Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

gCSP: A Graphical Tool for Designing CSP Systems[†]

Dusko S. JOVANOVIC, Bojan ORLIC, Geert K. LIET, Jan F. BROENINK
Twente Embedded Systems Initiative,
Drebbel Institute for Mechatronics and Control Engineering,
Faculty of EE-Math-CS, University of Twente,
P.O.Box 217, 7500 AE, Enschede, the Netherlands
d.s.jovanovic@utwente.nl

Abstract. For broad acceptance of an engineering paradigm, a graphical notation and a supporting design tool seem necessary. This paper discusses certain issues of developing a design environment for building systems based on CSP. Some of the issues discussed depend specifically on the underlying theory of CSP, while a number of them are common for any graphical notation and supporting tools, such as provisions for complexity management and design overview.

1. Introduction

In the last two decades of the 20th century the transputer [1], a processor specifically designed for simple parallel processing, was successfully applied in a number of engineering fields but eventually fell out of use. However, the occam language [3], designed for programming systems based on transputers, is still referred to in contemporary text books (for instance, [4]) because of its unique properties that do not yet have a match in industrial state-of-the-art languages. Much of the credit given to the occam language, transputers, and the overall design mindset stems from their foundation in the process algebra CSP (Communicating Sequential Processes) [5]. CSP provides a clear and simple approach for reasoning about concurrent systems. Thanks to its sound mathematical foundation, one of the most needed properties of modern system engineering – formal analysis – is incorporated in the paradigm inherently.

Research efforts have been pursued in both hardware and software application areas of CSP in the post-transputer era. Successful experiences composing complex distributed systems inspired development of several communication platforms and protocols [6-8]. On the software engineering side, CSP influenced the design of Ada and inspired development of occam-like libraries for Java, C and C++ [9-14]; although occam is now rarely used for programming transputers, research on extending the language is still active [15].

Before the proposal for graphical notation for CSP [16] by Hilderink, ways of drawing CSP designs were being adopted for each particular occasion in an ad-hoc manner. This, of course, brings difficulties in communication of ideas and concepts and easily introduces design ambiguities as well. A commonly accepted graphical notation would ease acceptance of CSP by a larger software community and provide great assistance for teaching the CSP notions as well.

[†] This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

The proposed graphical notation has been used in practical applications at the Control laboratory of the University of Twente in the last couple of years. It has proved to be useful for transforming block diagrams to data flow models which can be further refined by introducing relationships describing concurrent aspects of both communication and process composition [17]. Block diagrams are well established for control applications and used in all recognized modelling and simulation tools, such as Matlab/Simulink [18], or – as reported in [17] – 20-SIM [19]. Moreover, CSP diagrams were successfully applied for accomplishing a more complicated task: describing mode switching among a set of control laws developed for different operational modes of a mechatronic system [20].

In [20], CSP diagrams were edited manually in general-purpose drawing tools. The absence of special tools for editing CSP diagrams hampers many other facilities a CSP-based paradigm could offer. Development of a specially crafted CSP design tool permits much more design freedom and reuse, managing complexity through process hierarchy, automatic code generation for concurrent networks, exporting designs to formal checkers and so forth.

In the original proposals of CSP diagrams [16, 21], the graphical CSP language is referred to as GML (Graphical Modelling Language); the tool discussed in this paper is named gCSP (graphical CSP). Basic language and tool elements are summarized in Sections 2, 3 and 4. Section 5 deals with managing complexity in CSP (graphical) designs. The tool's potentials for automatic code generation are presented in Section 6. Section 7 summarizes the status of the tool development and announces further points of attention.

2. General issues of modelling concurrent networks

Composition of any occam program and hence any program based on occam-like libraries is always shaped as a strict tree-like hierarchy of SEQ, (PRI)PAR and (PRI)ALT constructs as branches and user-defined processes as leaves. As such, it can be depicted easily and naturally using a tree hierarchy.

2.1 Tree-based modelling

This fact led to the development of a tree-based CSP design tool [2]. Although the modelling approach based on a strict hierarchical structure provided by the tree resembled the vital compositional aspect of the occam reasoning, the tool's design abilities exhibited two serious flaws.

Firstly, the communication patterns over channel nets were hardly readable from the processes' interfaces scattered over the branches of the tree. An additional view representing data flow was needed.

The second problem was much more serious. The primary aim of implementing such a tool was to assist in designing concurrent programs. A strict hierarchical view can only depict the design that is already shaped as a hierarchy of constructs and processes. But during the design, one starts with process blocks existing in isolation or connected only using data-flow diagrams. Modelling compositional structure in a tree hierarchy does not allow compositional ambiguities (i.e. underspecification) in the course of the design and, therefore, limits the design freedom. In further research, the tree concept was discarded.

Instead, development focused on the fact that during a design process constructs might not yet be formed while compositional relationships between some processes are known. In the follow-up research conducted by Hilderink and Volkerink, the initial idea of GML was conceived.

2.2 GML design principles

A construct can be seen as a set of processes with the same type of relationship between any two processes. In sequential, prioritized parallel, and prioritized alternative constructs, these relationships are not symmetrical and strict order must be maintained within a construct.

In GML, the recommended design process starts with a data flow model. This model is represented by the network of communicating processes. The concurrency structure is then added to this model by specifying compositional relationships between the processes involved. Some concurrency relationships between processes are known in advance and some are subject to various trade-offs. A tool that would allow one to make arbitrary compositional relationships (sequential, parallel, alternative) between any two processes would offer greater flexibility than a tool based solely on the tree compositional hierarchy. GML models can express designs that are not fully specified. Compared to tree views, GML views seem to be better suited for entering designs.

Refining the data flow model with compositional relationships expressing the concurrency structure can be done without changing the layout of the original data-flow model. This feature makes a prospective tool based on GML suitable for use in a chain of tools. In our research group, focus is on development of control systems; one possible predecessor to a GML-based tool in such a chain is 20-SIM [19].

20-SIM is a tool for modelling and simulation of control systems. It can generate code for specified controllers, but the code is generated *after* sequentialization of the data-flow models. It would be advantageous if one could import data-flow models from 20-SIM into a GML tool, extend them with compositional relationships specifying the concurrency structure, perform formal checking, and then automatically generate code free of unwanted concurrency phenomena such as deadlock and livelock.

It is expected that the combination of 20-SIM and gCSP will result in a tool chain that can support the design of control applications, whereas gCSP alone can serve as a graphical tool for using CSP.

3. Purpose of the tool and its specification

The most elaborate standard for describing software graphically, UML [22], is described as "a graphical language for visualizing, specifying, constructing, documenting and communicating the artefacts of a software-intensive system". The same goes for the general idea of GML. In short, the purpose of the language and the tool can be described as "supporting the building concurrent software based on the Communicating Sequential Processes algebra". In order to meet this goal, the development of the tool started with the following set of requirements. The tool should:

1. allow the modelling of concurrent systems using the Graphical Modelling Language (GML) for drawing CSP diagrams.
2. preserve notions of the CSP theory and its peculiarities, but bring it closer to implementation needs.
3. support means for managing complex CSP models – allowing hierarchical organisation by containment relations among parent (complex) and children (leaf or also complex) processes.
4. allow the expression of communication and compositional patterns of process networks, the latter not only in terms of an extended set of CSP constructs, but also in terms of compositional relationships (likewise communication relationships,

which represent CSP channels), as defined in the GML proposal.
5. transform the software models to a number of types of human- and machine-readable code.
6. allow semantic and integrity checks of the specified models.
7. allow visualization also in the domains of (formal) analysis and other imaginable CSP model processing.
8. generate CSP networks suitable for incorporation of operational code derived from other tools (for instance one-shot processes from 20-SIM, as reported in [17] and [20]).

