Deadlock Detection Based on Automatic Code Generation from Graphical CSP Models

Dusko Jovanovic, Geert K. Liet and Jan F. Broenink
Twente Embedded Systems Initiative,
Drebbel Institute for Mechatronics and Control Engineering,
Dept. of Electrical Engineering, University of Twente,
P.O.Box 217, NL-7500 AE Enschede, The Netherlands
Phone: +31 53 489 2788 Fax: +31 53 489 2223
E-mail: D.Jovanovic@utwente.nl

***Abstract*— The paper describes a way of using standard formal analysis tools for checking deadlock freedom in graphical models for CSP descriptions of concurrent systems. The models capture specification of a possible concurrent implementation of a system to be realized. Building the graphical models and transforming them to a textual machine-readable form is supported by a CASE tool under development called gCSP. The model transformation allows checking certain important behavioral properties of a candidate implementation before it gets refined with application specific details and deployed in exploitation conditions. A short introduction to the modeling forms and tools is given before a demonstration of the checking procedure on two examples of (embedded) control systems is presented. These systems are modeled by a special class of CSP processes, for which a graphical mechanism for revealing and healing ill-posed concurrent compositions is prototyped too.**

***Keywords*— Embedded Control Systems; deadlock, Formal checking; CSP**

I. Introduction

“One of the main motivations for introducing *concurrent processes* into a language is that they enable parallelism in the real world to be reflected in application programs. This enables such programs to be expressed in a more natural way and leads to the production of more reliable and maintainable systems”. [1, p318] Besides the most important reason given above for involving a concurrent paradigm in the development of software for embedded systems, many other advantageous qualities apply too: components reusability, design distributiveness, better performance and responsiveness (real-time behavior) [2, 3]. Nevertheless, the concurrent programming is not widely accepted in the industrial practice. It is infamous as hard to understand, test and debug. This is in contradiction with the citation given above.

The reason for this notorious reputation is bad experience with using bare multithreading as a way for programming concurrency. The most popular languages for programming embedded systems do not include explicit notion of processes: Java provides some language (compiler) support for multithreading, while C and C++ are purely sequential – any concurrent behavior is programmed in special components (often libraries) which are usually operating-system dependent.

Using the languages with explicit notions of processes (like tasks in Ada or processes in Modula) makes programming concurrency in a process-oriented way just better structured – the most troublesome problems with concurrent implementations are not yet solved on the level of any programming language. These are problems like *deadlock* or *livelock*, behavioral phenomena in systems composed of parallel processes whose executions are dependent on each other only due to exchange of data – *communication*. This means that they can execute in any order until some of them must input data produced by other processes. At the points of data delivery by producer processes and data reception by consumer processes, *synchronization* in their execution must take place. A consumer must suspend its execution until the data it needs are produced. In *rendezvous* (synchronous) concurrency models, a producer must also wait for a consumer to reach the point in its execution at which it can take the data prepared by the producer. An asynchronous model allows the producer to continue its execution after leaving data for a consumer to pick it up later.

This means that synchronization by interprocess communication serializes execution of processes running in parallel. Regardless how complex a parallel system can be, each is composed of sequential processes at the lowest level of granularity. These processes implement processing algorithms whose causality determines sequences of data inputting, calculations and data outputting. A sequence of inputs and outputs constitutes a *communication pattern* of a process. In certain

[*]) 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 an the Technology Foundation STW.

constellations of sequential processes running in parallel, it may happen that two or more processes wait on each other for data necessary to continue with further executions. This kind of condition is called deadlock.

In the central section (III) of this paper deadlock occurrences and their detection are presented in two examples taken from the realization of control systems. The procedures that help eliminating deadlock conditions are illustrated as well. Before that, a design environment that allows appropriate reasoning about the concurrency in systems is introduced (section II). At the end, section IV summarizes existing and prospective benefits of the described procedures and tools.

II. CSP MODEL OF CONCURRENCY AND THE DESIGN ENVIRONMENT

CSP (Communicating Sequential Processes, [4-6]) is a process algebra which is defined as "a notation for describing *concurrent systems* (i.e., ones where there is more than one process existing at a time) whose component *processes* interact with each other by *communication*" [6, p. 1]. The key words (emphasized by the authors of this paper) in the two citations mentioned in this paper do not coincide accidentally. Hoare's seminal paper on CSP [4] is inspired by the problems of concurrent programming at that time (mostly the same as nowadays). Ergo, CSP is invented to describe concurrent systems, but with a mathematical rigor. This implied the feasibility of *formal verification methods* for a concurrent program correctness, that are developed in further research [6-8].

CSP offers to the process-orientation in designing concurrent systems a fundamental and natural architectural vocabulary: processes as building blocks, (waiting) rendezvous channels for interprocess communications and three basic operators for process compositions (sequential, parallel and alternative).

The two famous (now also classical) practical implementations of CSP principles for software development are the occam programming language (for transputer platforms) and Ada. Particularly occam is interesting as being completely a process-oriented language in CSP fashion. By being used successfully for implementation of complex systems [9-11], occam proved the simplicity of building large parallel systems on CSP principles. The original set of CSP operators, in occam implemented as constructs `SEQ`, `PAR` and `ALT`, for real-time and performance reasons is augmented by prioritized variants `PRI PAR` and `PRI ALT`.

In this text, formal analysis will be demonstrated on a class of parallel systems described in the original CSP concept [4]. The name CSP stands for *sequential processes* that *communicate* over rendezvous channels. The processes considered in this text are *sequential* in the sense that they are composed of actions taking place in a prescribed, consecutive order – no parallelism within the processes is allowed. However, the processes are arranged in parallel compositions. Actually, the deadlock analysis makes sense only if the processes are parallel composed. Sequentially composed processes connected by rendezvous channels always deadlock, as well as alternatively composed processes. Explanation is quite simple: if a process (for example, a data producer) needs to synchronize with another process (consumer), the execution of the consumer must coexist with the execution of the producer. If the producer and the consumer are sequentially composed, that means that the consumer executes only upon the producer has terminated – thus rendezvous synchronization is not possible. In the case of alternatively composed processes the situation is similar: since they have alternative executions, it cannot happen that the executions coexist, thus synchronization is again not possible.

The revised theory of CSP [5, 6] allows parallel compositions be encapsulated within processes, making these process parallel inside. For simplicity of the deadlock interpretation, simpler version of sequential processes will be used in the two examples in section III. The impact of allowing internal parallelism will be discussed in section IV.

A. CSPm and formal checking tools for CSP

The CSP language is a mathematical, algebraic notation, and as such can be manipulated only by humans. While it is usable for manual modeling systems with a few tens of processes, formal checking of the behavioral properties of the models even of such modest size is beyond the capabilities of the human mind. The computer tools support for dealing with models larger than just toy-examples is necessary. The gCSP tool for managing graphical CSP models of arbitrary size is presented later in this section.

The two most known tools for analyzing CSP models, FDR and ProBE, are products of Formal Systems Ltd.[12]. FDR (Failures-Divergence Refinement) is a model-checking tool for state machines [13]. It is built on operational semantics (state machine representation) of CSP models. ProBE is a much smaller tool that can animate a sequence of events in a CSP model.

The CSP models must be represented in a machine-readable form in order to be processed by the tools. Machine-readable CSP, CSPm [8], is a subset of CSP that can be textually (ASCII) coded in scripts loadable by FDR and ProBE. A small example of a producer-consumer model in CSPm would look like this:

```
datatype theType = someValue | anotherValue
channel ch1 : theType
channel ch2 : theType

Producer12 = ch1!someValue -> ch2!anotherValue
               -> Producer12
Consumer12 = ch1?aVariable -> ch2?bVariable
               -> Consumer12
Consumer21 = ch2?bVariable -> ch1?aVariable
               -> Consumer21
```

```
SystemDF = Producer12 [|{| ch1, ch2 |}|]
                Consumer12
SystemDC = Producer12 [|{| ch1, ch2 |}|]
                Consumer21
```

The interpretation of the first three lines is quite obvious: a datatype `theType` consist of values (constants) `someValue` and `anotherValue`, and the channels `ch1` and `ch2` can carry that type.

A process communication patterns in CSP and CSPm are basically specified by constructs for writing (“`!`”) a value to a channel and reading (“`?`”) a value from a channel to a variable (that need not be declared). Thus, `Producer12` attempts writing the constant `someValue` to `ch1`, `anotherValue` to `ch2` and then repeats, while `Consumer12` attempts to read a value first from `ch1` and store it in the variable `aVariable`, then attempts to read from `ch2` and store and store in `bVariable`; then repeats itself. `Consumer21` does the same, but the reading is in reverse order. Writing and reading to/from channels represent communication *events*. The sequence of events, always ending with quoting a process name, is composed by prefixing operators “`->`”, that can chain an event to another event or an event to a process (whose behavior succeeds that event – in CSP terminology: the event *guards* the process). The fact that a process repeats infinitely is expressed by putting the name of the process at the end of its own sequence (thus becoming a recursive process). CSPm semantics defines a few special processes with predefined behavior that also may close the sequence, for instance `SKIP`, which means an unconditional termination and will be used in the first example (see section III.A). Prefixing cannot be used for composing processes in sequence (for that, the sequential operator “`;`” is used).

The two last lines in the script specify two different synchronized parallel compositions (that are also legitimate processes). Each of them in fact represents a parallel model whose analysis is desired. Such a right-hand side expression as a top level composition (consisting of other processes and/or simpler compositions) is called a *network builder*. The expressions of `SystemDF` and `SystemDC` are both network builders describing two different models (that share the same `Producer12` specification).

Parallel process `SystemDF` is build by the first network builder out of two processes, `Producer12` and `Consumer12`. “`[|{| ch1, ch2 |}|]`” is one of the CSPm parallel operators, so-called shared parallel. It means that `Producer12` and `Consumer12` must synchronize on each event on the channels `ch1` and `ch2`. The same applies for `SystemDC` that is built out of `Producer12` and `Consumer21`. Another parallel construct that requires no synchronization (“interleaving”) is introduced in the first example (see section III.A). For a complete list of operators consult the FDR manual [13, p. 60].

As specified in this introductory script, `SystemDF` is deadlock-free (“DF”) while in `SystemDC` a deadlock condition (“DC”) occurs. The reason is simple: communication patterns of `Producer12` and `Consumer12` are compatible, while those of `Producer12` and `Consumer21` are not.

