Communicating Process Architectures 2000
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 2000

CSP Design Model and Tool Support

H.J. Volkerink, G.H. Hilderink, J.F. Broenink, W.A. Vervoort, A.W.P Bakkers
University of Twente, P.O.Box 217, 7500 AE Enschede
Control Laboratory, The Netherlands

Abstract: The CSP paradigm is known as a powerful concept for designing and analysing the architectural and behavioural parts of concurrent software. Although the theory of CSP is useful for mathematicians, the programming language occam has been derived from CSP that is useful for any engineering practice. Nowadays, the concept of occam/CSP can be used for almost every object-oriented programming language. This paper describes a tree-based description model and prototype tool that elevates the use of occam/CSP concepts at the design level and performs code generation to Java, C, C++, and machine-readable CSP for the level of implementation. The tree-based description model can be used to browse through the generated source code. The tool is a kind of browser that is able to assist modern workbenches (like Borland Builder, Microsoft Visual C++ and 20-SIM) with coding concurrency. The tool will guide the user through the design trajectory using support messages and several semantic and syntax rule checks. The machine-readable CSP can be read by FDR, enabling more advanced analysis on the design. Early experiments with the prototype tool show that the browser concept, combined with the tree-based description model, enables a user-friendly way to create a design using the CSP concepts and benefits. The design tool is available from our URL, http://www.rt.el.utwente.nl/javapp.

1. Introduction

The theory of CSP (*C*ommunicating *S*equential *P*rocesses) fully specifies the behaviour of synchronisation of threads at a higher level of abstraction, which is based on processes, compositions and synchronisation primitives [15]. It provides a mathematical notation for describing patterns of communication by algebraic expressions and it comprehends a formal proof for analysing, verifying and eliminating among others race hazards, deadlocks, livelock and starvation. The CSP concept is well thought-out and provides fundamental concepts for realising concurrent software for real-time and embedded systems. These fundamental concepts form a formal foundation for tools, programming languages, design methods and libraries/kernels for supporting concurrency at all levels of software engineering.

Currently, research by Peter Welch at the University at Kent (UK) [24] and by Gerald Hilderink at the University of Twente (NL) [8] resulted in packages/kernels that enable the use of the occam/CSP concepts for other programming languages, such as Java, C, and C++. For Java there is the *Communicating Threads for Java (CTJ)* package [9] and the *Java Communicating Sequential Processes (JCSP)* package [23]; for C there is the *Communicating Threads for C (CTC)* package [11] and the *CCSP* package [19]; and for C++ there is the *Communicating Threads for C++ (CTCPP)* package [11]. The CSP model implements a much more reliable thread pattern than for instance the conventional Java thread model [12].

The research presented here is initiated by finding a bridge between the modelling and simulation package 20-SIM [2] and the previously mentioned CTJ package. 20-SIM is a modelling and simulation package, suitable for modelling and simulation of the dynamic behaviour of engineering systems. Engineering systems as application domain means that focus is on systems spanning multiple physical domains as well as the information domain. Control applications, as developed in 20-SIM, are characterised by the fact that they are based upon abstract data-flow modelling concepts of hierarchical structured object models consisting of bond-graphs and block-diagrams. The modelling approach is closely related to *object-oriented physical-systems modelling*, which is currently often used. The 20-SIM models are declarative, hierarchically structured, and encapsulation is fully supported. Furthermore, due to allowing hierarchy, the notion of *definition* and *use* of models can be distinguished. Extending the modelling and code generation modules of 20-SIM with the occam/CSP concepts will result in more powerful industrial applications that are based on the sound foundation of the theory of CSP. To enable integration, an appropriate CSP design model is needed.

The mathematical notation of CSP is detailed and powerful, but does not give an optimal design overview. Gee [6] studied three real-time development methods in the context of occam. The development methods are the Ward and Mellor method [22], the Jackson System Development method [17] and the Mascot method [20]. A problem encountered with these methods is representing ALT constructs in the diagrams; i.e. showing alternation as well as potential parallelism. Besides that, there is a lack of means of conveniently depicting the hierarchy of processes, which exists in any occam program. To overcome these problems a more radical approach than adding complexity to diagrams or not showing all the information is advised. It was noted that linking the diagrams with a formal notation such as CSP could facilitate automatic generation of code for real-time systems. Recently, also several design/specification methods have been developed. The most widely used at this moment is the UML method [3]. The application domain of UML is huge. UML is object-oriented, whereas CSP is process-oriented [23]. The research behind CTJ showed that the CSP concepts could be modelled in UML in the form of design patterns. However, describing the CSP concepts in UML does not always give a good overview of the design (e.g. priority/parallel/choice relationships), which is induced by weaknesses in UML.

