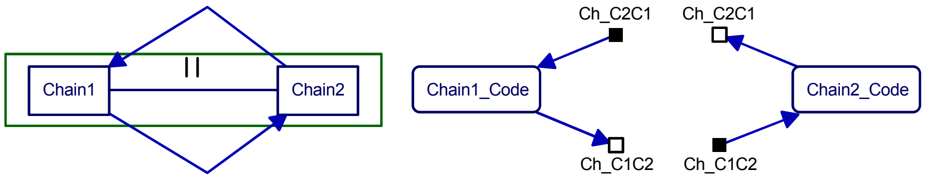


Figure 18. The simplified ProducerConsumer model, rebuild using the runtime analyser results

The original gCSP model needs quite some remodelling in order to be usable for time measurements, as can be seen in Figure 19. Mainly, this is because of the lack of 'init' and 'finish' events for code blocks, so other code blocks are required to simulate these events. Therefore, the results will not be exactly accurate since extra context switches are introduced by the measurement system. Also errors, due to the modified models compared their originals, occur but these errors will be the same for both models and for comparison it does not matter.

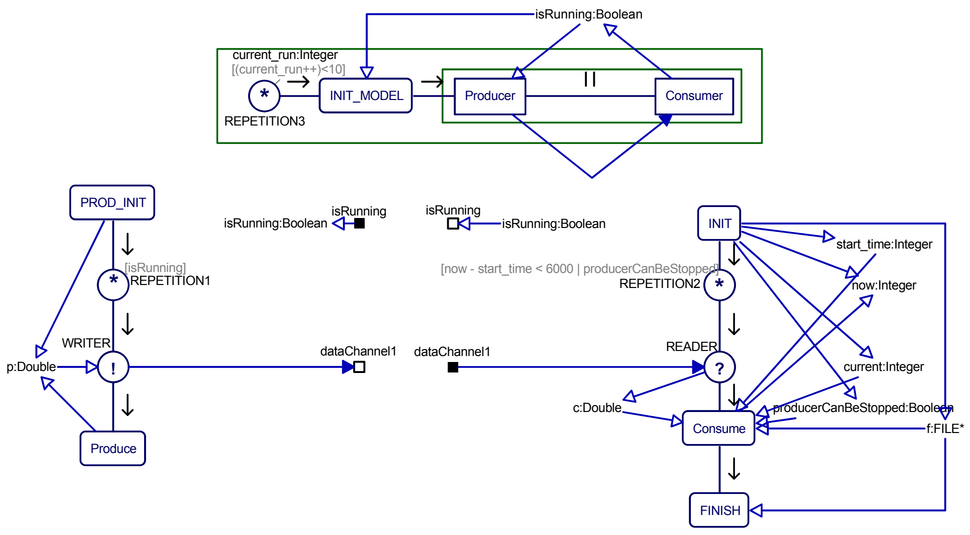


Figure 19. The ProducerConsumer model with measurement added

To test the speed differences between the original and the optimised model, they are build without animation functionality in order to have less external disturbances. For the comparison, the producer and consumer will be running during a certain interval and afterwards the amount of communicated data is stored. This test is repeated 10 times and the average of the resulting values is calculated. In order to get an even more accurate result, this series of 10 tests is repeated 60 times and each average result is plotted in Figure 20. In the figure, the x-axis shows the measurement number and the y-axis shows the average amount of data produced and consumed for the each measurement run. The more data is processed the better, since it indicates that the model is running faster.

It is clear that the optimised model indeed has better results compared to the original model. The average factor between the processed data of both models is 1.3, so the optimised

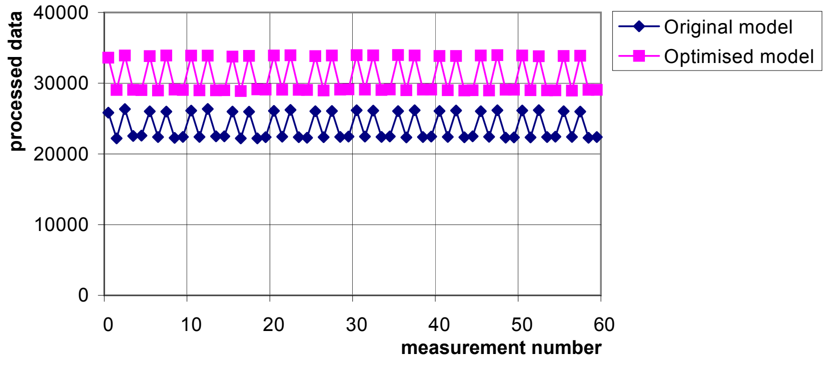


Figure 20. Measurement results

model is 30% faster. Since the optimised model, using the results produced by the runtime analyser, is better than the original, the tool is functional. Whether the speedup is optimal or not, must be determined by performing more tests while taking more details into account, which is out of scope of this paper.

1.5.2. Scalability Test

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 a real control model, in order to see that the analyser is able to handle big models aswell.