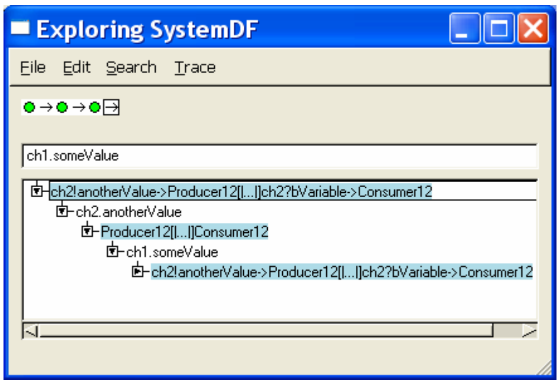
These conditions can be captured by both ProBE and FDR, but quite differently. ProBE is an interpreter of CSPm scripts, and thus requires progressing through the event sequences until processes are terminated (what would be never a case with recursive processes) or no events are offered before termination of all processes (that means a deadlock). Obviously, using ProBE does not make sense for an exhaustive checking on deadlock conditions. It is rather used to understand the sequence that leads to a deadlock occurrence already indicated by FDR.

Two screens opened by ProBE for exploring the two network builders are presented in Figure 1 and Figure 3. It should be noticed from Figure 1 that ProBE, while mimicking the synchronizations among processes, a pair of communication attempts in processes (like `ch2!anotherValue` from `Producer12` and `ch2?bVariable` from `Consumer12`) represents as one communication event indicating the written value (`ch2.anotherValue`), what is semantically correct.

In this particular case, SystemDC deadlocks immediately, which is indicated by an empty event pane in Figure 2 An experienced ProBE user can spot in the event pane in Figure 1 a repeating pattern at the top and at the bottom – this may lead to a conclusion that SystemDF infinitely repeats without deadlock.

The power of detecting or “proving” deadlock freedom by ProBE is limited to the problem size like in the presented script. For an exhaustive analysis of more interesting systems the FDR is used. For the previous script, the FDR window is shown in Figure 3.

At the bottom all processes defined in the script are listed. From that list one may chose processes interesting for checking against deadlock. In the upper list it is shown that FDR finds all sequential process (first three



ch1.someValue

```
ch2!anotherValue->Producer12[|...|]ch2?bVariable->Consumer12
  ch2.anotherValue
    Producer12[|...|]Consumer12
      ch1.someValue
        ch2!anotherValue->Producer12[|...|]ch2?bVariable->Consumer12
```

Figure 1. Event sequence in ProBE for the `SystemDF` process.

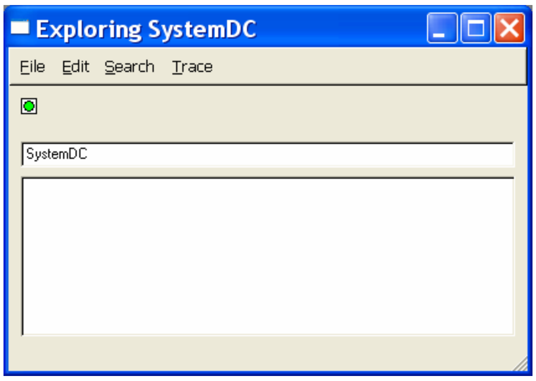


Figure 2. Event sequence in ProBE for the `SystemDC` process

in the list) deadlock free – in front of them a tick sign is displayed. It is also shown how a deadlock condition is indicated (a cross sign in front of `SystemDC`). Besides the indication, FDR may report the event trace that leads to a deadlocked situation, as well as a certain interactive debug engine for understanding the erroneous situation. Using these facilities for debugging a specification requires practice and good understanding of the CSPm modeling techniques.

As the tabs at the top of the window suggest, FDR is also capable of analyzing CSPm models for the properties other than deadlock, such as livelock, determinism and checking refinement relations between different processes [13].

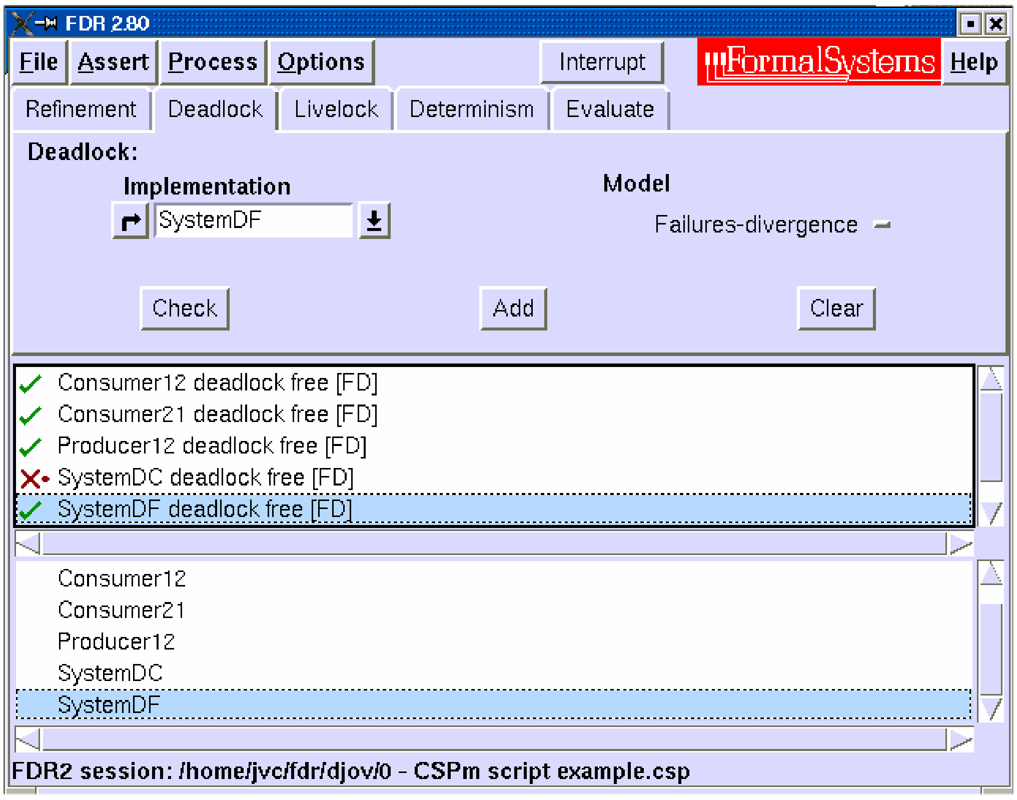
B. CT libraries

After transputers vanished from the market as deployable processing units (in mid 1990's), the attempts to preserve the CSP design methodology for building concurrent software have been initiated. They went in a few directions. The most known are:

- The language Handel-C [14] used for programmable hardware circuits.
- A cross-compiler (also supporting additional language features) for occam KRoC [15].
- occam-like libraries for the mainstream languages for embedded systems, C/C++ and Java [16-20].

CT libraries, CTJ [16] for Java, CTC and CTC++ [17] for C/C++ have been initially developed by Hilderink at the Control Laboratory [21]. Robotic applications of CTC++ library are reported in [22, 23]. A significant advantage of supporting CSP-based concurrent programming in the form of interface-compatible libraries for different programming languages is the possibility to implement the same (CSP) abstract concurrency model in various development environments. In the first place, this dispels the criticism against concurrency at the level of programming languages because they are not uniform [1, p. 182]. Secondly, the proven concepts become usable to larger programmer communities already having highly efficient code for specific applications.

Furthermore, the CT libraries, with an integrated real-time kernel independent of OS scheduling mechanisms, facilitate easier portability of a concurrent design.


FDR 2.80
File Assert Process Options Interrupt FormalSystems Help
Refinement Deadlock Livelock Determinism Evaluate
Check Add Clear

Deadlock:

Implementation: SystemDF

Model: Failures-divergence

✓ Consumer12 deadlock free [FD]
✓ Consumer21 deadlock free [FD]
✓ Producer12 deadlock free [FD]
✗ SystemDC deadlock free [FD]
✓ SystemDF deadlock free [FD]

Consumer12
Consumer21
Producer12
SystemDC
SystemDF

FDR2 session: /home/jvc/fdr/djov/0 - CSPm script example.csp

Figure 3. FDR window

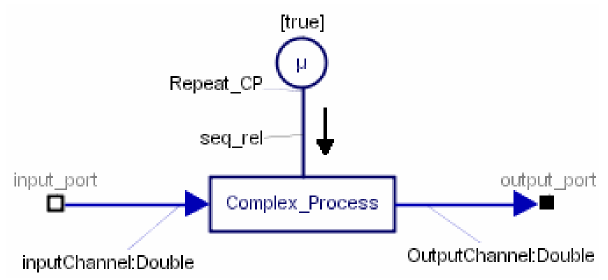


Figure 4. A non-terminating process (a full CSP view)

Having process execution management as a part of the application, the real-time behavior is more likely preserved when ported to another platform. Moreover, being independent of an OS, the CT-based embedded designs are portable with minimal adjustments to non-OS ("bare metal") embedded processors.

C. CSP diagrams: Graphical Modeling Language and gCSP tool

Along with his research on the CT libraries, Hilderink developed a graphical notation to support modeling systems to be implemented with the libraries. This graphical notation, called Graphical Modeling Language (GML, [24]) was the starting point in the development of a CASE tool for CT-based systems, called gCSP. All CSP diagrams (GML models) in the paper are created in the graphical editor of gCSP. The tool features are in detail described in [25]. In short, the tool allows modeling and managing arbitrarily large process-oriented systems by providing standard as well as specific means for partitioning systems in hierarchical organizations.

Besides modeling CSP/CT designs in GML, the equally important feature of the tool is automatic code generation. The GML specification of a design can be transformed in textual CSPm form – a CSPm script. Upon deadlock analysis of the design, its CTC++ implementation can be generated. The support for other CT libraries is postponed to a later tool development stage.

In the following four figures a basic graphical vocabulary of GML used for examples in this text is presented.

In Figure 4, a process is presented depicted with a rectangle and the name (Complex_Process). The process is connected for communication with the rest of the world by channels, inputChannel and outputChannel that carry values of type Double. Channels are in GML terminology *communication relationships* between processes. In this case, Complex_Process is connected with two other processes that are not presented at the same hierarchical level. From this diagram it may be assumed that Complex_Process is encapsulated within another process whose (communication) interface is presented by one input port (small empty square) and one output port (small filled square) – input_port and output_port respectively.

A third relationship in Figure 4 (seq_rel) does not describe communication between processes, but a pattern of execution. Therefore, it belongs to another group of relationships in GML, called *compositional relationships*. In Table 1 the set of GML compositional relationships that coincide to occam constructs (and therefore also existing in CT) is listed.

Table 1. Compositional relationships

Relationships symbols	**Compositional relationships**
$\underline{\rightarrow}$	sequential
$\underline{\|}$	parallel
$\overset{\rightarrow}{\underline{\|}}$	priparallel
$\underline{\square}$	alternative
$\overset{\rightarrow}{\underline{\square}}$	prialternative

As both communication (channels) and compositional relationships are represented by lines, a convention is used to mark channels with an arrow at one end. The arrow marks a role in a communication relationship (sender-receiver, producer-consumer or server-client) and represents the direction of data transfer along the channels.