This paper describes a tree-based description model together with a prototype tool that is able to describe occam/CSP aspects visually. Optionally, higher-level visualisation methods are possible or easily made possible in the 20-SIM framework. The prototype tool, based on the description model, is a kind of browser that is able to assist modern workbenches with coding concurrency. In our field of control engineering, the browser can serve as a supplemental tool for 20-SIM, allowing the control engineer to refine control designs with knowledge about concurrency and details for the dedicated hardware. An important advantage is that this knowledge is added aside from the original design so that the original design will not unnecessary be messed up with this kind of information. As a result, the original design will continue to scale well with complexity and this likely preserves the readability of the design.The tree-based description model can be described with a developed specification language.

This specification language is explained in section 2. Section 3 describes the tree-based description model. Section 4 describes the developed prototype tool. Section 5 describes a case with the developed prototype tool. The paper ends with the conclusions in section 6.

2. Specification Language

The developed specification language CsPSPEC forms the basis of the tree-based description model and the tool. It enables a formal description of the semantics and syntax that is compatible with the target packages of CTJ, CTC and CTCPP, for respectively Java, C and C++[8]. It can be seen as an intermediate language between the generated code and the design.

Both the specification language and occam [16] are based on CSP concepts, and consequently, there is a strong resemblance. However, our specification language features enhancements that are common in the object-oriented paradigm. These enhancements provide a connection with modern object-oriented languages and methodologies, such as C++, Java, and UML [3]. Besides that, the specification language features additional/modified instructions to describe for example: systems, processors, link drivers (this is a device driver, see section 3.5), bioses (this is a set of link drivers, see section 3.5), and to enable a mapping with the tree-based description model. The specification language focuses on the patterns of communication and the hierarchy of the execution framework of the processes, that is similar to occam. However, occam has much more instructions to describe the functionality of the processes in more detail. In our specification language these details can be embedded using the code segment instruction. The code segment instruction allows the user to add specific instructions in a target language to be executed by the process. The Backus-Naur (BNF) based description [4] of the specification language is given in appendix A.

3. Tree-based description model

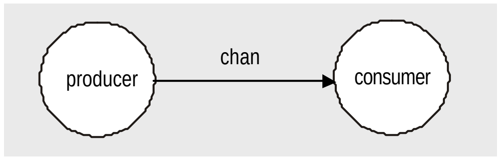
The patterns of communication and the hierarchy of the execution framework of a design can be described using the tree-based description model. The tree-based description model consists of a tree structure of elements. Each tree element represents a part of the design. The design tool comprises unique dialog boxes for every tree element type for specifying its properties.

When the designed hardware-independent processes are targeted on a distributed heterogeneous hardware architecture [1], this also affects the software. These consequences can also be modelled. The tree-based description model will be explained incrementally by describing simple examples, if possible in combination with the corresponding pseudo occam code. For space reasons, not all the properties of the tree element types will be described here.

3.1 The Producer-Consumer example

This section will describe an example in which two processes, a producer and a consumer, communicate with each other. The rendezvous communication between the producer and consumer takes place through a channel, see figure 1.

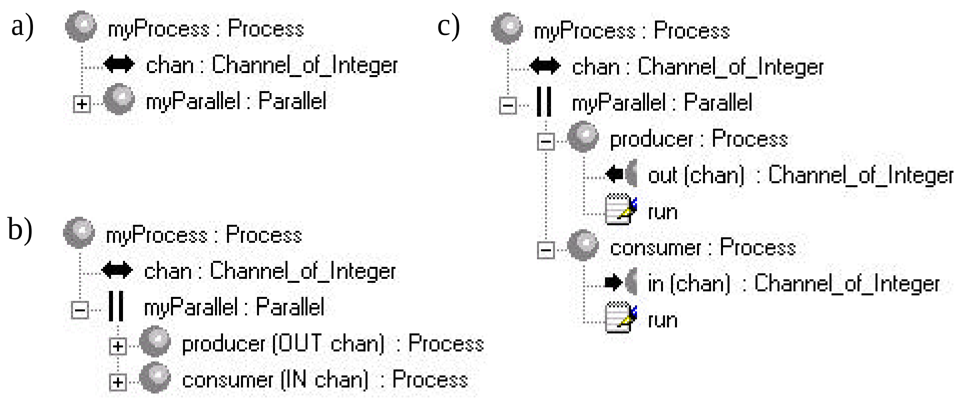
a)



b)
```
CHAN OF INT chan
PAR
     producer(chan)
     consumer(chan)
```

Figure 1: Producer-Consumer example: (a) data-flow graph, (b) pseudo occam code

The producer process writes a value to the channel and the consumer process reads that value from the channel. The data will be copied from the producer to the consumer only if they both commit in communication, otherwise they wait for each other. After the communication has been completed, they both continue. The tree-based description model of this example is illustrated in figure 2.