The model [14] shown in Figure 21 is used to control a cartesian plotter. The motion sequencer uses a data file to create a motion path for the pen. The motor controllers block contains a 20-sim [15] 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 receives correct signals.

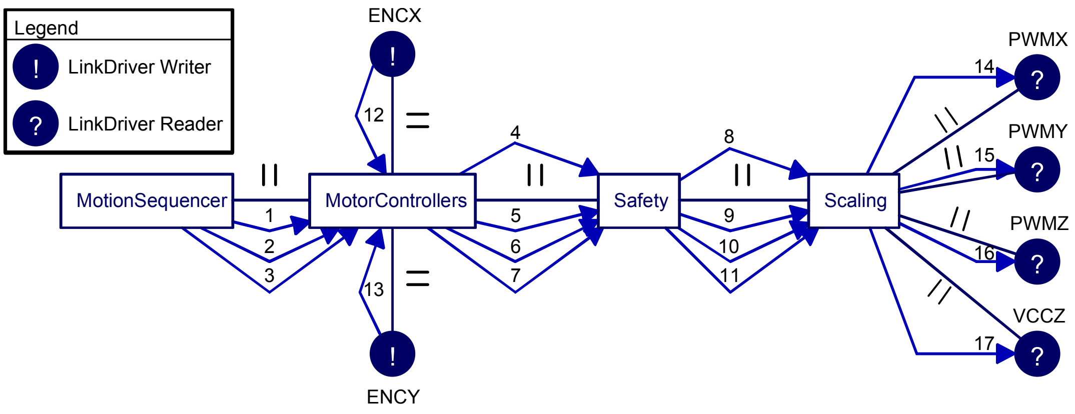


Figure 21. Plotter model

Normally the LinkDrivers, visible in the figure, act as glue between the model and the hardware. For analysis purposes they are implemented to be non-blocking and to generate dummy data, instead of receiving data from the hardware. 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.

After the runtime analyser finished the analysis, the results shown in Figure 22 are obtained. The first two letters of the reader and writer process is the abbreviation of the parent process and the number at the end corresponds with the numbers of the channels in the Figure, for example ‘Sc_Rd8’ is the reader from Scaling which reads the values from channel number 8.

```
Sc_Rd8->DoubletoBooleanConversion->Sc_Wr17->Sa_Wr8->Sa_Rd4->MC_Wr4->Sa_Rd7->MC_Wr7
->Sa_Rd6->MC_Wr6->Sa_Rd5->MC_Wr5->Sa_Rd_ESX2_2->Sa_Rd_ESX2_1->Sa_Rd_ESX1_2
->Sa_Rd_ESX1_1->MC_Rd12->MC_Rd13->Safety_X->Sa_Rd_ESY1->Sa_Rd_ESY2->Safety_Y
->Safety_Z->MC_Rd1->MS_Wr1->MC_Rd2->MS_Wr2->Sa_Wr9->Sc_Rd9->MC_Rd3
->LongtoDoubleConversion->Controller->MS_Wr3->Sc_Rd10->Sa_Wr10->(Sc_Rd11)

Sc_Rd11->DoubletoShortConversion->Sc_Wr14->Sc_Wr15->Sc_Wr16
->Sa_Wr11->(Sc_Rd8, HPGLParser)

MC_Rd12->MC_Rd13->Sa_Rd_ESX2_2->Sa_Rd_ESX2_1->Sa_Rd_ESX1_2->Sa_Rd_ESX1_1->MC_Rd1
->MS_Wr1->Safety_X->Sa_Rd_ESY1->Sa_Rd_ESY2->Safety_Y->Safety_Z->MC_Rd2->MS_Wr2
->MC_Rd3->LongtoDoubleConversion->Controller->MS_Wr3->Sa_Wr9->Sc_Rd9->Sc_Rd10
->Sa_Wr10->(HPGLParser)

HPGLParser->(MC_Rd12, Sc_Rd11)

[start]->MC_Rd12->MC_Rd13->HPGLParser->MS_Wr1->MC_Rd1->MC_Rd2->MS_Wr2->MC_Rd3
->LongtoDoubleConversion->Controller->MS_Wr3->MC_Wr5->Sa_Rd5->MC_Wr6->Sa_Rd6->MC_Wr7
->Sa_Rd7->MC_Wr4->Sa_Rd4->(HPGLParser)
```

Figure 22. Result of the analyser for the plotter controller

This shows that the analyser also works for big, real controller models. It is practically impossible to validate the results. From '`[start]`' to '`Sa_Rd4`' the order seems reasonable. After that part some optimisation 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 optimisation effects are the result of a channel optimisation in the CT library (described in [12] section 5.5.1). They only allow context switches when really required, for example: after reading a result from the channel, the group containing the reading process stays and continues with the execution. As opposed to the possibility that the corresponding writer becomes active again and the flow is more natural. The reader might try to read data from the channel again at some point, but now the channel will be empty, since the writer did not became active yet. Now the reader blocks and the writing process will be activated again. This results in hard to explain analysis results.

On the long run every process is called as often as the other processes, which is as expected. To see whether the results are indeed valid the set of chains should be used to create a optimised model to if the model still runs as expected.