The set of CSP constructs applied and extended in occam with prioritized variants, recently formally described in the work of Lawrence [23], is appended with one more construct for modelling exception handling, as described by Hilderink [16]. Table 3-1 summarizes the set of constructs used in the GML and the gCSP tool.

Table 3-1. Constructs in GML.

Construct symbols	Constructs
→	sequential (SEQ)
||	parallel (PAR)
$\overrightarrow{\|\|}$	priparallel (PRIPAR)
□	alternative (ALT)
$\overrightarrow{\Box}$	prialternative (PRIALT)
$\overrightarrow{\Delta}$	exception (EXCEPTION)

The following vocabulary was derived to describe some important terms for the gCSP tool development:

- *Processes* and *channels* are defined as in the CSP theory [5]. A process in a CSP diagram is depicted as a rectangle, while a channel is represented by an arrowed line.
- SEQ, (PRI)PAR, (PRI)ALT and EXCEPTION are called *constructs*. The occam WHILE loop is proposed in [16] to be represented as the SEQ of the μ primitive process (see Table 3-2) and the process whose repetition is required. Being an idiom of GML, the SEQuences with the μ primitive process are, in the tree of constructs (see discussion of the C-tree in Section 4), optimized to be presented with one repetition construct marked by the μ glyph.
- *Relationships* are represented as lines augmented with the construct symbols (see Table 3-1) that connect processes. Relationships are divided in two sets: *compositional*, that connect (ordered) pairs of a construct's children, and *communication*, channels that connect two or more processes (shared channels are allowed).
- A *CSP diagram* consists of processes and their relationships.
- A *view* on a CSP diagram displays processes and a set of relationships. gCSP can display three standard views: the *communication view* that shows processes and channels (communication relationships), the *compositional view* showing processes and compositional interrelationships, and the *full CSP view* with processes and both

compositional relationships and channels. The topology of the processes is preserved on all the views (compare Figures 4-1, 4-2 and 4-3).

Hilderink's GML proposal [16] includes also the following primitive processes:

Table 3-2. Primitive processes.

Symbol	Primitive processes
(!)	Writer
(?)	Reader
(*)	Barrier sync process
(μ)	Repetition

Use of the μ primitive process is already mentioned in the context of repetitions. Writer and reader primitive processes denote points of communication among processes, as shown on the example in Figure 5-9. On notions of barrier and the use of the barrier synchronisation process the reader is referred to [16].

In the GML proposal grouping of processes into constructs is depicted by attaching indexed bubbles to compositional relationships tied to borders of newly created constructs. For detailed semantics of the grouping mechanism see Subsection 5.1.

4. Graphical user interface

gCSP has a standard windowed user interface (Figure 4-1) that consists of several panes, a menu and toolbars:

- The graphical editor (G-editor) consisting of:
  - *Communication view* and the corresponding toolbar.
  - *Compositional view* and the corresponding toolbar.
  - Full CSP view.
- Tree views:
  - *Compositional tree* that shows organization in constructs (C-tree).
  - *Graphical tree* that summarizes graphical elements in the G-editor (G-tree).
- The *Messages pane* intended for giving feedback to the user.
- A standard window application toolbar extended with a few specific icons for navigating through the model hierarchy in the G-editor (,) and saving and retrieving submodels (,).

Entering a design (inserting processes and relationships) is performed *exclusively* through the views of the graphical editor (G-editor). The full CSP view (as in Figure 4-3), anticipated already in the GML proposal, has been added upon suggestions of users. While the compositional and communication views are clearly suitable for focusing on one of the corresponding architectural aspects, users of the tool suggested that in the design phase it is also handy to have an overview of both compositional and communication relationships at the same time.

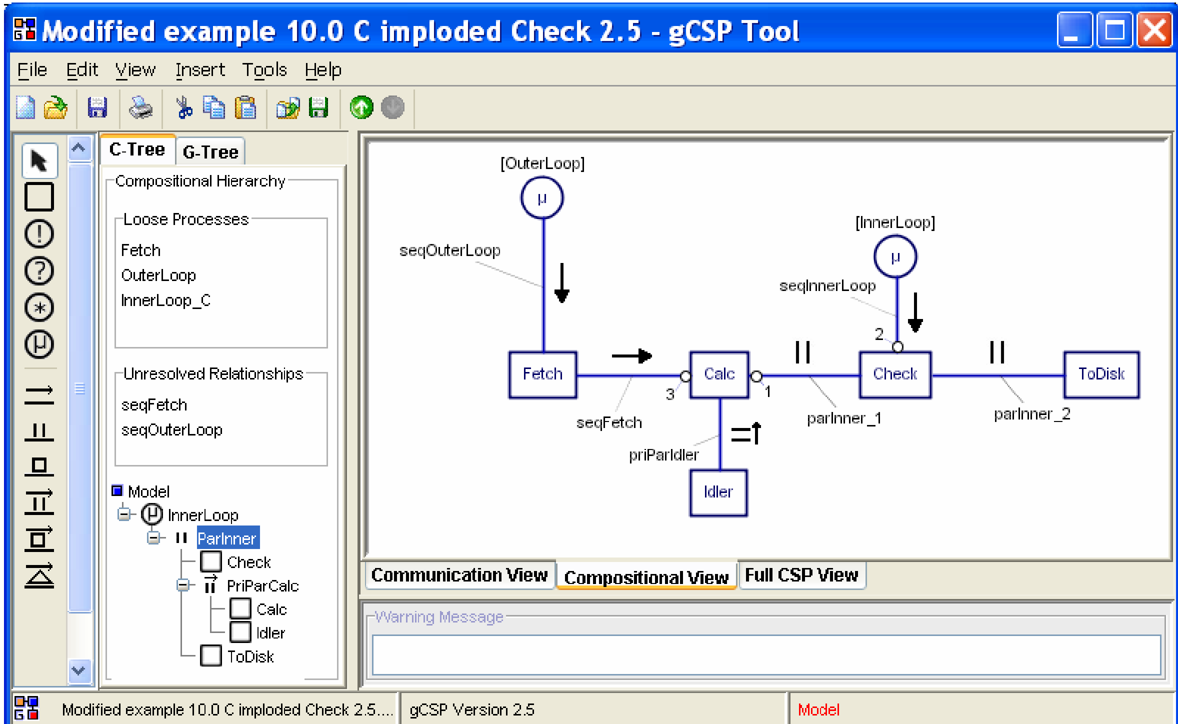


Figure 4-1. Graphical user interface of gCSP (with displayed Compositional View).