The sequential relationship `seq_rel` determines that `Complex_Process` executes upon termination of the `Repeat_CP` process, represented by a circle in Figure 4. The `Repeat_CT` process is one of the special ("primitive") processes defined in GML, and that is the reason it is not depicted by a rectangle (as user-defined processes) but as a circle. Primitive GML processes that appear in this text are listed in Table 2.

Table 2. Primitive processes

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

Specialty of a μ process (repetition) composed by a sequential relationship with another process is that it indicates repetition of that another process as long as the condition (stated in rectangular brackets next to the μ process) is fulfilled. Value `true` of the condition means endless (infinite) repetition of a process. Implemented in a programming language, this implies an infinite loop. Although not very common in many application programs, this is not strange for some components of embedded systems, which are supposed to operate as long as a device is in service. Moreover, for deadlock checking with FDR it is necessary to have constituent processes non-terminating.

Thus, for the example in Figure 1, the sequential composition of `Repeat_CP` and `Complex_Process` would literally means that `Repeat_CP` executes first and after it terminates, `Complex_Process` executes. Upon termination of `Complex_Process`, sequence would normally terminate as well, but due to the special nature of the `Repeat_CP` process, it is being executed again, and so on. In practice (code generation), a μ process with a `true` condition just denotes that the accompanied processes (`Complex_Process` in this case) should be made recursive (like those sequential processes in the CSPm script), i.e. surrounded by infinite loops (for CT code generation).

In Figure 4 a *full CSP view* is displayed, that means with both communication and composition relationships. gCSP tool allows also showing only one kind of relationships, that introduces *communication* and *compositional* views to CSP diagrams, i.e. GML models. In the following three figures internals of the `Complex_Process` process are rendered in all three views.

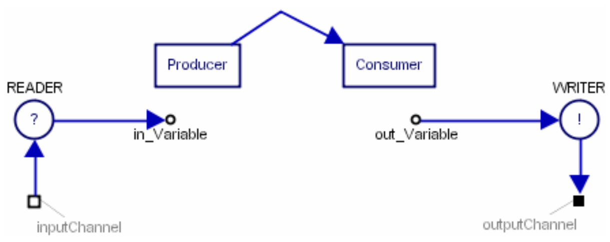


Figure 5: Communication view to `Complex_Process`'s internals

Figure 5 introduces a use of primitive reader and writer processes for storing/writing values from/to channels (or ports) to/from variables local to processes. The symbols for these primitive processes coincide with already introduced CSPm operators "`?`" and "`!`" for reading and writing from/to a channel. Variables are represented by small circles and names (`in_Variable`, `out_Variable`). The mentioned convention of orienting channels is here obvious: `Producer` is supposed to supply `Consumer` with data. In this example it may be assumed that `Producer` encapsulates some processing upon `in_Variable`, forwards an interim result to `Consumer` that finishes the computation off and stores the final result in `out_Variable`.

This introductory example just illustrates the graphical vocabulary used in the remainder. Processing variables is rather performed by code blocks, more primitive processing entities than processes, represented by rounded rectangles (see examples in section III). The code blocks are defined in the scope of a surrounding process, thus all variables in the scope of the process are directly visible to the code within the code blocks, what is not the case for nested processes (this is due to encapsulation principles of object-oriented implementation of the CT libraries).

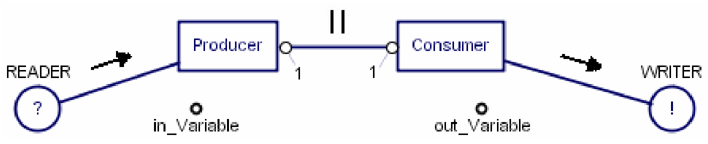


Figure 6: Composition view to `Complex_Process` 's internals

In Figure 6 a possible composition of child processes of `Complex_Process` is specified: `Producer` runs in sequence after `READER`, while `Consumer` precedes `WRITER`. Then these two groups are composed in parallel. The pair of bubbles at the ends of the parallel relationship denote that the parallel composition applies to the groups connected with the relationships without bubbles (in this case the two sequential relationships). Otherwise, it would be ambiguous what the compositional structure really is, since the construct precedence in graphical models is not defined. In the later examples bubbles will be used only to denote repetitions of sequences of primitive communication processes and code blocks.

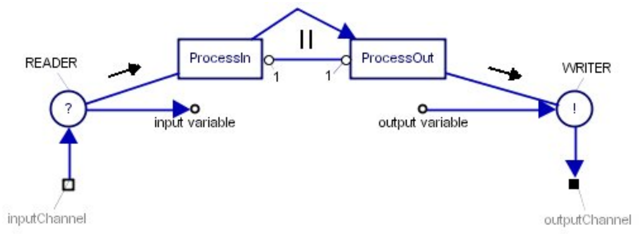
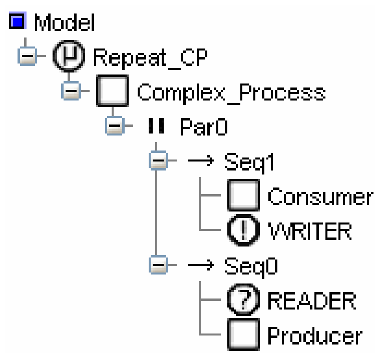


Figure 7. Full CSP view to `Complex_Process` 's internals

Figure 7 demonstrates a complete design overview that is obtained by combining the communication and composition views in the full CSP view. Although providing all graphical information one may want to inspect at a certain design phase, some models can become unreadable, especially if names of all relationships are displayed. For the focal issues of this paper, a simplification is introduced along the examples elaboration.

One more crucial view is provided by the compositional tree, or *C-tree*. The C-tree provides the information on compositional structure expressed in compositional view of the graphical editor and moreover the parent-child containment relationships between complex processes (composed of other processes) and their child processes. Hence the C-tree provides an overview of the overall model and makes navigation



Model
Repeat_CP
Complex_Process
Par0
Seq1
Consumer
WRITER
Seq0
READER
Producer

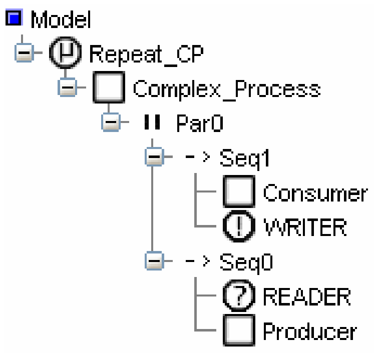
Figure 8: The C-tree

through the model easier. For the running GML example, the C-tree is shown in Figure 8.

The tree structure naturally suits presenting hierarchical structures, what a compositional structure written in CSP/CT essentially is. The grouping of the two sequential compositions (`Seq0` and `Seq1`) further composed in a higher `Par0` parallel construct is also much more obvious. Note that in the C-tree constructs are presented, not the compositional relationships (although having the same symbols, see Table 1). Actually, a construct in the C-tree appears when two of more processes connected by the same compositional relationships are grouped together ("composed in a construct").

The code generation engines of gCSP are based on the information contained in the C-tree. As can be seen from Figure 8, sequential construction with a repetition (μ) is optimized to be represented just by "μ" as a construct. This implies an optimization to the generated CT code: no additional constructs for each non-terminating process reduces the execution overhead and saves memory.

One CSP operator, not defined in GML, can be used in C-tree for further optimizing code generation; both for CT and CSPm. Namely, GML relationships can connect only processes, regardless primitive or user defined. This would mean that in a CSPm script, generated from a GML model with primitive readers and writers, processes that read and write would have to be composed of reading and writing *processes*, that in turn would contain only one "`channel!variable`" (or respectively "`channel?variable`") statements each. This would increase size of a CSPm script, make it less readable and more importantly: would be very uncommon for experienced CSPm modelers. Since the reading/writing CSPm statements are always serialized by the prefixing operator "`->`" (that belong to a group of sequential CSP operators [13, p. 60]), in the C-tree the sequential operators can be switched to the prefixing operators, compare `Seq0` and `Seq1` in Figure 8 and Figure 9.
After this, CSPm code is produced in a way that all reader and writer processes are generated as



Model
Repeat_CP
Complex_Process
Par0
-> Seq1
Consumer
WRITER
-> Seq0
READER
Producer

Figure 9: C-tree with prefixing operators

reading/writing statements, like in the example CSP script.

III. Examples

A. *Example1: Control loop*

In Figure 10 a basic closed loop control system, modeled in 20-sim, is shown. It consists of three functional blocks:

1. `Plant` is a controlled object characterized by its dynamic behavior. In this case it is a first order system in complex s-domain described by:

$$\mathtt{state}(s) = \frac{1}{s+0.8}\mathtt{steering}(s)$$

2. `Loop_Controller` takes care to keep the measured variable (MV, signal `state` in this case) in proportion with required set point (SP, here `reference`) signal. The most simple proportional action controller (P-controller) is used in this example:

$$\mathtt{steering}(s) = K(\mathtt{reference}(s) - \mathtt{state}(s))$$

   Coupled with `Plant`, the controller influences the controlled `state` variable by manipulating the `steering` values.

3. `Sequence_Controller` in general is a control system component governing separate control loops that maintain values of local variables (`state` in this case) according to a prescribed higher system-level sequence of activities. Having a system reduced to one control loop, the role of this component is reduced to a set point generator providing the controller with a desired reference profile.

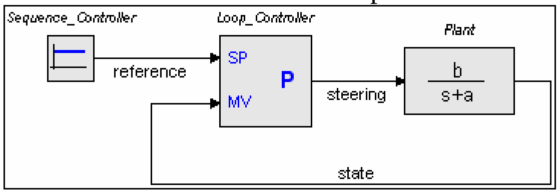


Figure 10. Control loop

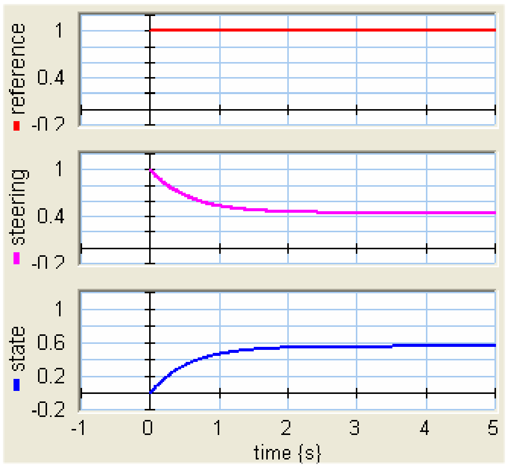


Figure 11. Response of the closed loop

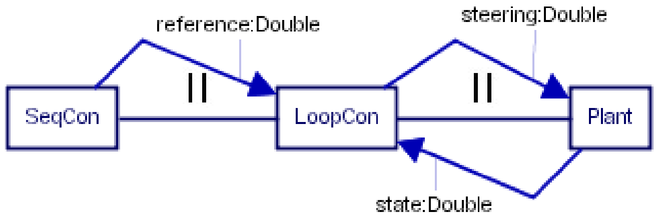


Figure 12. GML model of the block diagram

Figure 11 renders the variations of `steering` and `state` values for the `reference` being a unity step signal. Translated in a data-flow (communication) view in gCSP, the names of signals from 20-sim model coincide with names of channels among the three processes corresponding to the three functional blocks from Figure 10 . The full CSP view to the GML model of the control loop is shown in Figure 12. The names of the two leftmost blocks are abbreviated to `SeqCon` and `LoopCon`.