Determining which chain is started when, is not possible from the analysis results themselves. Using the stepping option of the tool, it becomes possible to manually determine that the order of chains is as shown in Figure 23. 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->MC_Rd12->HPGLParser->Sc_Rd11->Sc_Rd8->Sc_Rd11->(HPGLParser*)
```

Figure 23. Execution order of the chains

1.6. Discussion

First of all the runtime analyser seems to work as expected, being able to analyse most (precompiled) models. Two known problems are:

- the use of One2Any or Any2Any channels with three or more processes connected. Models which use these types of channels might fail during analysis, because of the lack of deterministic behaviour of such constructs. In these cases the will give a warning that it is not possible to continue analysing.
- models using recursion blocks or alternative channels. This may lead to processes which are not used during the analysis, because they might depend on (combinations of) external events which might not be available during analysis.

The rules, described in section 1.4, are sufficient for deterministic models and for simple non-deterministic models as well. However, the rules are not proven to be complete, since they are defined by comparing the results of relevant tests and the expected results. When new (non-deterministic) situations are analysed it might be possible that new rules are required.

The bigger the models becomes, the harder the results become to interpret. In the plotter example, it is clear that certain sets of process occur multiple times, like:

```
Safety_X->Sa_Rd_ESY1->Sa_Rd_ESY2->Safety_Y
```

In order to make the results easier to interpret, such sets could be replaced by a group symbol, so the groups can used multiple times.

Currently the results of the analysis are based on a single thread scheduler. When multi-core systems are used and the scheduler of the CT library is able to schedule multiple threads simultaneously, the analysis results are undefined. The processes will be running really in parallel and the received state changes are not representing a sequential order, but represent multiple sequential orders. To solve this problem the received state changes should be accompanied by thread information, so the multiple sequential orders can be separated. The analysis algorithms can then construct the chains for each thread separately using this extra information.

Another usage possibility of the current runtime analyser tool would be post game analysis [16]. 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 analyser 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, as became clear with the scalability test, usable information still can be obtained. Processes which are unused will not become part of the model tree, this information can be used as a coverage test. Of course these processes might become active depending on external inputs, like a safety checking process that might incorporate processes that only run when unsafe situations occur. This situation might never occur when just analysing a model. It also might indicate that an alternative channel is behaving unexpectedly or that a recursion is infinitely looping. We can however use this information to detect processes which should be reviewed.

A more complex usage of the results is to actually implement them. By using the chains it becomes possible to create big processes containing the processes of each chain. By replacing channels, which are used internally in the new processes, with variables the processes will not block and they also become even simpler. Most processes are control related, these processes mostly do not contain (finite) state machines or very small ones, thus combining these processes will not likely give state space explosion related problems. These steps result in an optimised model which requires less system resources and runs faster as shown in section 1.5.1.

2. Model Analyser

This section describes the developed model analyser tool. After a short introduction, an explanation of the algorithm is given. Next the testing and results are given using the same models as the ones used for the the runtime analyser in section 1. This section ends with conclusions about the model analyser and its usability.

2.1. Introduction