While the compositional view can be seen in Figure 4-1, the communication view and the full CSP view are shown in Figure 4-2 and Figure 4-3 respectively.

These figures reveal something that may look rather odd to an occam or a CSP person. The *Fetch* process is composed in sequence with the rest of the system on the right hand side; still, there is a channel in between (*chRawData*). This would certainly cause a deadlock in a CSP model. However, a GML communication relationship (i.e. channel) between processes that run in sequence is not a channel in the CSP sense. In occam it would be a variable. In the CT libraries, due to OOP data encapsulation, variables defined in a parent construct are not automatically visible in a child process. Therefore, in CT, type `ChannelVar` [20] is created as a non-blocking channel that passes variables between sequential processes.

The option to have a channel between sequential processes comes from the GML's intention to capture all interprocess communications by drawing them as channels. The tool is supposed to allow a control engineer to impose explicitly certain sequences of a system's component activations, i.e. in mode switching (see [20]). The sequentialisation in execution would anyway happen in practice if the two processes were composed in parallel – channel synchronization would automatically force them to run in a sequential order. While this is obvious for those familiar with CSP, the intention of the tool is to be receptive to a wider community.

In generating machine-readable CSP (CSP_M scripts) this problem needs to be solved by detecting channels between sequentially composed processes and handling them in a proper way. Making an option for different visualization of the non-blocking channels in gCSP is under consideration.

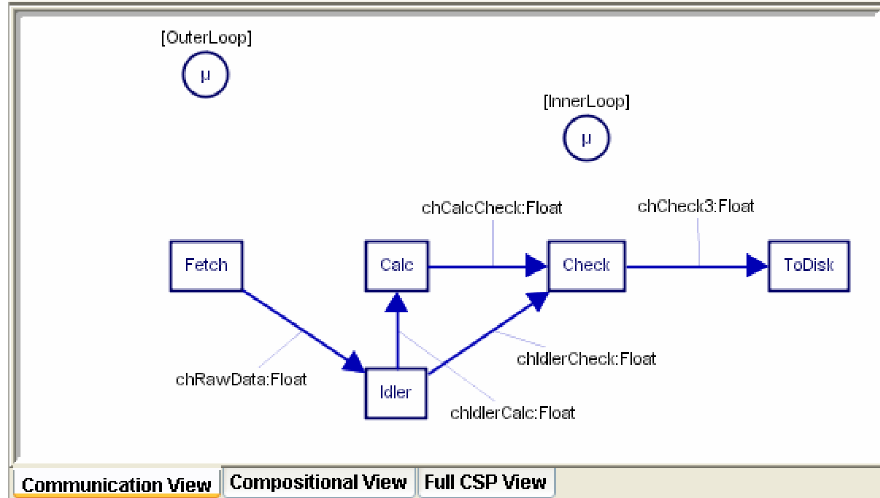


Figure 4-2. Communication view.

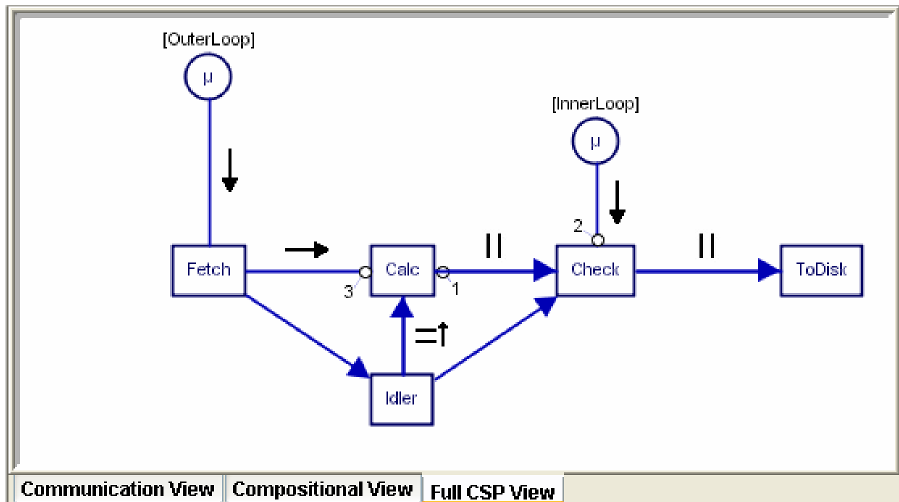


Figure 4-3. Full CSP view.

Trees have shown as the best means for navigation through a model hierarchy. The compositional tree (C-tree in Figure 4-4) also corresponds to a model representation that is most suitable for code generation.

The C-tree actually consists of three compartments:

- list of Loose processes
- list of Unresolved relationships
- trees representing emerging hierarchies of constructs and processes

The term *loose process* is used for a process whose parent construct in the compositional hierarchy is not yet determined. *Unresolved relationship* is a term used for a compositional relationship that connects two processes that are not yet composed in a construct. These issues are discussed in detail in Subsection 5.2.

The G-tree (Figure 4-5) allows inspection of the graphical objects and browsing through the hierarchy; furthermore it assists working with the full CSP view when compositional relationships and channels are overlapping.

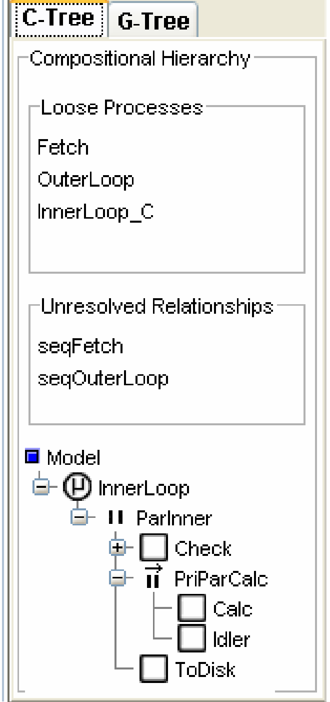


Figure 4-4. Compositional tree.

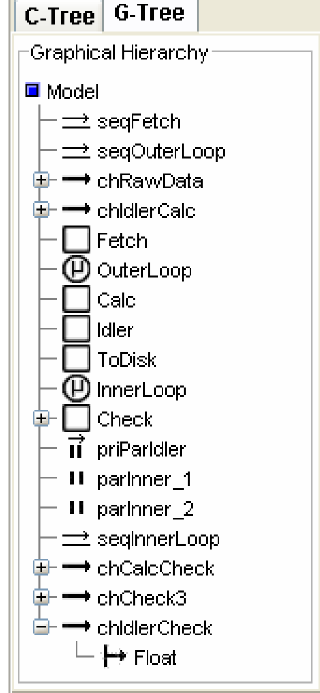


Figure 4-5. Graphical tree.

5. Managing a CSP model complexity

5.1 Representing compositional hierarchies in flat models

A bottom-up approach of building a complex CSP model starts by connecting processes with compositional relationships. Since any process can be connected with many others, some kind of grouping processes and relationships is necessary to establish a proper compositional hierarchy. Otherwise, the model would be compositionally ambiguous.