Compositional (concurrent) behavior of the three blocks from Figure 10 is modeled by a parallel composition on this GML model's top level by the two parallel relationships. If it is not specified differently, composing system components in a parallel execution should be a default design pattern: after all, it can be assumed that Figure 10 models three independent components that naturally operate simultaneously. This is the most general assumption, while any other restrictions (as specifying sequential design patterns) should and can be imposed explicitly.

That is actually done on the level of each process specification (Figure 13, Figure 14 and Figure 15). In order to precisely specify the order of inputting, processing and outputting the variables in the system components, sequential relationships (actually, prefixing operators) are used. As explained in the previous section, sequential relationships in combination with primitive μ indicate (endless) repetitions of the all three processes' implementations. This is a typical pattern when using computational algorithms for CSP/CT processes out of 20-sim models. Namely, the 20-sim simulation engine takes care of giving the processing algorithms of the building blocks a proper execution model. When extracted from a 20-sim model, these algorithms are "one-shot", which means that they produce outputs on basis of inputs for one sampling period. In order to let the model execute over successive sampling periods, a proper execution framework is provided by using a parallel construct among the processes that are, in turn, internally composed in sequences with the μ repetitions.

Internal structures of the 20-sim blocks (Figure 10) use ports (the same squared shapes like in gCSP) for inputting and outputting signals to and from the blocks. For specifying communication patterns within the processes representing the blocks, this port-based communication in GML is delineated by the primitive reader and writer processes. As outlined in the previous section, an association of a variable, primitive process and a port is visualized by the symbols of channels (for example, in Figure 13, those between `step`, `writer_SP` and `reference`), although the implementation in code is optimized by eliminating overhead of the channel communication. The variables are in the scope of the process where the code blocks are defined as well. The code blocks are intended for defining actual data processing within a process. Specified variables which are not connected by channels are used as local in code blocks, and are not relevant for the deadlock detection. Actually, a full process specification with all elements important for an executable implementation is far more elaborate than necessary for analysis of the communication phenomena. The diagrams are unnecessarily detailed for this purpose, which is illustrated in Figure 17. A possible simplification is used afterwards. The compositional information contained by this model in the C-tree representation is in Figure 16.

Before demonstrating checking the GML model of the closed loop example, the internals of the processes are commented. These specifications intend to capture the serialization of the data instances over the channels. Note that signals in the control loop scheme (Figure 10) abstract away the actual order of evaluating signals values. For simulation purposes, the actual order results from a causality analysis performed by the simulation tool (20-SIM in this case). On the contrary, a precisely

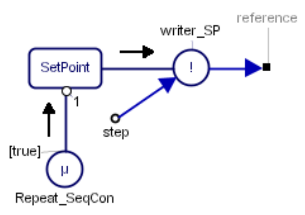


Figure 13. `SeqCon` internals

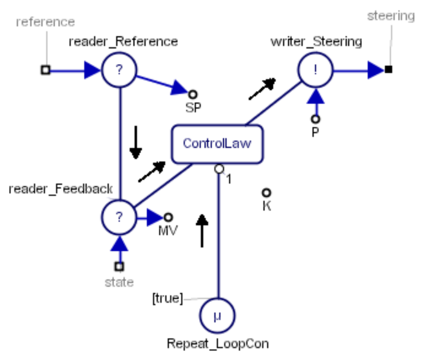


Figure 14. `LoopCon` internals

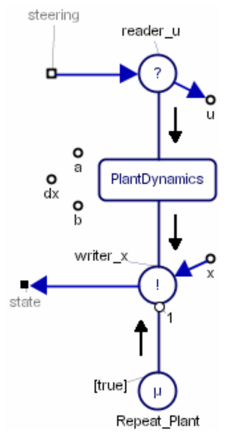
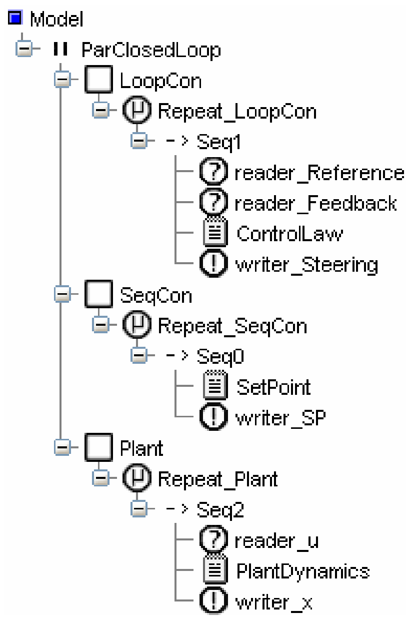


Figure 15. `Plant` internals

specified implementation requires an explicit ordering of the calculation and communication activities. Only in this way the properties of a specification, as deadlock-freedom, can be checked prior the actual deployment of the implementation.

`SeqCon` process (Figure 13) implements outputting set point values to the `reference` channel. The written instances are values of the variable `step` that is manipulated by the `SetPoint` code block. The horizontal sequential relationship indicates the order of the two elements activation (the code block `SetPoint` first and than the writer `writer_SP`) and in the C-tree (Figure 16) is specified as a prefixing (`Seq0`). The vertical relationship is a true sequential relationship used in combination with the `Repeat_SeqCon` μ process, indicating repetition of the prefixed sequence of activation. The bubble with index 1 (as described in Section II.C) indicates that the group of `SetPoint` and `writer_SP` is being repeated. The `true` value of the repetition condition implies that the repetition is endless.

Similarly, the sequence of the readers



```
Model
  ParClosedLoop
    LoopCon
      Repeat_LoopCon
        -> Seq1
          reader_Reference
          reader_Feedback
          ControlLaw
          writer_Steering
    SeqCon
      Repeat_SeqCon
        -> Seq0
          SetPoint
          writer_SP
    Plant
      Repeat_Plant
        -> Seq2
          reader_u
          PlantDynamics
          writer_x
```

Figure 16: The C-tree of the control loop example

`reader_Reference`, `reader_Feedback`, the code block `ControlLaw` and the `writer_Steering` writer is being endlessly repeated by the `Repeat_LoopCon` process (Figure 14). `reader_Reference` stores data read from the channel `reference` to the variable `SP`, while `reader_Feedback` does so with `state` and `MV`. The values of the variable `P`, produced by the code block `ControlLaw`, are written to the channel `steering` by `writer_Steering`. Variable `K` is an internal parameter of the `ControlLaw` block.

A straight-forward (or rather naïve, as the deadlock analysis will show) specification of the `Plant` process would follow a “natural” computation causality (Figure 15): first the values from the `steering` channels are read to the `u` variable by `reader_u`, than the `x` variable gets filled by `PlantDynamics`, and consequently its value is outputted to the `state` channel by `writer_x.` (Variables `a`, `b` and `dx` are used for calculating the plant dynamics).

The specification in the following CSPm script is generated by gCSP.

```
datatype Double = step_val | P_val | x_val

channel reference : Double
channel steering : Double
channel state : Double

ParClosedLoop              =              LoopCon
[|{|reference,steering,state|}|]   (SeqCon   |||
Plant)

SeqCon = reference!step_val -> SeqCon

LoopCon    =    reference?SP    ->    state?MV    ->
steering!P_val -> LoopCon

Plant = steering?u -> state!x_val -> Plant
```

gCSP generates the CSPm code in the order as in the given script: declaration of values written to the channels, channels declaration, then the network builder and finally the process specifications.

In the first row the values of all variables associated to writer processes are defined as of type `Double` – in

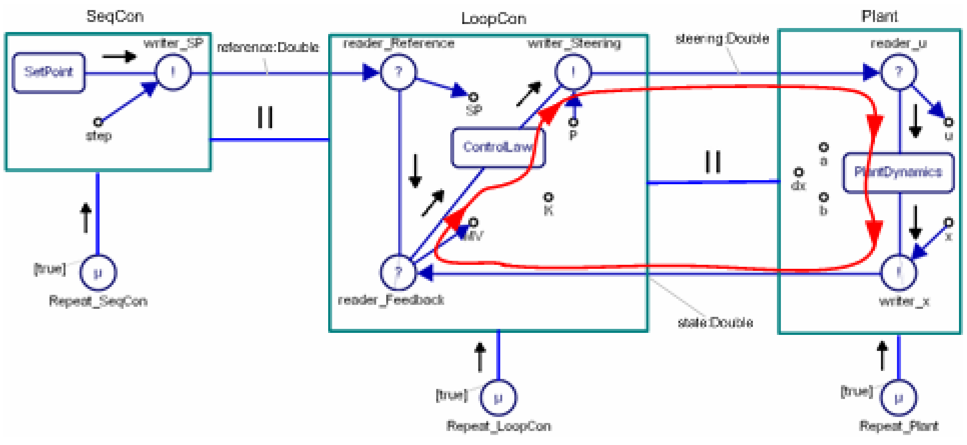


Figure 17:. Deadlock path

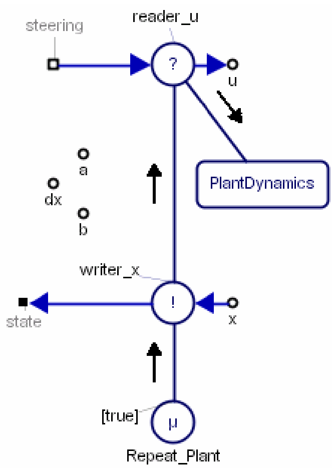


Figure 18: Modified `Plant` process

order to distinguish the values of the variables and the variables themselves, the suffix "`_val`" is used.

It is quite obvious to interpret the last three statements describing the sequences of channel activities in the system (i.e. the constituting processes). Note that code blocks' processing is not represented as being irrelevant for the communication patterns of the process – provided that *all* communication activities are captured by primitive communication processes.

The network builder `ParClosedLoop` describes the structure and communication (i.e. synchronization) of constituting process behaviors via the channels `reference`, `steering` and `state` declared to carry values of the type `Double`. Looking from behind, the `(SeqCon ||| Plant)` expression states that processes `SeqCon` and `Plant` interleave, which means that there is no direct channel connection between them. However, they are synchronized with `LoopCon` through channels `reference`, `steering` and `state` by a shared parallel operator `[| {| reference, steering, state |} |]`.