The model analyser consists of several algorithm blocks connected to each other, shown in Figure 24. For each block the corresponding section describing it is shown within the parentheses.

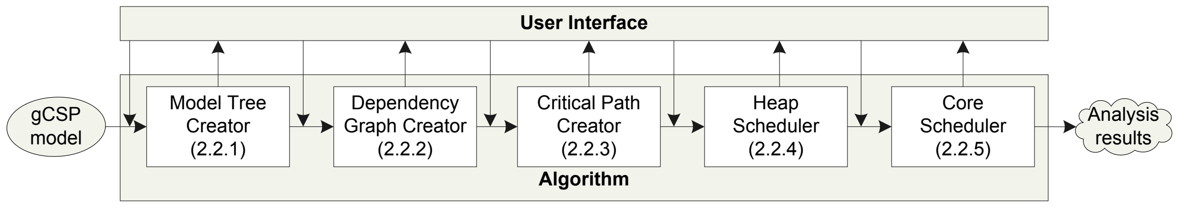


Figure 24. Algorithm steps of the model analyser

The analysis results are calculated using the gCSP model information combined with extra data from the user interface. The following user data is being used by the algorithm:

- **Number of cores**: the number of cores available to schedule the processes on.
- **Core speed**: the relative speed of a core.
- **Channel setup time**: the time it takes to open a channel from one core to another.
- **Communication time**: the time it takes to communicate a value over the channel.
- **Process weight**: the time a process needs to finish after is was started.

The results of the separate algorithms steps are made visible in the user interface. All algorithm steps build in the model analyser are automatic, the user only needs to open a gCSP model and wait for the analysis results. The analyser has a possibility to step through the algorithm steps, in order to find out why certain decisions are made. For example why a certain heap is placed on a certain core, or why several processes are grouped on the same heap.

2.2. Algorithms

The algorithm created by van Rijn [9] is used as stated before. The important parts are based on a set of rules that can be extended, in order to add new functionality to the analyser. Originally it was created to be used for model equations running on a Transputer network. Our improved algorithm is usable for scheduling CSP processes on available processor cores or distributed system nodes.

Scheduling the processes on the available cores is too complex for a single algorithm step, therefore heaps are introduces by van Rijn which can be created by the heap scheduler. Heaps are groups of closely related processes which should be scheduled onto the same core. The heaps can be handled as one process, so the heap scheduler reduces the number of elements to be analysed by the core scheduler and lightens its task. Van Rijn noticed a disadvantage in the use of heaps: they might have lots of dependencies to other heaps, since the process dependencies become heap dependencies after they are placed onto the heaps. Heaps might even contain circular dependencies with other heaps, these circular dependencies cannot be solved by the core scheduler. To solve this problem the processes on heaps are regrouped using index blocks, which start at processes which have an external incoming dependency. The core scheduler is able to schedule these index blocks since they solve the circular dependencies.

The algorithm blocks, shown in Figure 24, are independent blocks chained together. So it is fairly easy to extend these blocks or to add a new one in between, without the need to rewrite surrounding blocks, assuming that the data sent to the existing blocks stays compatible of course. This data is also used by the user interface to update its views after each step.

Unless stated otherwise, the model of Figure 25 is used to create the results shown in the following sections while explaining the behaviour of the blocks. It is the same as Figure 15, but all sub models are exploded to make all dependencies clearer.

2.2.1. Creation of the Model Tree

The model tree is only for informational use and is shown in the user interface of the tool, the algorithm itself does not use it. It is different from the real Model Trees as used in gCSP, this

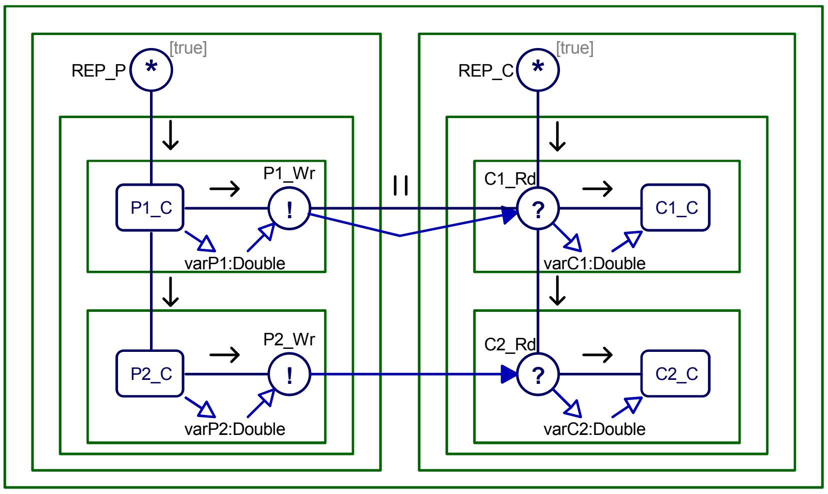


Figure 25. Exploded view of the dual ProducerConsumer model

tree only shows elements which influence the analysis results. This algorithm step is simple:

- it loads the selected model,
- it builds the tree by recursively walking through all submodels to add the available parts,
- and it sends the loaded model to the dependency graph creator.

2.2.2. Dependency Graph Creator

The dependency graph is extensively used by the algorithm blocks to find and make use of the dependencies of the processes. The dependency graph creator recursively walks through the model. First to add all vertices which represent the available processes and a second time to add edges representing the dependencies. The first step, adding vertices, is done by finding suitable processes: code blocks, readers and writers. For the second step, adding edges, sequential relations are used: rendezvous channels and sequential compositions.