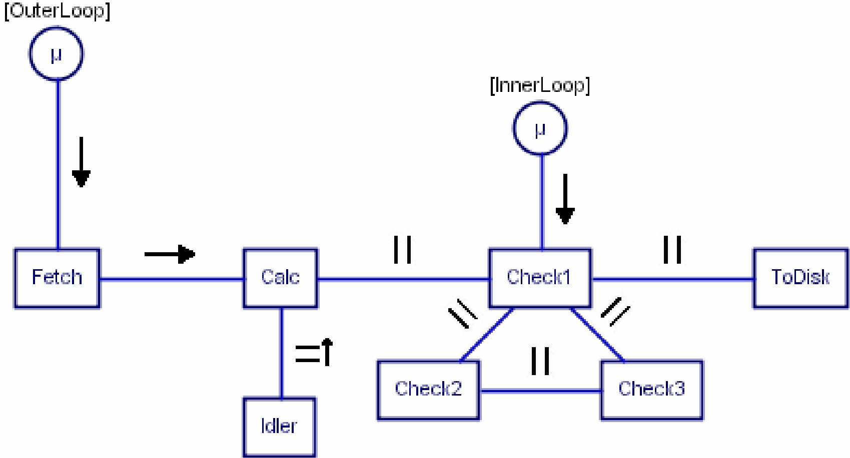


Figure 5-1. An ambiguous model.

Figure 5-1 depicts an example of a compositionally ambiguous model. For instance, it is ambiguous whether the sequence of *Fetch* and *Calc* is composed in a priparallel

composition with *Idler*, or after the termination of *Fetch* the priparallel construct consisting of *Calc* and *Idler* takes place. The reader may spot many other ambiguities as well.

A construct in a compositional graphical view is represented by a *group* of processes connected with compositional relationships of the same kind. An intuitive representation of a construct (group) would be a rectangle ("box") embracing the grouped processes. One possible solution that eliminates compositional ambiguities turns the model from Figure 5-1 into the one depicted in Figure 5-2.

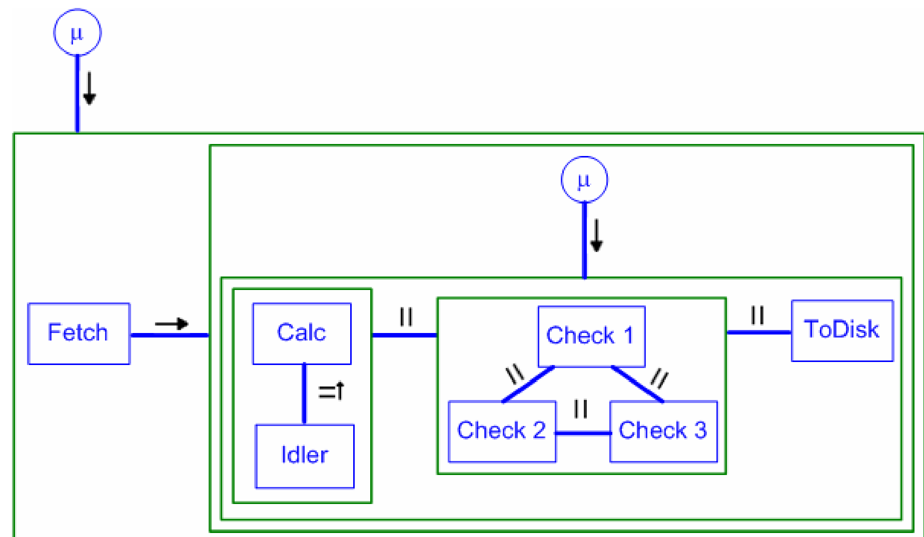


Figure 5-2. Boxed notation.

However, in a complex network, rectangles that mark nested structures take a lot of display space. Adapting this kind of diagram by rearranging the structure may result in a rather serious amount of editing work. Furthermore, maintaining the boundaries of the constructs may be very laborious for a proper GUI development.

In order to compensate for these drawbacks, GML has so-called "parentheses", little indexed bubbles at the end(s) of a composition relationship to indicate the nesting, as in Figure 5-4. An intermediate step between the boxed and parenthesized ("bubbled") representations is given in Figure 5-3. Only parts of rectangles intersecting relationships carry the nesting information.

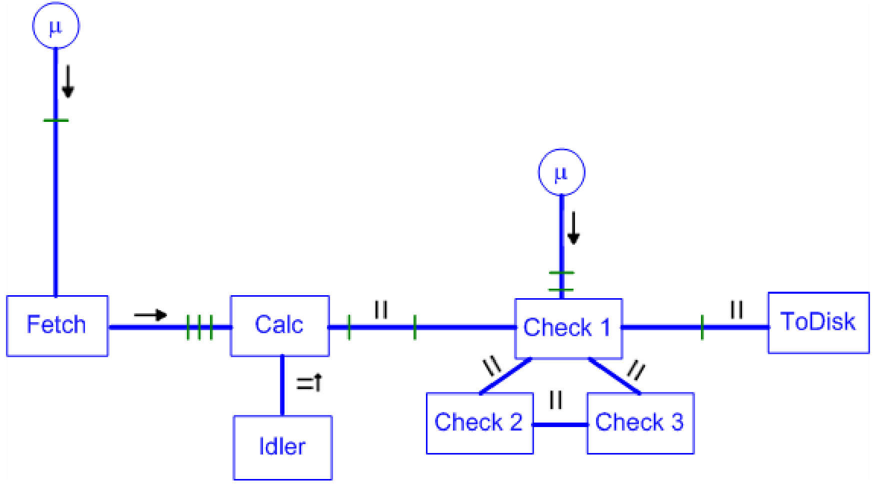


Figure 5-3. The intermediate step between the two grouping notations.

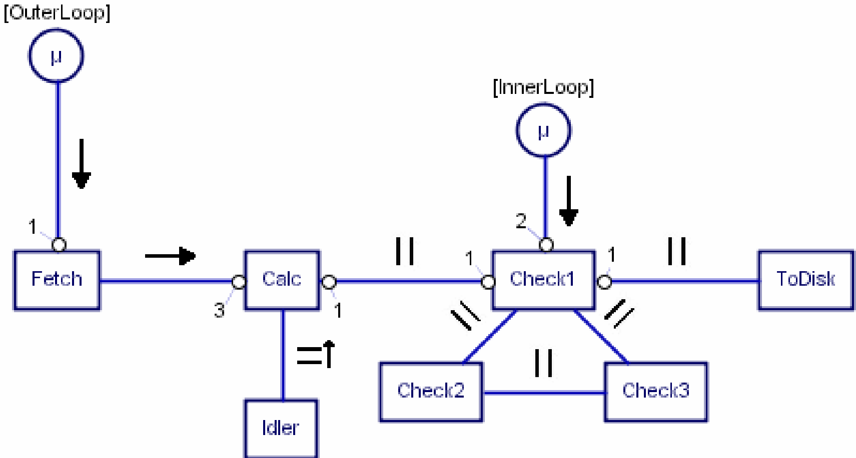


Figure 5-4. Compositional hierarchy shown by parenthesizing.