An equivalent network builder can be expressed by:

```
SeqCon [| {| reference |} |] (LoopCon [| {|
          steering, state |} |] Plant)
```

that would reflect a spatial order from Figure 12, i.e. Figure 10. However, the CSPm code generation engine determines the actual order of the processes in a network builder expression.

This script can be analyzed both by FDR against deadlock and interpreted by ProBE. For the given CSPm script, all processes exhibit a deadlock-free behavior, except the network builder! *This means that the specified system deadlocks!*

For simple examples consisting of a few processes (like this one), the reason for the deadlock can be traced in the CSPm description manually. Also, by ProBE one may inspect that `ParClosedLoop` performs just the first communication event `reference.step_val` and than stops. A by-hand analysis of the CSPm script would examine the sequences of channel activations in processes and check if the communication patterns of all processes composed in the network builder expression match. In this way, one can find a match for the first communication event on the `reference` channel (writing `reference!step_val` in `SeqCon` and reading `reference?SP` in `LoopCon`). After this event `SeqCon` repeats writing to the channel reference, but since `LoopCon` has progressed to reading from channel `state`, it cannot accept new value on channel `reference`. In the other hand, `Plant` does not offer the data at the channel `state`, rather waiting for input on channel `steering`. The reason for deadlock is reversed order of channel activations in the current states of `LoopCon` and `Plant`.

Looking back at the model on Figure 12, it is hard to find a visual interpretation of the erroneous condition by delving separately into specifications of constituent processes. However, the deadlock condition can be given very well a graphical interpretation once all process specifications are brought to the same hierarchical level.

As already has been revealed by the script analysis, the cause of deadlock are incompatible communication patterns of the processes `LoopCon` and `Plant` over the `steering` and `state` channels. The reversed order of channel activations in `LoopCon` and `Plant` is depicted as a *closed path* along the channels and channel activation sequences *oriented* by all prefixing arrows in the same direction. (So, the problem is that a closed path is oriented – in Figure 17 represented with the closed curve indicating the prefixing operators all oriented in the same direction). As already stated, the presence of the code blocks does not influence a communication within a process. Also note that coherent orientation of channels along a closed path is irrelevant and accidentally matches the orientation of the deadlocked path in this example.

Visualization of the closed (deadlocked) path gives an indication of a possible remedy. Namely, the sequence of the channel activations along the closed oriented path

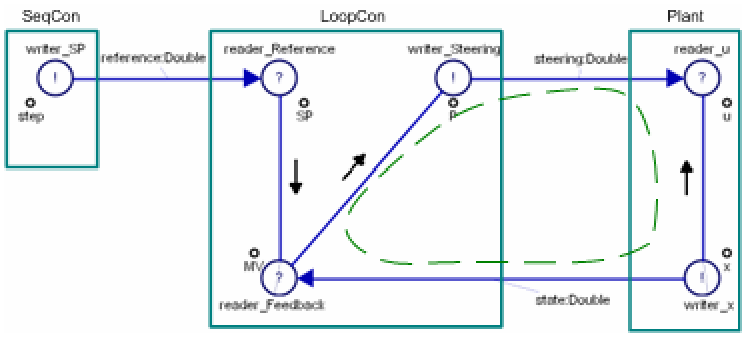


Figure 19: Deadlock-free loop

should be changed in a way the closed path is not coherently oriented any more.

In this simple case that can be done either in the `LoopCon` or `Plant` process. Note that in any case a causal order ("read inputs first, then calculate, and then write outputs") of one of the processes has to be modified, so one of them has to write an initial output value first. Effects of this kind of modifications are commented after presenting the repaired model, which follows.

By reversing the order of communication activities in the `Plant` process (Figure 18) the closed path is not coherently oriented anymore – that is indicated with the dashed closed curve in Figure 19

In this visual representation several simplifications are adopted, that make reading of more inhabited (like in the next example) graphical models simpler. The variables not communicated over channels are dropped, as well as the connections between the primitive processes and associated variables (the proximity will suffice to indicate their association). The code blocks are omitted as well, since, as described, does not influence the order of channel activations that is relevant here. The endless repetitions in the considered examples can be neglected as well, since the illustrated deadlock conditions are encountered before any of the processes reach their first repetition. Finally, parallel relationships among the processes need not be explicitly drawn, as it is stated that checking against deadlock makes sense only for parallel processes.

CSPm specification of the processes' behavior for the modified system is

```
SeqCon = reference!step_val -> SeqCon
LoopCon = reference?SP -> state?MV
            -> steering!P_val -> LoopCon
Plant = state!x_val -> steering?u -> Plant
```

A pairing of consecutive writing and reading on channels is easy to establish. Now FDR declares the deadlock freedom of the composition of the three processes.

From this example it can be concluded that a proper implementation of a physical model in a concurrent environment may not reflect always a "natural" processing causality. Before implementing (and sure before deploying the implementation of) the model, it is strongly recommended to check the model specification like it is demonstrated in this example. The checking may show that communication patterns of composed processes are exhibiting deadlock conditions. An analysis of the CSPm script or, equivalently, finding an oriented closed path in the graphical model, may indicate in which process(es) an altering of the order of communication would remedy the situation.

But a new question arises: will a modified order of activities that deviates from the original causality influence the correctness of computation? If the deadlock cannot be avoided keeping the original causality, an additional analysis should be performed. Unfortunately, there is no a general prescription how to do that, and it is out of scope of this text; however, in this example such a check is done easily. Also, an alternative way of deadlock elimination is presented. The next example shows finding a solution that avoids this problem.

The essence of the problem inherent to this example is in fact a simulation of a physical system. In reality, the components from Figure 10 really coexist (in continuous world), and physical quantities are continually present on the physical „channels" `steering` and `state`. For simulating on a digital system, an initial value on at least one channel should be initially injected. Reversing the order of communication in the `Plant` process has the same result, but is not as generic as outputting an initial value first.

The modified `Plant` outputs an initial value of the variable `x` to the channel `state`. Then a reading from the channel `reference` (by `reader_u`) is attempted, but since `LoopCon` has not yet read and processes the data from the channel `state`, `Plant` will block on the reading. After writing to `steering` by `LoopCon`, `Plant` updates its variable `u` and calculates the next state (value of variable `x`) based on previous (initial) value of `x` and just updated `u`. Since the initial value is used anyway for calculating the first `x` instance influenced by the first instance of `u` (hence the controller) – in this example reversed communication pattern that accomplishes first writing to the closed path does not affect the correctness of calculated values.

An alternative solution, often used to start up the networks of rendezvous channels, is based on the same idea. An initial writer (precisely: a writer that writes an initial value in one of the channels along a closed path) is being added in sequence with a non-terminating process.

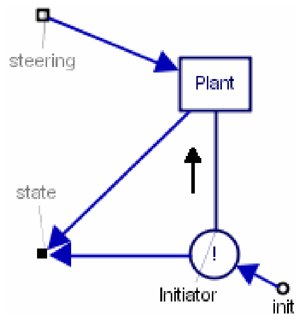


Figure 20. `Plant` in sequence with a startup writer

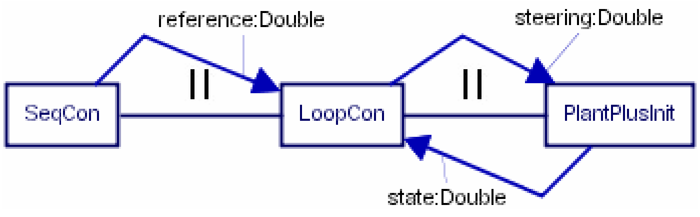


Figure 21: Modified top level model of the control loop

In this example, original `Plant` process has to be augmented by a startup writer `Initiator`. As specified in Figure 20, after writing to the `state` channel, `Initiator` terminates (since a true sequence ";" CSP operator is used), and the `Plant` process (from Figure 15) takes over. The resulting CSPm script passes the FDR deadlock check:

```
datatype Double = step_val | P_val |
                    init_val | x_val

channel state : Double
channel steering : Double
channel reference : Double

ParClosedLoop = LoopCon [|
             {| steering, state, reference |}
             |] (SeqCon ||| PlantPlusInit)
SeqCon = reference!step_val -> SeqCon
LoopCon = reference?SP -> state?MV
              -> steering!P_val -> LoopCon
PlantPlusInit = state!init_val -> SKIP ; Plant
Plant = steering?u -> state!x_val -> Plant
```

Instead of the `Plant` process, an extended version of it, `PlantPlusInit` takes place in the network builder. `PlantPlusInit` consists of the original `Plant` process sequentially preceded by a nameless process (writer `Initiator` in Figure 20) that terminates (`SKIP`) after an one-shot writing to the `state` channel (`state!init_val`).

This solution, that uses the original `Plant` sequence, additionally verifies the correctness of former deadlock solution with the modified `Plant` process: the order of communication events is the same, as well as the values of data written to the channels. Another verification can be done by generating CTC++ programs out of two versions and compare outcomes.

The disadvantage of the latter solution is a structural discrepancy with the starting 20-sim model. Namely, if the original `Plant` process is kept, it gets placed deeper in the structure, becoming the part of the newly introduced top-level `PlantPlusInit` process.

The `PlantPlusInit` process from Figure 21 with internal structure shown in Figure 20 is structurally asymmetric with respect to `SeqCon` and `LoopCon`.

A remark that holds for both deadlock solutions is that care should be taken about the initial value written to the `state` channel. This value may be copied from the 20-sim model.

B. Example 2: JIWY

In this example one part of a multimodal controller for a robotic end-effector is considered. The end-effector (called JIWY) is a small positioning device, used for orienting a web camera. The mechanical construction consists of two revolute joints allowing rotation of the camera around the horizontal and vertical axes (Figure 22). The joints are equipped with DC motors and incremental encoders.

The multimodal controller is in details presented in [23]. In this example the most important, servo position mode, is extended and modeled in a way that allows deadlock check.

Detailed modeling of this control mode in 20-SIM and GML is presented in [22]. It consists basically of two controller processes (`Horizontal` and `Vertical`) running in parallel (Figure 23). `Horizontal` is a position controller for the horizontal axis (upper motor/shaft/encoder) while the `Vertical` process controls lower motor/shaft/encoder. `SanityCheck` is a process that guards the JIWY motors of overly high steering values that may appear due to errors in controller processes or communication lines (in a distributed case) on `hor_san` and `var_san` channels. It implements a classical lightweight safety design pattern Sanity Check [26].