Figure 26 shows an example dependency graph. The vertices are created using the sequential relationships, except for the dependencies between the writers and readers. Those two dependencies are derived from the corresponding two channels. The graphical representation also uses the predecessors and the successors to place the vertices in a partial ordered way: in the direction of time flow (i.e. vertical direction in the figure).

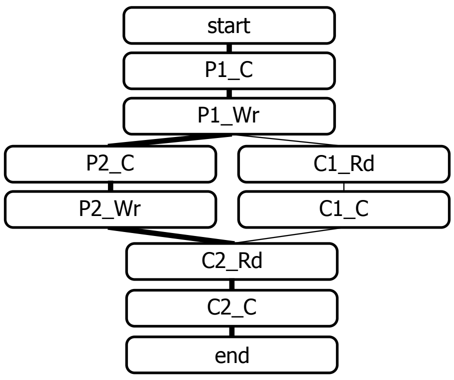


Figure 26. Dependency graph of the Producer-Consumer model

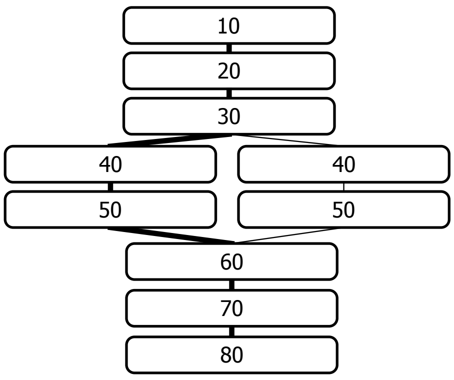


Figure 27. Critical path of the ProducerConsumer model

2.2.3. *Critical Path Creator*

A critical path is required for the heap scheduler, see Figure 24, to schedule the heaps efficiently. Finding this critical path is easy: first the highest cumulative weight for each vertex is determined by following paths from the start vertex to the end vertex and storing the highest calculated weight for each vertex. Secondly the path containing the highest calculated weights is determined to be the critical path. Figure 27 shows the critical path with the thick line, the values in the vertices are the highest cumulative weights of the vertices.

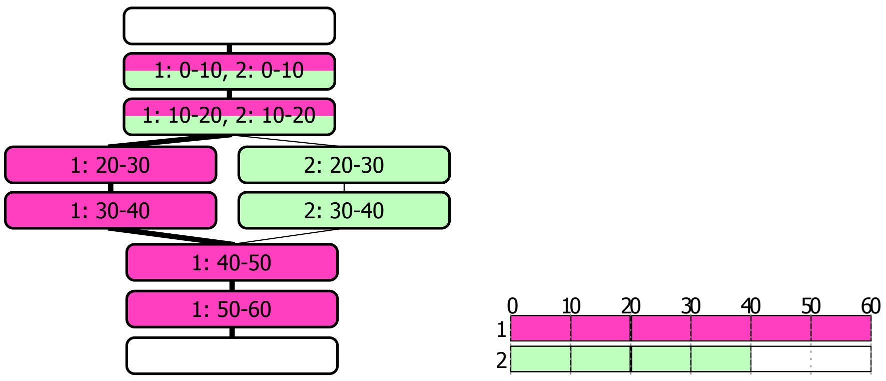
2.2.4. *Heap Scheduler*

As described earlier, the heap scheduler groups processes onto heaps, can be seen as groups of closely related processes and lighten the work of the core scheduler. Each process on a heap has a start time set to indicate when the heap need to start the process. Using this start time and the process weight data, a heap end time can be calculated. Since all processes scheduled on heaps have these values, the heaps can be seen as a sequentially scheduled groups of processes. The heap scheduling algorithm is complex and comprehensive, therefore the complete algorithm and its rules are explained in [17].

In general the algorithm puts the vertices of the critical path in heap 1. The remaining vertices are grouped in sequential chains and assigned to their own heaps. The start time, assigned to each process placed on a heap, is the end time of the previous process. As described before, the end time of a process is calculated by adding the weight of a process to the start time and can be used for the start time of the next vertex on the heap. Sometimes, copying a vertex and its predecessors onto multiple heaps might be cheaper than communicating the result to from one heap to another, so several heaps might contain the same vertices and the start and end times might vary for a process placed on different heaps.

Having processes scheduled on multiple locations does not give any problems in general, because a process receive some values and performs a calculation of some sort and sends the results to another process. The communication takes place by using rendezvous channels, this mechanism makes sure that processes receive the correct information and whether it is calculated multiple times because that is faster does not matter. An exception are processes which have IO-pins, have these kind of processes duplicated is not possible since only one set of IO-pins is available in general.

The result of the example can be found in Figure 28. At the left the dependency graph is shown, the colours of the vertices correspond with the heaps they have been placed on. The texts inside the vertices represent the heap number, the start time and the end time. At the right bars are visible which represent the heaps, with the vertices placed on them in the determined chronological order.



(a) Dependency graph (b) Heap timing

Figure 28. Results of heap scheduler

The figure shows that ‘P1_C’ and ‘P1_Wr’ are scheduled on both heaps, which is understandable since the communication weight is set by default to 25 and the weight of a vertex to 10. So the vertices have a combined weight of 20 and putting it on both heaps saves a weight value of 5, resulting in less workload for the cores.

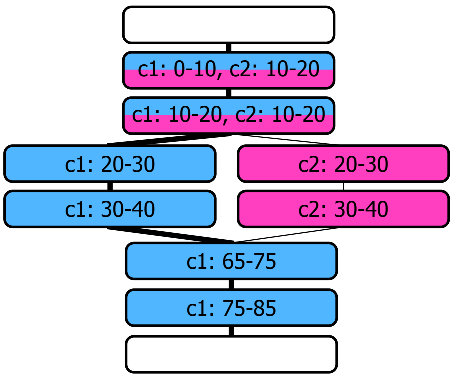
2.2.5. Core Scheduler

The last algorithm block is the core scheduler. This block schedules the heaps onto a number of available cores, which can be seen as independent units, like real cores of a processor or as nodes in a distributed system. The scheduler tries to find an optimum, depending on the amount of available cores and their relative speeds. Like the heap scheduler, the core scheduler is also very complex and comprehensive so only the basics will be explained here.