An index of a parenthesizing bubble is equal to the number of rectangles to be crossed going from a process to an external relationship in the boxed notation, as can be seen by comparing Figures 5-2, 5-3 and 5-4. It is quite obvious that parenthesis notation, although compact, demands practice to become readable.

5.2 C-tree as compositional hierarchy representation

It is not possible to generate code directly for the ambiguous model from Figure 5-1, but it becomes possible once compositional ambiguities are solved – for instance as depicted in Figure 5-4. In other words, one cannot generate an occam-like code in a unique way before a model is shaped as a strict hierarchy of constructs as branches and user defined processes as leaves.

A tool based on a strict tree hierarchy, as described in Subsection 2.1, is naturally suited for the generation of occam-like code. GML models, however, offer much more design freedom. The price to be paid is that a GML model which contains compositional ambiguities or conflicts cannot be uniquely displayed in a strict tree hierarchy. Therefore, a view based on a strict tree hierarchy and a view based on the GML model cannot be used together. One is forced to choose one or the other. In either case some valuable properties of the model are lost.

Possibly, in the design tool, an engine could be constructed that detects compositional underspecification by making queries to a model database. It would rely on some prescribed methods or perhaps built-in heuristics in resolving compositional ambiguities by deriving unspecified relationships, as suggested in the original GML proposal [16].

However, at this stage of the research these issues would put too much of a burden on the tool. Even the necessary minimal check whether a model is free of compositional conflicts would probably not scale well with the complexity of the design. Instead, a practical decision was made to share the responsibility of detecting compositional conflicts between the tool and the user. A supplementary view into the GML model is constructed that can give better insight in the hierarchy of models while keeping the design freedom of the GML modelling approach. The idea is to let the user build this alternative view gradually while creating a graphical design. A complex structure containing one or several tree views and two additional compartments can completely reflect the compositional side of the model at any moment of the design process. Two compartments contain flat lists of loose processes and unresolved relationships. The third compartment contains a set of tree hierarchies of which the roots and the branches represent constructs or complex processes while the leaves represent user-defined processes. This structure is the C-tree presented in

Figure 4-4 in Section 4. The C-tree view and G-editor view are always two views of the same GML model.

When someone adds a process or a relationship in the G-editor, they are automatically added in appropriate compartments of the C-tree. The ambiguous model from Figure 5-1 would be represented with all processes listed in the *Loose Processes* compartment, all relationships shown as unresolved, and with an empty third compartment. That model can be transformed to an unambiguous model from Figure 5-4 in several steps. One can start by grouping the *Check1, Check2* and *Check3* processes connected with parallel relationships into the parallel construct. The same can be done with the *Calc* and *Idler* processes.

When several processes connected via relationships of the same type are grouped to make one construct, this change will be reflected in the C-tree by the appearance of the construct as a root or a branch in the appropriate tree hierarchy. Processes that are grouped by a newly created construct have a parent process and cannot be classified as loose processes any more. Therefore they are moved from the first compartment to become leaves of a newly created branch representing their parent construct in the third compartment. Associated relationships contained in this construct are not unresolved further. For the given example this is illustrated in Figure 5-5. The next steps in resolving compositional ambiguities are shown in Figures 5-6 and 5-7.

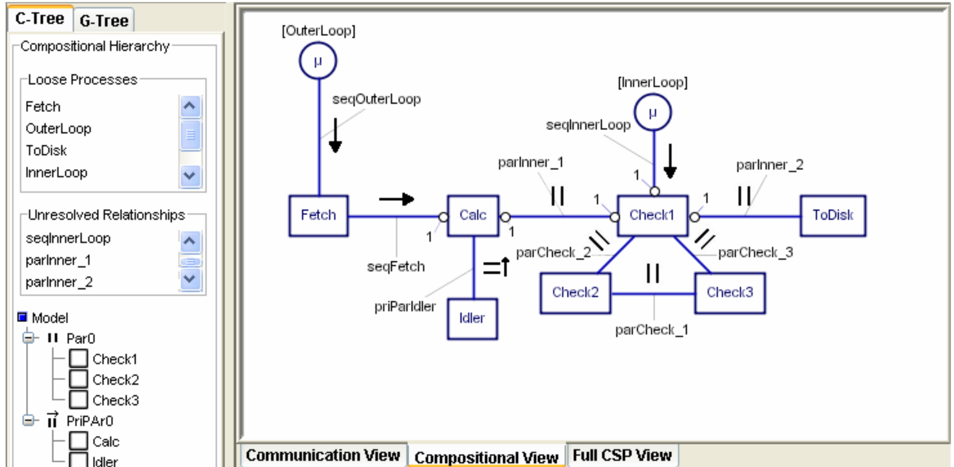


Figure 5-5. The model after creating two parallel constructs.

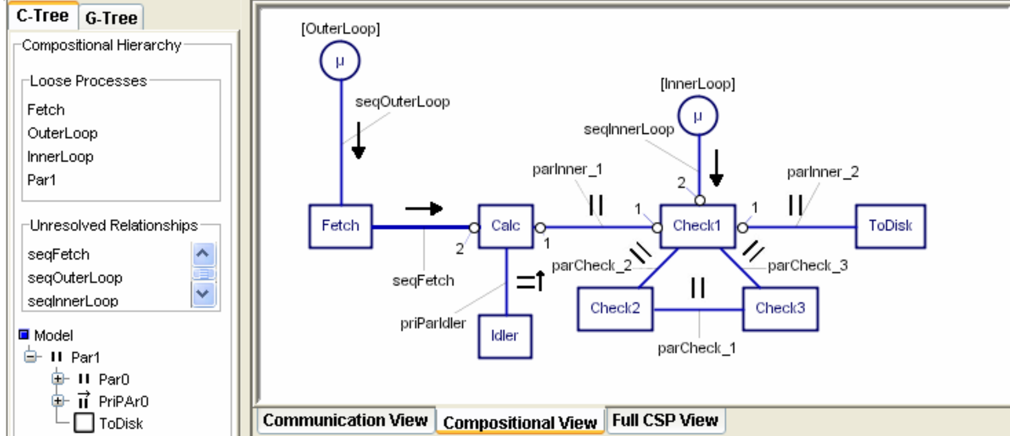


Figure 5-6. The model with *parInner* relationships resolved into *Par1* construct.

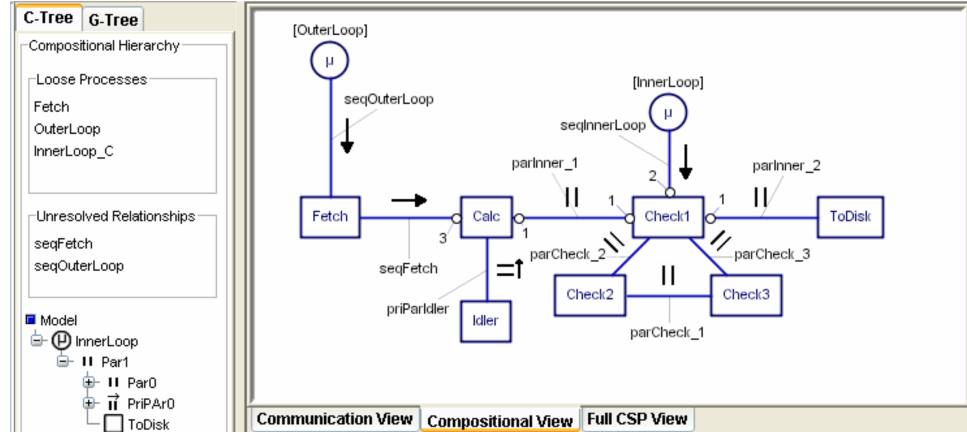


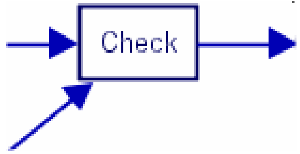
Figure 5-7. The model after creating the repetition construct.