Controllers on JIWY have incremental encoders as feedback sensors and an analog X-Y joystick as a set point generator. Modeling readings from these components would increase the total number of processes in this example to seven, which would not contribute to the demonstration of the power of the deadlock checking procedure, although making this example more that twice as complex as the previous example. Hence only reading from one pair of sensors (encoders) is modeled by `AngleHorizontal` and `AngleVertical` processes. Modeling reading from the joystick with separate processes is not substantial (even not desired) in this example because:

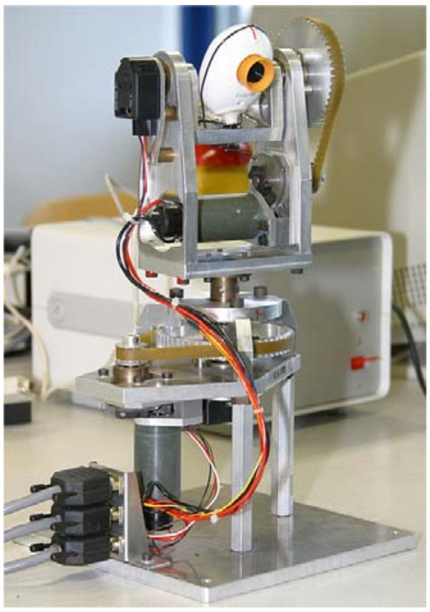


Figure 22. JIWY end-effector

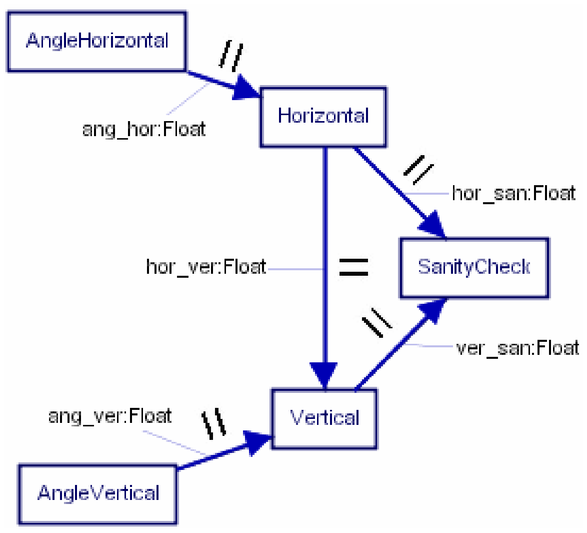


Figure 23. JIWY servo mode

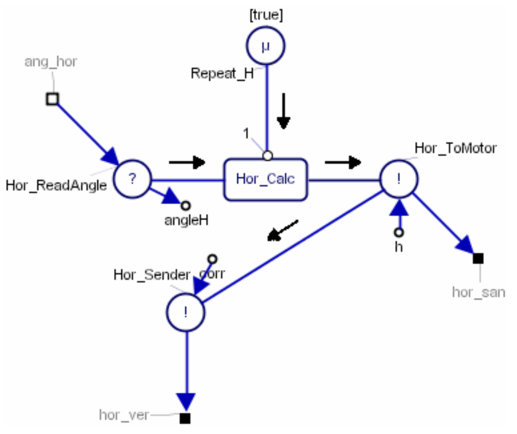


Figure 24. The `Horizontal` process

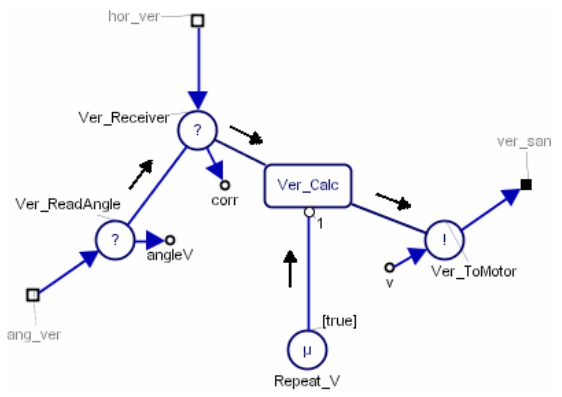


Figure 25. The `Vertical` process

1. Sampling the feedback and reference sensors happens usually consequently, thus those reading can be considered combined in the modeled communication of angle variables (that can consist of a pair of values, one for reference angle from a joystick axis and one for actual angular position measured by encoder) along the `ang_hor` and `ang_ver` channels.
2. Immediately after the first overview of the whole system (Figure 29), processes that sample angles are omitted from further analysis, since they do not participate in closed paths.
3. The entirety of the system model is already complex enough for this paper presentation, thus saving space on irrelevant details is certainly welcome.

Besides adding the `SanityCheck` process in servo control configuration modeled in [22], the model is extended by communication of a corrective value between `Horizontal` and `Vertical` processes. The additional communication substantially contributes to the risk of a deadlock condition occurrence. This kind of additions during the maintenance of existing concurrent systems is a practical problem that may seriously influence the behavior of the system at hand. For instance, after a successful end-effector prototyping a small and compact web camera may be replaced by much different device in a real robotic system that also can be itself significantly enlarged. The `Vertical` controller may need to use corrective information on the momentum of inertia in the vertical axis changing with the position of the device mounted on the horizontal shaft.

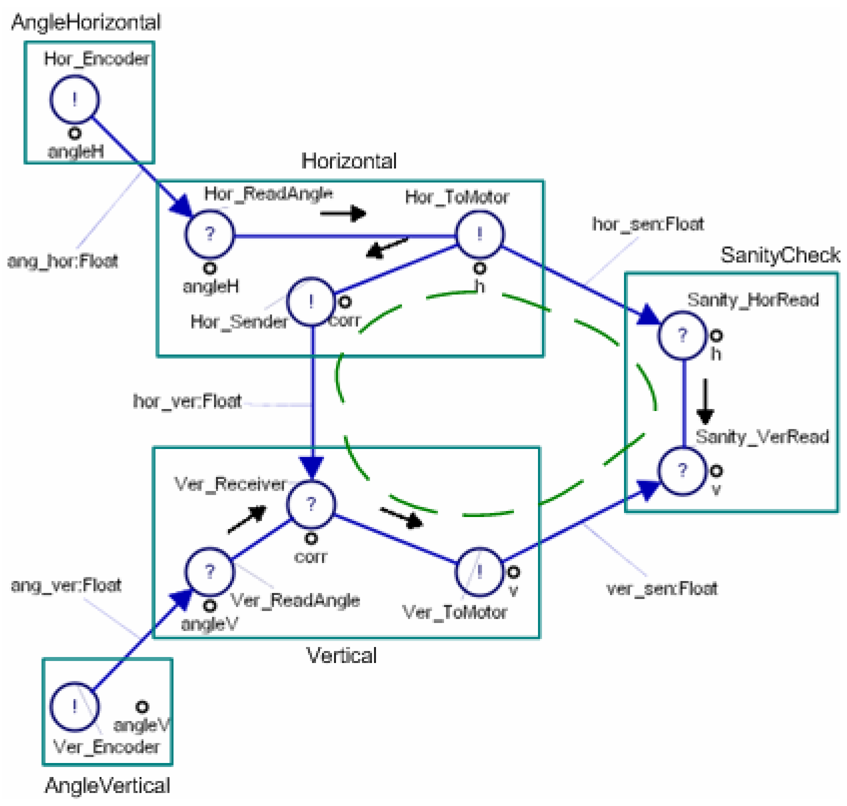


Figure 29. Communication patterns in the JIWY servo mode

The internals of five processes from Figure 23 are presented in the following five figures.

Figure 24 shows that the process `Horizontal` is initially designed first to input information on the angular position of the horizontal axis (from the channel `ang_hor` to the variable `angleH` by the `Hor_ReadAngle` reader), then activate the control law calculation for the axis contained by the `Hor_Calc` code block that stores the calculated motor steering instances to the variable `h`. After that it sends (by the writer `Hor_ToMotor`) the value of `h` along the `hor_san` channel and before the next repetition (ruled by the sequential composition with the μ processes `Repeat_H`) writes the value (by `Hor_Sender`) of the corrective variable `corr` to the `hor_ver` channel (it is assumed that this value is also calculated by `Hor_Calc`).

Figure 25 renders similar sequence for the process `Vertical`. The order is different only in the fact that for the accurate control law calculation by `Ver_Calc` reading of the `corr` variable should precede the code block. This reading is performed by the `Ver_Receiver` reader process correspondent to the `Hor_Sender` writer in `Horizontal`.

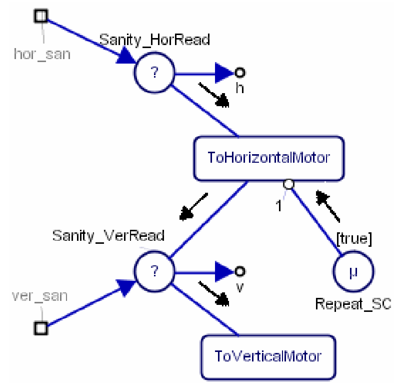


Figure 26. The SanityCheck process

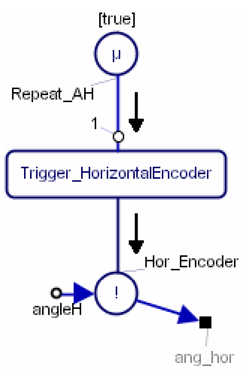


Figure 27. The `AngleHorizontal` process

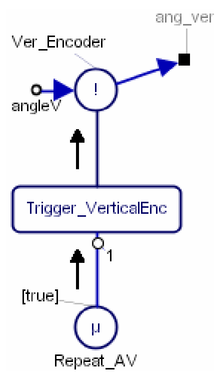


Figure 28. The `AngleVertical` process

In Figure 26 specification of the `SanityCheck` process is displayed. Via the `hor_san` and `ver_san` channels the reader processes `Sanity_HorRead` and `Sanity_VerRead` receive steering values for the motors calculated by the controllers. These values, being stored in the local variables `h` and `v` are examined by the code blocks `ToHorizontalMotor` and `ToVerticalMotor` and eventually sent to the motors (that would be modeled by a pair of writers, which are omitted as irrelevant for the further discussion). The readings are initially ordered in such that `Sanity_HorRead` is activated before `Sanity_VerRead`.

Figure 27 and Figure 28 complete the system specification with the writers sending the information of axes positions along the `ang_hor` and `ang_ver` channels.

The global overview displayed in Figure 29, as discussed in the previous example, is made more understandable after omitting some elements from processes' internals that are not relevant for interpreting (oriented) close paths. As in the previous example, all compositions are being endlessly repeated by μ processes.