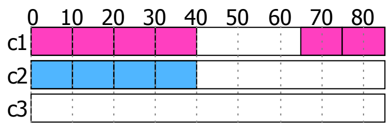
The algorithm is divided in two parts, an index block creator and an index block scheduler part. Index blocks are groups of vertices within heaps, which are easier to schedule than the heaps themselves: for example if two heaps are dependent on each other they cannot be scheduled according to the scheduling rules, since a heap is scheduled after all dependent heaps are scheduled. Therefore index blocks are defined in such a way that this schedule problem and others will not occur.

The actual scheduler consists of many rules to come to a single optimal scheduling result for each index block. A distinction is made between heaps which are part of the critical path and the rest and the other heaps, 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 29 shows the results of the core scheduler in a same way as the heap results are shown. Three cores, named c1, c2 and c3, were made available. Their core names are visible in the dependency graph with the vertex timing behind them. The bars view also shows the core names with the vertices placed on chronological order. The figure shows that core c3 is unused, which is no surprise since only two heaps are available both completely placed on their own core.



(a) Dependency graph



(b) Core timing

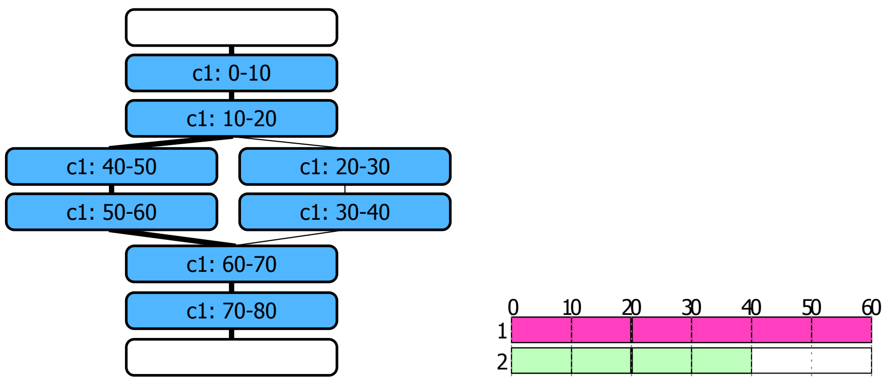
Figure 29. Results of core scheduler

2.3. Results

This section describes the results of the analyser. The setup is the same as the results of the runtime analyser described in section 1.5.

2.3.1. Functional Test

As a functional test the dual producer consumer model is used again, as shown in Figures 15 or 25, but now the number of cores is reduced to one. The result can be seen in Figure 30.



(a) Dependency graph (b) Heap timing

Figure 30. Results of the core scheduler with one available core

Both heaps now are scheduled on the same core, the following constraints should be met in order to pass the functional test:

- 'P1_C' and 'P1_Wr' are available in both heaps, but of course they 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 varying the number of cores, the communication weight or the weight of processes, which is not shown here. In [17] more functional tests are performed and it concludes that the analyser is functional.

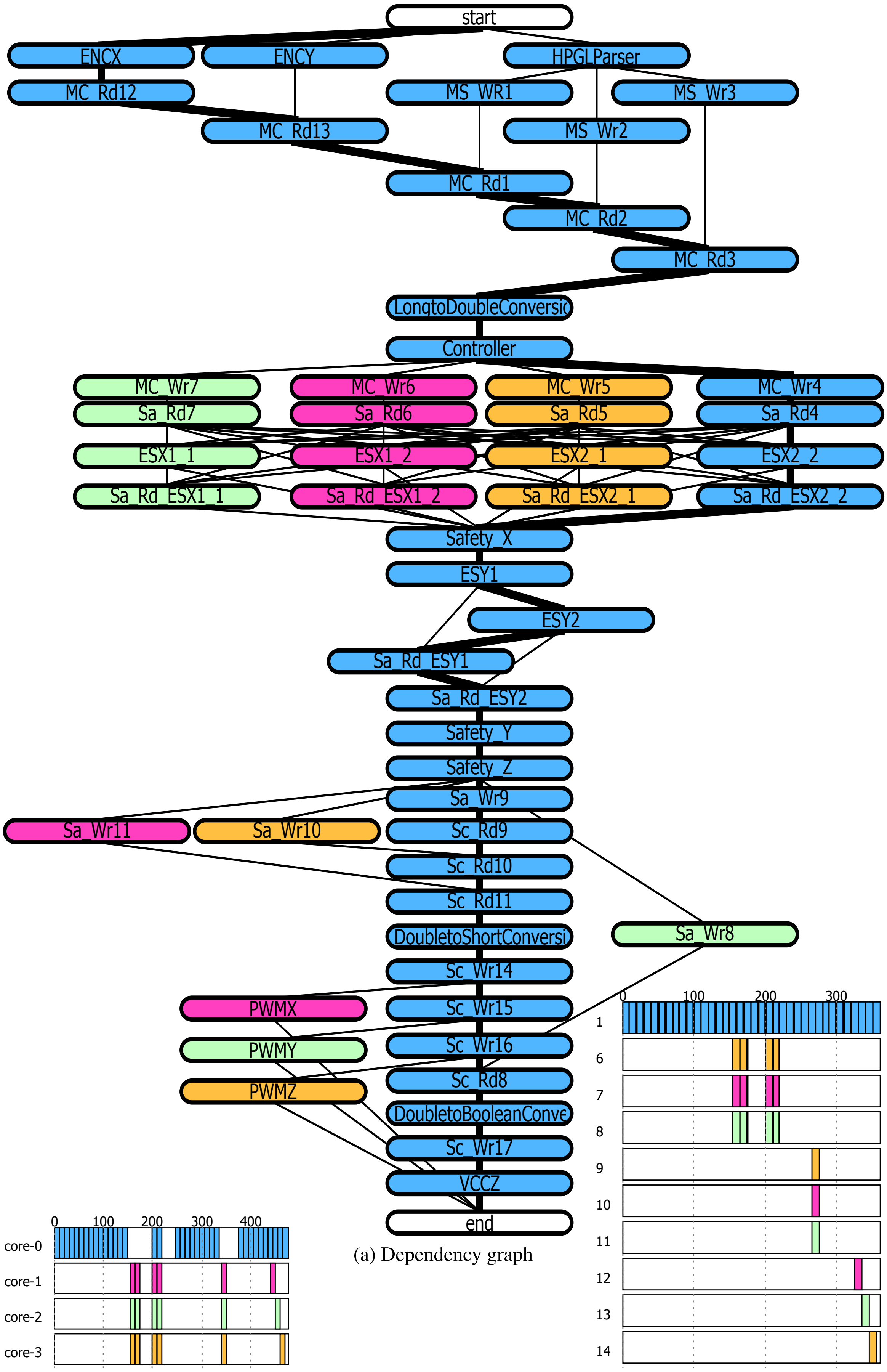
2.3.2. *Scalability test*

The plotter model is analysed and its results are shown in Figure 31. It has a long sequential path, which is expected since the model was designed to be sequential: first sensor information is read, it gets processed, checked for safely and motors are controlled. The model even has sequential paths for the calculations for the X, Y and Z directions of the plotter.

The scalability test shows that the analyser indeed is working for bigger models. In this case, a long sequential path does not give any problems and the analyser is still able to run the algorithms.

When looking at the figure, it becomes clear that the cores are scheduled as optimal as possible, meaning that the end time of the schedule is as low as possible for the current configuration. This is visible by the fact that the gaps in the schedule of the first core are smaller than the combined weight of the processes scheduled on the other cores beneath the gaps: it would not fit to put these processes in the gaps. In order to make the gaps smaller or even to get rid of the them, the communication delays must be made smaller.

By looking at the analysis results it can be concluded that the model should be made more parallel by design when the model needs to run on a multi-core target system. Currently, it is almost completely scheduled onto a single core, mainly due to the amount of dependabilities between the processes resulting in the long critical path. Parallelising the X, Y and Z paths in the model so that the scheduler schedules these paths onto their own cores, results in a model which is more suitable to run on the target. Another solution is creating a pipeline for the four tasks and to put each step in this pipeline on each own core.



(a) Dependency graph

(b) Core timing

(c) Heap timing

Figure 31. Plotter results

2.4. Discussion