The C-tree gives to a designer a better insight in the transient phases of hierarchy building. The designer can easily visually inspect a C-tree to determine how far the design is from a strict hierarchy. Code can be generated only after a strict tree hierarchy is obtained. More details about the role of the C-tree in code generation are given in Section 6.

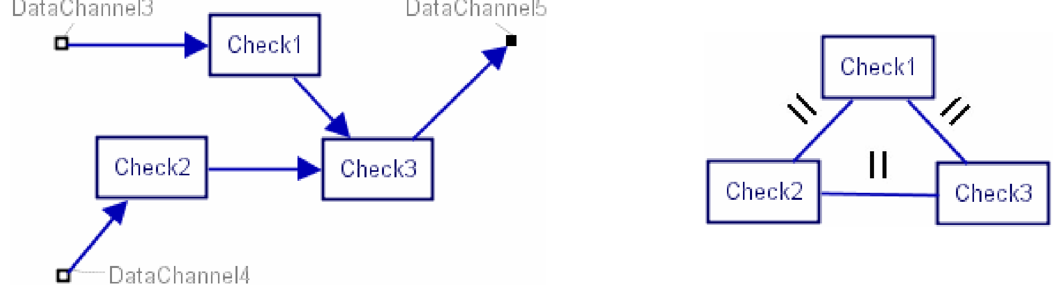
A *complex process* is a process that contains other processes. By definition, every construct is a type of complex process. During a design one can have complex processes whose internals are not yet shaped as a strict hierarchy of constructs and user defined processes. Contents of complex processes can be encapsulated using containment; this is the subject of the next subsection.

5.3 Structuring compositional hierarchy by the parent-child containment

So far, flat models have been considered. With a growing process network, a G-editor view quickly becomes overpopulated. In order to manage complexity of a model, in gCSP one can partition a model in larger logical chunks by building complex processes.



(a) Communication interface of the *Check* process.



(b) Communication view of *Check*'s internals. (c) *Check*'s internal composition.

Figure 5-8. Complex (parent) *Check* process.

At any moment the designer may decide to encapsulate several processes inside a separate process, which becomes the complex one. For example, the parallel composition of *Check1, Check2* and *Check3* (Figure 5-4) could be encapsulated inside the complex *Check* process (Figure 5-8a).

In turn, the body of the *Check3* process could be further refined as a sequence consisting of primitive input processes, output processes, and blocks for specification of the processing code (note that such blocks are not yet implemented in gCSP), as Figure 5-9 suggests.

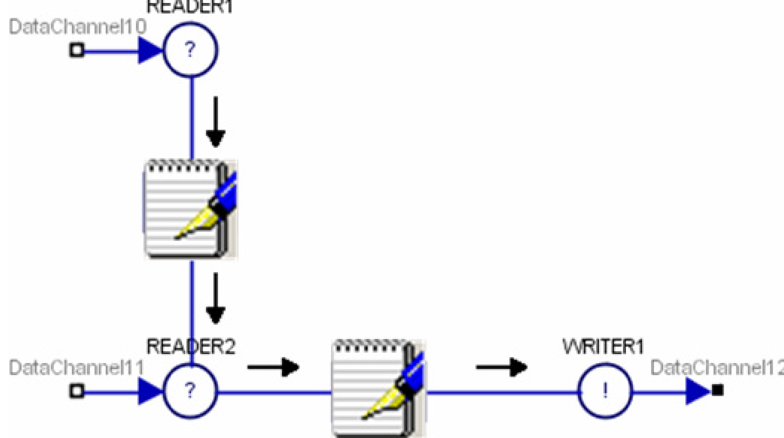


Figure 5-9. Possible representation for the body code place holders (for process *Check3*).

When building complex processes, channel interfaces to the next-higher level are indicated with symbols that are in line with the 20-SIM submodels notation: empty squares for input channel interfaces, and filled squares for output channel interfaces.

Note that one who prefers the top-down design approach would start with a smaller number of processes (as in the figures in Section 4, with the *Check* process capturing the functionality later distributed over three processes). When a global network of processes with well-defined interfaces is established, each of them can be further refined with subprocesses that fulfil the interface contract of the parent.

In the trees, the existence of hierarchical organization is indicated by means of the usual "+" and "–" (respectively expanding and collapsing) boxes in front of the complex processes and the constructs as well (Figure 5-10).

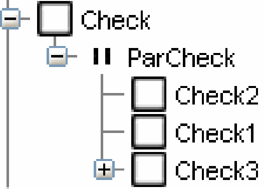


Figure 5-10. Containment hierarchy in the trees.

5.4 Containment hierarchy versus overview

Clearly, it is the user's decision to what extent to build complex compositions in a flat view and at which moment to "implode" some groups of processes into complex ones. Facilities for both building containment hierarchies based on complex processes made out of simpler ones *and* flat compositions with several constructs on the same visual level are implemented. Obviously, both are necessary to allow the user to find a proper balance between managing complexity of the design and providing sufficient overview.

Having both these facilities available, with a limited complexity of the subprocesses, the first disadvantage (inefficient space usage) of the boxed grouping notation discussed in 5.2 is no longer significant. The idea of adding the boxed notation along with the parenthesized one is not definitely abandoned and remains a subject for the future work.

While flat models at one visual level represent only the current abstraction level, the C-tree always represents the whole system. Of course, one can represent the whole system with one flat model as well, but this is not recommended for complex designs.

As is shown in Figure 5-10, the C-tree treats complex processes and constructs coherently. When a model is completely specified, every complex process contains exactly one (top) construct. The children of the top constructs are at the same time children of a complex process containing that construct.

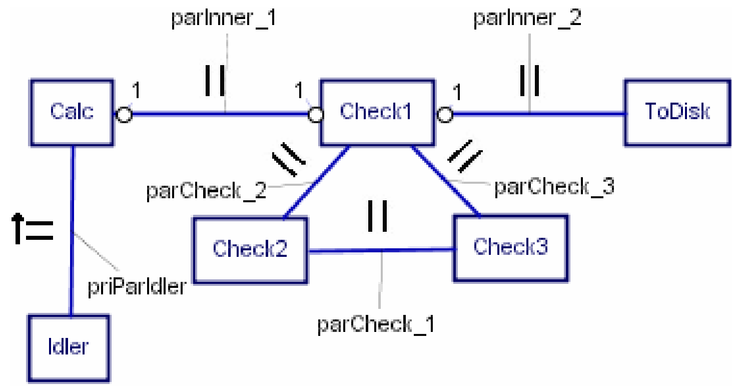
6. Code generation

As stated on the list of the tool development requirements, converting the graphical (human-readable) models into machine-readable forms by automatic code generation is an essential feature.