By the initial design, the only closed path is not oriented, and it is indicated by the dashed curve (Figure 29). The writers in AngleHorizontal and AngleVertical cannot become part of any closed path, so they will be omitted from the further graphical analysis, while they must be present in the CSPm code, in order to contribute to the behavior of the system as a whole. CSPm model for this deadlock-free configuration is as follows.

```
datatype  Float  =  angleV_val  |  corr_val  |
angleH_val | v_val | h_val

channel ang_ver : Float
channel hor_ver : Float
channel ang_hor : Float
channel ver_san : Float
channel hor_san : Float

ServoParallel = Vertical [|
       {| ang_ver, hor_ver, ver_san |} |]
        (Horizontal
       [| {| ang_hor, hor_san |} |]
        (SanityCheck ||| (AngleHorizontal |||
        AngleVertical)))
Horizontal = ang_hor?angleH -> hor_san!h_val
       -> hor_ver!corr_val -> Horizontal
Vertical = ang_ver?angleV -> hor_ver?corr
       -> ver_san!v_val -> Vertical
SanityCheck = hor_san?h -> ver_san?v
       -> SanityCheck
AngleVertical = ang_ver!angleV_val ->
        AngleVertical
AngleHorizontal = ang_hor!angleH_val ->
       AngleHorizontal
```

It is obvious that the top network builder expression in this example gets complicated with the growth of the system, as all parallel composed processes appear in it. Consequently, composing the script manually after each modification (especially structural) is error-prone. Automatic code generation by the tool helps a lot in this respect.

In order to illustrate dealing with an occurrence of deadlock condition in the indicated closed path, suppose that during the maintenance of the system, for some reason, the order in checking steering signals within `SanityCheck` becomes reversed.

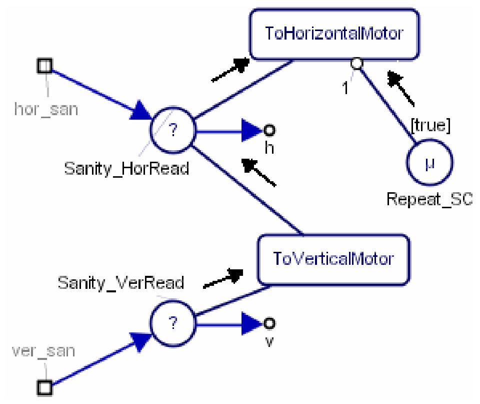


Figure 30. Reversed `SanityCheck`

The statement for the SanityCheck process in the corresponding CSPm script is

```
SanityCheck = ver_san?v -> hor_san?h ->
        SanityCheck
```

The system animation in ProBE stops after the angle reading events `ang_hor.angleH_val` and `ang_ver.angleV_val` and FDR reports deadlock for `ServoParallel`. The reason is that the modified `SanityCheck` wants first to accept the variable `v` from `Vertical`, but `Vertical` (after getting `angleV`) wants to receive `corr` from `Horizontal`, which, in turn, before sending out the value of `corr` wants to send the value of `h` to `SanityCheck`. This is a typical deadlock condition where three processes wait for getting data from each other. It is directly reflected as a closed oriented path on the overview diagram in Figure 31.

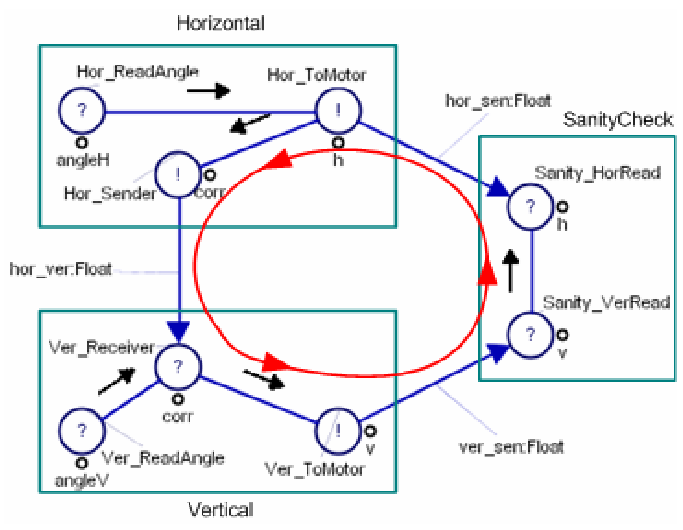


Figure 31. Deadlocked path after reversing sequence in `SanityCheck`

A solution would be, of course, restoring the `SanityCheck` back. Supposing that there was a strong reason to reverse the order of the readings in `SanityCheck`, one may look for a solution by modifying design patterns in other processes. The candidates here are `Horizontal` and `Vertical`, since they are, besides `SanityCheck`, involved in the closed path, i.e. deadlock. Looking in the processes in isolation, solving deadlock in complex network may not be an easy task, especially if causal constrains are involved. For instance, rearranging the primitive processes in `Vertical` like in Figure 33 solves the deadlock (see Figure 32). But, this solution leads to an inaccurate calculation for the steering of the vertical motor, since the value of the corrective variable from the previous sampling period is used – prefixing rules that it is read after outputting the steering instance by `Ver_ToMotor`, which means after the calculation in the current sampling period.

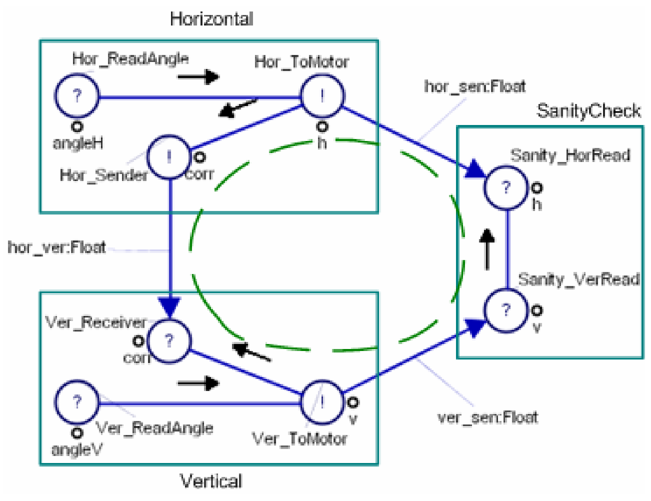


Figure 32. Deadlock freedom attained by the modification to `Vertical` (Figure 33)

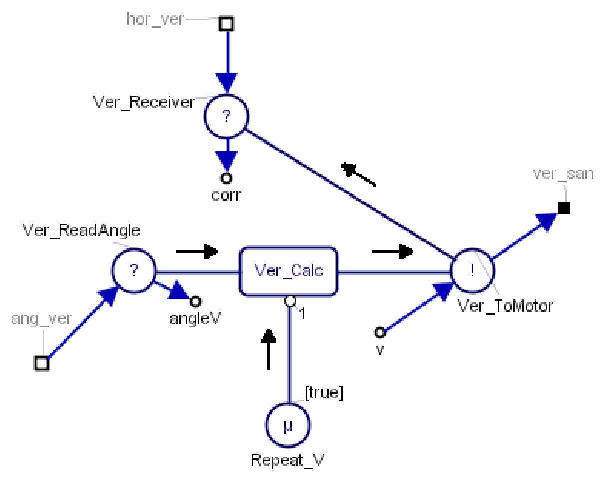


Figure 33. An inappropriate solution of the deadlock condition

Another attempt to solve the deadlock in `Vertical`, by reversing inputs of variables (Figure 34), leads only to a longer oriented closed path (Figure 35). FDR reports deadlock in generated CSPm transformation of this model.

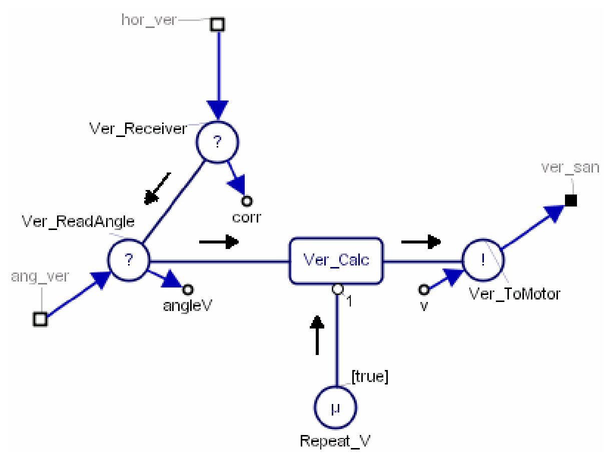


Figure 34. A modification that preserves calculation accuracy

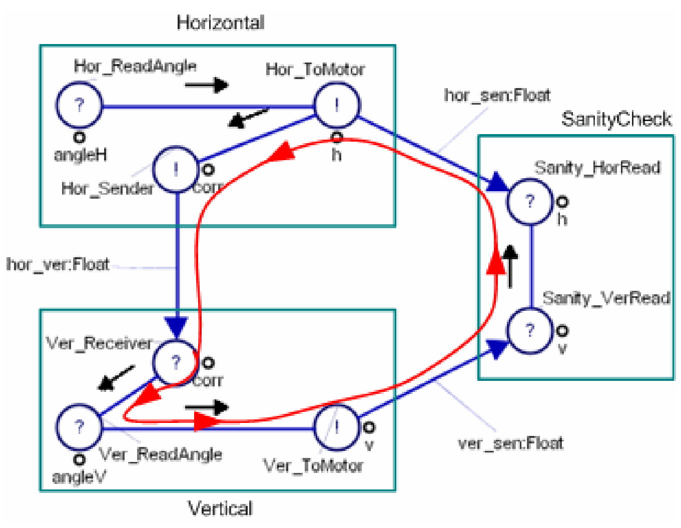


Figure 35. An unsuccessful attempt to solve the deadlock

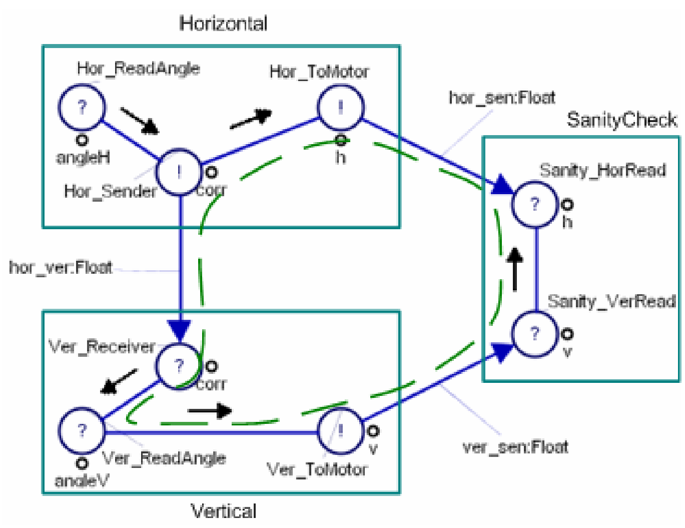


Figure 36. Deadlock free model

However, the solution is possible (and elegant), but in `Horizontal` (Figure 37). There it makes no difference if the steering is output to the motor first or the corrective value is communicated with `Vertical` first – so, the order of these two writings can be changed, and that breaks the cycle (Figure 36).