As seen in the results section the analyser provides useful information about a model. It allows us to see whether a model is suitable to be scheduled on multiple cores efficiently, or to see how timing is estimated and which processes are the bottleneck depending on the target system. However, a couple of things which need to be improved:

- the comminication costs only have one value which is too simple.
  A delay between the predecessor and the process is added, but in a more realistic situation this is not sufficient. In such situations the delays values should be different for each combination of cores, for example depending on the type of network between the cores or the speed of the cores.
- the communication delays are also made too generic.
  When defining the heaps it is not known whether communication is going to take place or not. Therefore the default communication costs should be removed for the heap scheduler and added to the core schedulers. This should result in less communication costs, since they are only
- 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 heap scheduler need to be improved to produce more balanced heaps.
  Currently heaps tend to be very long, mainly due to the fact that the communication costs are too dominant. To prevent these dominant communication costs, some experiments are required in order to get (relative) values for average communication costs and process weights. These values can then be used as defaults values for the analyser, so the analysis will result in better results. Big heaps are hard to schedule optimal, since a heap is completely placed on one core.
- processes might be node dependent
  This is the case for example processes which depend on external I/O, which probably is wired to one node and the process should be present on that particular node. For these processes a possibility to force them to be scheduled on a particular node would be useful. More information about process-core affinity is described by Sunter [16] section 3.3.

More improvements and recommendations are available in [17].

Besides these recommendations, the algorithm could be improved 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. An example of a problematic situation are circular paths in the dependency graph. These paths indicates that the processes keep on waiting on each other. Circular 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 building a model designer application (like gCSP) this method could be used to check for deadlocks during the design of a model.

Another improvement would be a new algorithm which is able to bundle channels into a multiplexed channel. This would result in less context switches, since multiple writer and readers are combined into a single multiplex writer and reader. After the heap scheduler is finished, it is known which processes will be grouped together. Parallel channels between these groups, could probably be combined into a single channel which transfers the data in a single step.

3. Conclusions and Recommendations

Both algorithms can be used to analyse gCSP models in order to make the execution of gCSP models more efficient, each from a different point of view with different results. The runtime analyser returns set of static chains, which can be used to build a new gCSP model. This model implements each chain into *one* process, which saves context switches and thus system resources. The model analyser presents schedules for gCSP models keeping a given system architecture in mind. The system load, available resources and communication costs are kept as optimal as possible.

However, some work on refining both algorithms must be done. The runtime analyser results appear hard to interpret, so a better way of representing them is required. As described, grouping set of processes which occur more than once, might decrease the lengths of the chains, making the set of chains more understandable. Automatically creating a new optimised model from the runtime analyser results would be the most appropriate solution. As described in section 1.6, support for analysing multi-core applications is required as well, especially when the CT library gets updated to support multiple parallel threads. The section also describes that is not likely to get state space explosion related problems when combining multiple processes into one. It is recommended to look further into this statement in order to see if this indeed is the case.