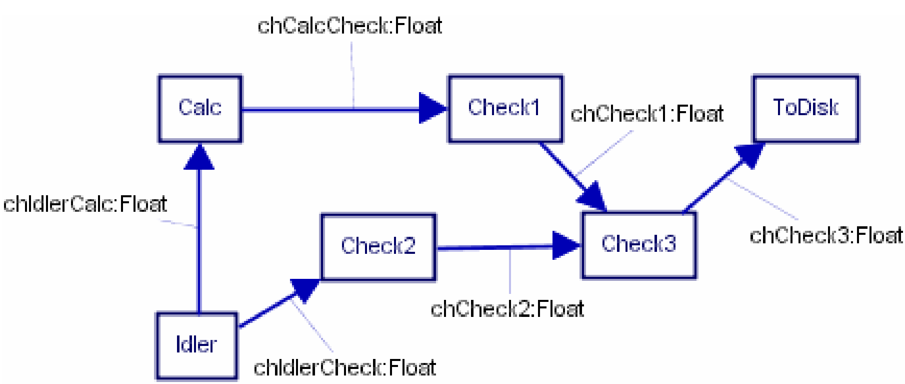
The following targets are of interest:

- CSP_M (machine-readable CSP), the input for the model checker FDR [24]. Analyses with FDR can be done directly to check the quality of a specification. Furthermore, using existing converters [25], the CSP_M code can be translated to CTJ, JCSP or CCSP.
- Executable programs based on use of the CSP libraries like CTJ [11], [10], [26] or JCSP [27], or their C or C++ versions. Graphical CSP models directly correspond to process networks that are built with the CSP libraries. As discussed later, after filling in application-specific code, the programs can be compiled and run directly.
- occam, which can be executed on transputers, or by compiling with KRoC [15] on standard processors.

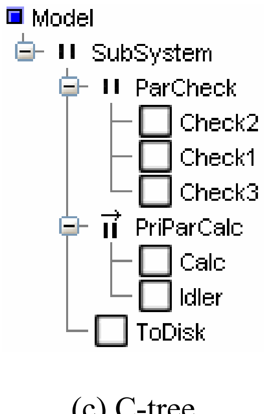
CSP is a notation and algebra that describes communication patterns in which different components interact. Parts of code that contain only pure computation are not of interest to CSP. Therefore this tool is also oriented towards generation of *network builders* - program files implementing concurrent networks that reflect the modelled concurrent structure, leaving empty place holders for the application-specific code. Filling this part of operational code can be done in different ways. One way is to do that also within the tool by using code blocks as sketched in Figure 5-9. The other option is importing code produced by other tools, for instance automatically coded control laws from 20-SIM, as described in [17] and [20].



(a) Compositional view.



(b) Communication view.



(c) C-tree

Figure 6-1. Example model for code generation.

For an example of code generation, a submodel of the previous example is shown in Figure 6-1. The C-tree structure gives a good starting point for any kind of code generation. Code can be generated whenever the state of strict hierarchy is reached. This state is reached when two conditions are met:

- there must not exist any unresolved relationship.
- only loose process that can exist is the top level construct.

Verifying whether code can be generated or not is therefore straightforward and also intuitively clear to the user (it is visual) - there is no need for a complex parsing algorithm. This does not yet mean that such a code is free of deadlocks, livelocks, etc. For this purpose a CSP_M script will be generated.

Listing 6-1. CSP_M code generated by gCSP.

```
channel  chCheck3
channel  chIdlerCheck
channel  chCheck2
channel  chCalcCheck
channel  chCheck1
channel  chIdlerCalc

SubSystem = ParCheck [| {| chCheck3 |} |] (PriParCalc [| {| chIdlerCheck,
chCalcCheck |} |] (ToDisk))
ParCheck = Check2 [| {| chCheck2 |} |] (Check1 [| {| chCheck1 |} |]
(Check3))
PriParCalc = Calc [| {| chIdlerCalc |} |] Idler

Calc = Calc
Idler = Idler
ToDisk = ToDisk
Check1 = chCalcCheck?x -> chCheck1!x -> Check1
Check2 = chIdlerCheck?x -> chCheck2!x -> Check2
Check3 = chCheck1?x -> chCheck2?x -> Check3
```

The CSP_M script for the model in Figure 6-1 is shown in Listing 6-1. Note that the channel communication for *Check1, Check2* and *Check3* is added manually after the network is generated by gCSP.

It has been decided to model the interleaving parallel construct (|||) in the graphical models in the same way as the sharing parallel construct. Firstly, the tool easily checks whether a channel is present between any two processes composed in parallel, and consequently decides to generate an interleaving or a shared parallel operator in the CSP_M code as appropriate. Secondly, insisting on appearance of the interleaving operator (|||) in the CSP_M code would complicate the code generation in some situations. For instance, a human would probably describe the *ParCheck* process as

```
(Check1 ||| Check2) [| {|chCheck1, chCheck2|} |] Check3
```

since *Check1* and *Check2* do not synchronise with each other. However, the tool, by parsing the C-tree, generates an equivalent composition that does not exhibit the interleaving explicitly.

An executable C++ code generated for use with the CTC++ library consists of pairs of .h and .cpp source files for each defined process and one .cpp file for the network builder as well. The structure of the network builder clearly corresponds to the compositional structure captured by the C-tree, as shown in Listing 6-2.

Listing 6-2. Network builder in CTC++.

```
/** Auto Generated - gCSP **/

//-- Includes
#include ...

int main (void) {

//-- Channel allocations
Channel<float> *chCheck3 = new Channel<float>();
Channel<float> *chIdlerCheck = new Channel<float>();
Channel<float> *chCheck2 = new Channel<float>();
Channel<float> *chCalcCheck = new Channel<float>();
Channel<float> *chCheck1 = new Channel<float>();
Channel<float> *chIdlerCalc = new Channel<float>();

//-- Process allocations
Calc *Calc_1 = new Calc(chCalcCheck, chIdlerCalc);
Idler *Idler_1 = new Idler(chIdlerCheck, chIdlerCalc);
ToDisk *ToDisk_1 = new ToDisk(chCheck3);
Check1 *Check1_1 = new Check1(chCalcCheck, chCheck1);
Check2 *Check2_1 = new Check2(chIdlerCheck, chCheck2);
Check3 *Check3_1 = new Check3(chCheck3, chCheck2, chCheck1);

//-- Network builder
Parallel *ParCheck = new Parallel(
       Check2_1,
       Check1_1,
       Check3_1,
NULL);

PriParallel *PriParCalc =  new PriParallel(
       Calc_1,
       Idler_1,
NULL);

Parallel *SubSystem = new Parallel(
       ParCheck,
       PriParCalc,
       ToDisk_1,
NULL);

SubSystem->run();

//delete's...

return 0;
}
```

7. Conclusions and future work

A basic version of a graphical CSP editor and code generator has been built. The first tests, as shown in this paper, indicate that the tool has the potential to meet its initial requirements, and can help proliferate the ideas and usage of CSP.