This model is transformed into the following CSPm script (only process specification are shown) that passes the FDR deadlock check.

```
Horizontal = ang_hor?angleH ->
        hor_ver!corr_val -> hor_san!h_val ->
        Horizontal

Vertical = hor_ver?corr -> ang_ver?angleV ->
        ver_san!v_val -> Vertical

SanityCheck = hor_san?h -> ver_san?v ->
        SanityCheck

AngleVertical = ang_ver!angleV_val ->
        AngleVertical

AngleHorizontal = ang_hor!angleH_val ->
        AngleHorizontal
```

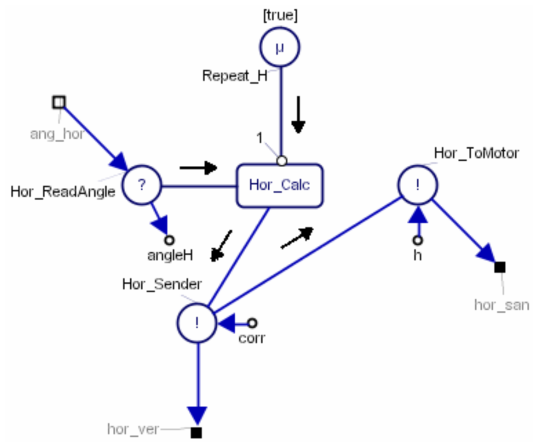


Figure 37. Sequence changed in Horizontal

IV. Conclusions and Future Work

The CSP modeling paradigm has matured over 25 years since it was established [4]. An intensive research towards practical applications in describing and analyzing concurrent systems has yielded a textual, machine-readable modeling language CSPm and the powerful computer tools for formal analysis of CSP models coded in CSPm. Creating CSPm models may require substantial knowledge of the CSP modeling principles, which have an extensive mathematical background. Moreover, manual transformation of models from various (engineering) domains to a CSPm description may be far from obvious and error-prone.

The gCSP tool spans the conceptual distance between data-flow graphical models and a subset of the CSPm expressive abilities. Therefore, the two main features of gCSP are graphical editing and automatic code generation. The graphical editor implements GMLanguage for modeling execution patterns of data-flow models. Along the three views of the graphical editor, an additional hierarchical view to graphical models in the form of a compositional tree facilitates code generation of CSPm and CTC++ transformations of a model. After successful checking of the behavioral properties upon the CSPm transformation of a GML model, CTC++ executable code may be directly used for its implementation.

Once a proper CSPm transformation of a GML model is obtained, the industrial quality model checking tool FDR can be used for testing several behavioral properties of a model description: deadlock, livelock, determinism and model refinements. In order to illustrate the matching between occurrences of deadlock in graphical models and the CSPm description generated from those graphical models, the processes are modeled as sequential. In turn, they are all connected by rendezvous channels and parallel composed, and in that they exactly correspond to the basic concurrency model of CSP, proclaimed in [4]. For this class of CSP models, a graphical interpretation of deadlock conditions, in form of closed oriented paths, may suggest modifications of the graphical models that eliminate deadlock occurrences. Of course, for proving that the deadlock is really eliminated, the CSPm script for a modified model has to be regenerated and verified by FDR.

From example III.A, it may be concluded that the closed oriented path is a necessary, but not a sufficient condition for a deadlock. Namely, it was shown that the original `Plant` specification causes a deadlock in the control loop. One possibility for solving this was demonstrated by changing the communication sequence in `Plant` – the CSPm script passes FDR analysis and the closed path is not oriented any more. However, another solution is presented as well, where the original `Plant` process is retained and another writer is added with it in sequence. This means that closed oriented path running through `Plant` is present, but it does not imply a deadlock, which is shown by the corresponding CSPm script analysis.

The debugging procedure in this example pointed out also an issue that may arise in a class of systems where a calculation initiation problem inherently exists, like in (numerical) simulations. For analyzing causality peculiarities, one should include the standard rules for this class of systems when solving possible deadlock conditions.

Restricting processes to be purely sequential has given a simple and clear graphical interpretation of deadlock conditions. However, CSPm code generation in gCSP is not limited to that restriction. Allowing internal parallelism in the processes, where it is possible, reflects the spirit of modern CSP and refines the granularity of the system. In the three core processes in example III.B consecutive activation among the primitive communication processes of the same kind may be reorganized in parallel, like in the Figure 38, Figure 39 and Figure 40 (compositional views are presented).

For this process compositions, FDR analysis upon generated CSPm script (which becomes more complicated) proves deadlock freedom of the model. Moreover, it can be said that this model is more natural – interchangeability of the readers from Figure 39 was already considered as causality-safe in solving the deadlock, while reversing writers from Figure 38 was even successful. That interchangeability makes them candidates for parallel processes. The same goes for readers in Figure 40. However, with the growth of the system, the deadlock conditions can still arise. Identifying closed oriented paths in that case would not be as obvious as for sequential processes, but the more elaborate rules may be constructed [6, 7, 27].

All in all, opposite to the complexity of algorithms for visual detection and interpretation of deadlock conditions in arbitrary CSP models, the CSPm code generation provided by gCSP produces the suitable model representation that scales linearly with a model complexity. However, the CSP scripts may become rather complicated, especially the network builders describing the synchronization patterns among parallel processes. While drawing the models with complex synchronization patterns is easy, writing a correspondent script manually is undoubtedly error-prone. Automatic code generation helps a lot in this respect. The same holds as soon as a model hierarchy gets more involved both compositionally and in the containment (parent-child) respect.

The presented framework has potentials to be

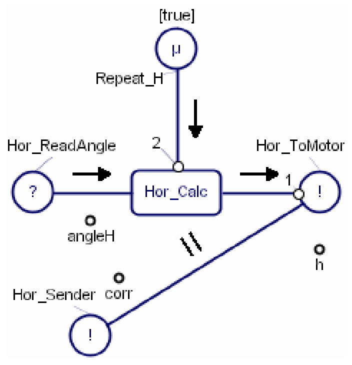


Figure 38. Parallel writers in `Horizontal`

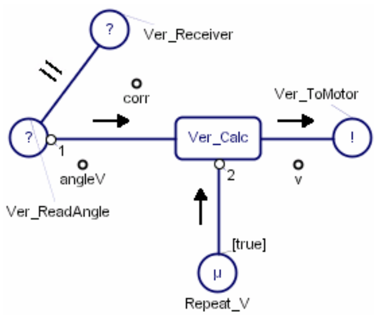


Figure 39. Parallel readers in `Vertical`

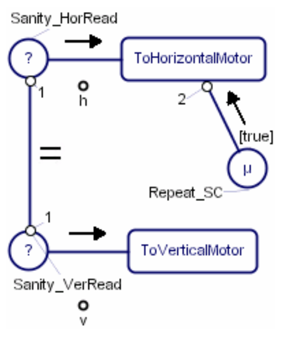


Figure 40. Parallel readers in SanityCheck

extended in the following aspects:

1. Extending the targeted subset of CSPm would allow the use of the CSPm expressiveness to a larger extent.
2. Working out the other kinds of checks (upon the very same CSPm scripts used for deadlock analysis) would make the quality assurance of the models more comprehensive.
3. The quality of the specifications would benefit significantly by implementing a ProBE-like animating functionality at the graphical level, which may assist in detecting event traces leading to a deadlock.
4. Flattening a hierarchical model with described simplifications for graphical interpretation of deadlock conditions; exploring possibilities for graphical feedback from FDR analyses.

V. Acknowledgements

Gerald Hilderink was involved in a number of discussions on implementing his GML notation.

The tool developers are indebted to Bojan Orlic, who substantially influenced the development of the C-tree, the backbone of the current code generation mechanism in gCSP [25].

References

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

[2] D. Jovanovic, G. H. Hilderink, and J. F. Broenink, "A Case Study for Tooling the Design Trajectory of Embedded Control Systems", In M. Schweizer, Ed., *Progress 2002 Workshop*. Utrecht, 2002.

[3] D. S. Jovanovic, G. H. Hilderink, and J. F. Broenink, "On properties of modeling control software for embedded control applications with CSP/CT framework", In F. Karelse, Ed., *PROGRESS 2003, Embedded Systems Symposium*. 2003.

[4] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, pp. 666-677, 1978.

[5] C. A. R. Hoare, *Communicating Sequential Processes*: Prentice Hall, 1985.

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

[7] J. M. R. Martin and S. A. Jassim, "A Tool for Proving Deadlock Freedom", *WoTug-20*. Enschede, The Netherlands, 1997.

[8] J. B. Scattergood, *The Semantics and Implementaton of Machine-Readable CSP*, D.Phil, Oxford University D.Phil thesis, 1998.

[9] K. C. J. Wijbrans, P. G. M. v. d. Klugt, and A. W. P. Bakkers, "The implementation of a transputer-based Rudder Roll Stabilization system (RRS) for ships using a CASE tool", *Proc. Transputer'91*. Sunnyvale, California, USA, 1991.

[10] K. C. J. Wijbrans, J. v. Amerongen, A. W. P. Bakkers, and J. F. Broenink, "Twente Hierarchical Embedded Systems Implementation by Simulation (Thesis) A structured approach to controller realisation on transputers," *J. A*, vol. 34, pp. 51-59, 1993.

[11] M. Visser, *Sliding mode impedance control*, MSc thesis, Faculty of Electrical Engineering, Control Laboratory, University of Twente, Enschede, Netherlands, 1997.

[12] Formal Systems, "CSP Tools" http://www.fsel.com, 2004.

[13] Formal Systems, *FDR2 User Manual*, 2003.

[14] Celoxica Ltd., *Handel-C Language Reference Manual, v3.0*, 2002.

[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, 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.

[17] B. Orlic and J. F. Broenink, "Redesign of the C++ Communicating Threads Library for Embedded Control Systems", *PROGRESS 2004*. 2004.

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

[19] 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.

[20] 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, Netherlands, 2003.

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

[22] 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.

[23] 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.

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

[25] D. S. Jovanovic, B. Orlic, G. K. Liet, and J. F. Broenink, "gCSP: A Graphical Tool for Designing CSP Systems", In I. East, J. Martin, P. Welch, D. Duce, and M. Green, Eds., *Communicating Process Architectures 2004*. Oxford, UK, 2004.

[26] B. P. Douglass, *Real-Time Design Patterns: robuts scalabe architecture for real-time systems*, 1 ed. Boston: Pearson Education, 2003.

[27] P. H. Welch, "Emulating Digital Logic Using Transputer Networks," in *Parallel Architectures and Languages Europe, LNCS 258*: Springer-Verlag, 1987.