The model analyser lacks a lot of functionality to make it a more usable tool. As described in section 2.4, things like improvement of communication costs, a new channel multiplexing algorithm and deadlock checking could help to mature the tool and make it part of our tool chain.

After both tools become more usable, it is possible to make them a part of out tool chain. This can be done to integrate them into gCSP2, which also needs more work first, in order to analyse the models while designing them. With the possibility of design time checks new features can be implemented as well, like multi node code generation so multiple sets of code results in multiple executables, one for each node. For this the results of the model analyser need to be included with the code generation of gCSP2, implementing this is not trivial and need more structured work.

From the designer's point of view, having lots of small processes for each task, continues to be a good way of designing models. The results of both algorithms are useful for converting these models to an execution point of view.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful and constructive feedback. Without this feedback the paper would not have been ready for publication as it is now.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. ISBN 0-13-153289-8.

[2] A.W. Roscoe, C.A.R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 0136744095.

[3] D.S. Jovanovic, B. Orlic, G.K. Liet, and J.F. Broenink. Graphical Tool for Designing CSP Systems. In *Communicating Process Architectures 2004*, pages 233–252, September 2004. ISBN 1-58603-458-8.

[4] T.T.J. van der Steen, M.A. Groothuis, and J.F. Broenink. Designing Animation Facilities for gCSP. In *Communication Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering Series*, page 447, Amsterdam, September 2008. IOS Press. ISBN 978-1-58603-907-3. doi: 10.3233/978-1-58603-907-3-447.

[5] G.H. Hilderink, J.F. Broenink, and A. Bakkers. Communicating Java Threads. In *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50, pages 48–76, University of Twente, Netherlands, 1997. IOS Press, Netherlands.

[6] D. May. Communicating process architecture for multicores. In A.A. McEwan, W. Ifill, and P.H. Welch, editors, *Communicating Process Architectures 2007*, pages 21–32, July 2007. ISBN 978-1586037673.

[7] J.E. Boillat and P.G. Kropf. A fast distributed mapping algorithm. In *CONPAR 90: Proceedings of the joint international conference on Vector and parallel processing*, pages 405–416, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 0-387-53065-7.

[8] J. Magott. Performance evaluation of communicating sequential processes (CSP) using Petri nets. *Computers and Digital Techniques, IEE Proceedings E*, 139(3):237–241, May 1992. ISSN 0143-7062.

[9] L.C.J. van Rijn. Parallelization of model equations for the modeling and simulation package CAMAS. Master's thesis, Control Engineering, University of Twente, November 1990.

[10] K.C.J. Wijbrans, L.C.J. van Rijn, and J.F. Broenink. Parallelization of Simulation Models Using the HSDE Method. In E. Mosekilde, editor, *Proceedings of the European Simulation Multiconference*, June 1991.

[11] N.C. Brown and M.L. Smith. Representation and Implementation of CSP and VCR Traces. In *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 329–345, Amsterdam, September 2008. IOS Press. ISBN 978-1-58603-907-3.

[12] G.H. Hilderink. *Managing complexity of control software through concurrency*. PhD thesis, Control Engineering, University of Twente, May 2005.

[13] T.T.J. van der Steen. Design of animation and debug facilities for gCSP. Master's thesis, Control Engineering, University of Twente, June 2008. URL `http://purl.org/utwente/e58120`.

[14] M.A. Groothuis, A.S. Damstra, and J.F. Broenink. Virtual Prototyping through Co-simulation of a Cartesian Plotter. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, number 08HT8968C, pages 697–700. IEEE Industrial Electronics Society, September 2008. ISBN 978-1-4244-1505-2. doi: 10.1109/etfa.2008.4638472.

[15] Controllab Products. 20-sim website, 2009.

[16] J.P.E. Sunter. *Allocation, Scheduling & Interfacing in Real-Time Parallel Control Systems*. PhD thesis, Control Engineering, University of Twente, 1994.

[17] M.M. Bezemer. Analysing gCSP models using runtime and model analysis algorithms. Master's thesis, Control Engineering, University of Twente, November 2008. URL `http://purl.org/utwente/e58499`.