The idea of the C-tree view can also be generalized and applied to similar problems where an additional view is needed to visualize the process of forming the hierarchy rather than the hierarchy itself.

Building a usable graphical tool is a laborious process; many interesting features can be thought of. The following features are recognized as vital for acceptance in a somewhat broader audience that could supply the tool and methodology developers with valuable feedback:

1. Further improvement of managing complex models; imploding a group of processes to a complex one; exploding complex processes by bringing their children higher up in the hierarchy with proper handling of the network topology.
2. Reassessing the benefits and drawbacks of the boxed and bubbled grouping notations.
3. Extending reusability of developed CSP models. At this moment reuse is possible only by saving and retrieving submodels (complex processes).
4. Letting the user enter the body (operational code) of the processes via code blocks or import external program files.
5. Allowing for diagrams partly populated – as desired by the user – with the processes and relationships from the model.
6. Enabling layered organization of complex models; for instance, a user may choose different ensembles of coexisting networks reflecting a control application layer, safety components layer, and hardware deployment layer to be displayed in various combinations. The preserving topology of the processes in different views with the layered compositions comes to its full effect.
7. Facilitating the tool with event trace analyses by mimicking the network interaction with the environment, as in ProBE [24]. This could lead to executable specification of CSP concurrent designs.
8. Bidirectional collaboration with model checkers and visualization of concurrency phenomena.
9. Allowing use of the C-tree for entering design as well. The manipulation of large structures (merging constructs or dragging and dropping processes) would be easily done in the tree.

8. Acknowledgements

The contribution of Gerald Hilderink to this project is substantial. Beside the fact that the tool is based on the GML modelling principles that he has established, Gerald has also supervised the development of a significant part of the GUI for gCSP during the internship project of Marvin Rumnit.

The authors are also grateful to the reviewers who carefully pointed out potentially weak points of the paper. Furthermore, we are grateful to Dyke Stiles for special diligence in correcting the English.

References

[1] INMOS, "INMOS Website" www.inmos.com, 2004.

[2] H. J. Volkerink, G. H. Hilderink, J. F. Broenink, W. A. Vervoort, and A. W. P. Bakkers, "CSP Design Model and Tool Support", *Communicating Process Architectures 2000, WoTUG-23*. Canterbury, United Kingdom 2000.

[3] INMOS, *occam 2 Reference Manual*: Prentice Hall, 1988.

[4] A. Burns and A. Wellings, *Real-TIme Systems and Programming Languages*, 3rd ed: Pearson Education, 2001.

[5] A. W. Roscoe, *The Theory and Practice of Concurrency*: Prentice Hall, 1997.

[6] S. Triger, B. C. O'Neill, and J. Clark, "Adapted OS Link / DS Link Protocols for Use in Multiprocessor Routing Networks", In A. Chalmers, M. Mirmehdi, and H. Muller, Eds., *Communicating Process Architectures 2001*. Bristol, UK, 2001.

[7] R. Peel, "A Reconfigurable Host Interconnection Scheme for occam-Based Field Programmable Gate Arrays", In A. Chalmers, M. Mirmehdi, and H. Muller, Eds., *Communicating Process Architectures 2001*. Bristol, UK, 2001.

[8] R. Mosely, "Reconnectics: A system for the Dynamic Implementation of Mobile Hardware Processes in FPGAs", In J. Pascoe, P. H. Welch, R. Loader, and V. Sunderam, Eds., *Communication Process Architectures 2002*. Reading, UK, 2002.

[9] G. H. Hilderink, "JavaPP project at UT: http://www.ce.utwente.nl/JavaPP", 2002.

[10] G. H. Hilderink, A. W. P. Bakkers, and J. F. Broenink, "A Distributed Real-Time Java System Based on CSP", *The third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2000*. Newport Beach, CA, 2000.

[11] G. H. Hilderink, J. F. Broenink, W. A. Vervoort, and A. W. P. Bakkers, "Communicating Java Threads", *Proc. WoTUG-20 on Parallel programming and Java*. Enschede, Netherlands, 1997.

[12] P. H. Welch, "Process Oriented Design for Java: Concurrency for All", *ICCS 2002*. Amsterdam, 2002.

[13] J. Moores, "CCSP - A portable CSP-based run-time system supporting C and occam", In B. M. Cook, Ed., *Architectures, Languages and Techniques - WoTUG-22*. Keele, UK, 1999.

[14] N. C. C. Brown and P. H. Welch, "An Introduction to the Kent C++CSP Library", In J. F. Broenink and G. H. Hilderink, Eds., *Communicating Process Architectures 2003*. Enschede, 2003.

[15] P. H. Welch and D. C. Wood, "The Kent Retargetable occam Compiler", *Parallel Processing Developments -- Proceedings of WoTUG 19*. Nottingham, UK, 1996.

[16] G. H. Hilderink, "Graphical modelling language for specifying concurrency based on CSP," *IEE Proceedings: Software*, vol. 150, pp. 108-120, 2003.

[17] D. Jovanovic, G. H. Hilderink, and J. F. Broenink, "A communicating Threads -CT- case study: JIWY", In J. Pascoe, P. H. Welch, R. Loader, and V. Sunderam, Eds., *Communicating Process Architectures 2002*. Reading UK, 2002.

[18] Mathworks, "Matlab, Simulink" http://www.mathworks.com: Mathworks, 2002.

[19] CLP, "20-SIM" http://www.20sim.com: Controllab Products, 2002.

[20] G. H. Hilderink, D. S. Jovanovic, and J. F. Broenink, "A multimodal robotic control law modelled and implemented with the CSP/CT framework", In J. F. Broenink and G. H. Hilderink, Eds., *Communicating Process Architectures 2003*. Enschede, Netherlands, 2003.

[21] G. H. Hilderink, "A graphical Specification Language for Modeling Concurrency based on CSP", In P. W. James Pascoe, Roger Loader, Vaidy Sunderam, Ed., *Communicating Process Architectures 2002*. Reading UK, 2002.

[22] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language, User Guide*: Addison Wesley, 1999.

[23] A. L. Lawrence, "CSPP and Event Priority", In A. Chalmers, M. Mirmehdi, and H. Muller, Eds., *Communicating Process Architectures 2001*. Bristol, UK, 2001.

[24] FormalSystems, "FDR2 Refinement checker for CSP models" http://www.fsel.com, 2004.

[25] V. Raju, L. Rong, and G. S. Stiles, "Automatic Conversion of CSP to CTJ, JCSP, and CCSP", In J. F. Broenink and G. H. Hilderink, Eds., *Communicating Process Architectures 2003*. Enschede, Netherlands, 2003.

[26] G. H. Hilderink, "Communicating Threads home page: www.ce.utwente.nl/JavaPP,", 2002.

[27] P. H. Welch, "Java Threads in the Light of occam / CSP", In P. H. Welch and A. W. P. Bakkers, Eds., *Architectures, Languages and Patterns for Parallel and Distributed Applications, WoTUG-21*. Canterbury, UK, 1998.