a)

```
myProcess : Process
    chan : Channel_of_Integer
  + myParallel : Parallel
```

b)

```
myProcess : Process
    chan : Channel_of_Integer
  - || myParallel : Parallel
      + producer (OUT chan) : Process
      + consumer (IN chan) : Process
```

c)

```
myProcess : Process
    chan : Channel_of_Integer
  - || myParallel : Parallel
      - producer : Process
            out (chan) : Channel_of_Integer
            run
      - consumer : Process
            in (chan) : Channel_of_Integer
            run
```

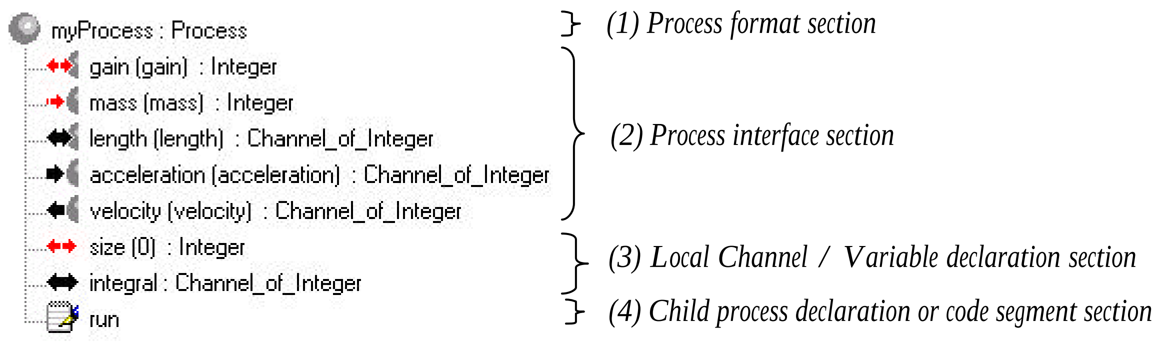
Figure 2: Producer-Consumer example: (a) parallel folded, (b) parallel unfolded, (c) all unfolded

The process `myProcess` (◉) consists of a channel declaration `chan` (⬌) and a sub-process parallel (◉), see figure 2a. The identifier `Process` after the colon designates the class from which `myProcess` is instantiated. Multiple instances can be created from the same class. The ◉ icon always designates a process. The unfolded process parallel shows that it is actually a parallel construct process (||). The parallel construct process is a composition of the `producer` process and the `consumer` process, see figure 2b.

The unfolded `producer` process, in figure 2c, shows that the process is a custom process (◉) that specifies a process output element (◀) and a code segment element (✎). The channel output element out shows that the process writes to the channel, and the code segment run describes the run body of the process. The custom process will be described in section 3.2 and the construct processes will be described in section 3.3.

3.2 *The custom process in detail – The communication interfaces and declarations*

The custom process element (◉) can have a list of child elements. This list can be classified in four sections: (1) the process identification section, (2) the process interface section, (3) the local channel/variable declaration section, and (4) the process declaration or code segment section. This is illustrated in figure 3.

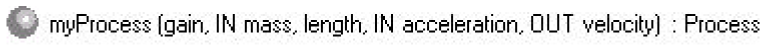


```
myProcess : Process
    gain (gain) : Integer
    mass (mass) : Integer
    length (length) : Channel_of_Integer
    acceleration (acceleration) : Channel_of_Integer
    velocity (velocity) : Channel_of_Integer
    size (0) : Integer
    integral : Channel_of_Integer
    run
```

(1) Process format section
(2) Process interface section
(3) Local Channel / Variable declaration section
(4) Child process declaration or code segment section

Figure 3: The four sections of the custom process

The *process identification section* (1) is the format of the custom process element. It identifies the process being an instance of its class. Every tree element in the tree-based description model corresponds with a unique instruction in the specification language. The conversion from the specification language instruction to the tree element format is done by a set of format syntaxes that are specific to the tree element type. The format syntax is such that the user will get the right information and will be protected from too much detail. The resulting format consists of a unique element icon, based on the symbols of the mathematical notation of CSP [15], together with a string for the element arguments. The Java / UML naming convention is used, i.e. an identifier starts with a lower case character and a class name starts with a capital letter. An instruction in the specification language can have multiple arguments (e.g. *identifier field*, *description field, and package name)*. However, the format of the corresponding tree element does not show all these arguments, because some of the arguments are less important in order to get an overview and understanding of the design. In addition, the used format syntax depends on whether the tree element is folded or unfolded. For instance, if the custom process of figure 3 is folded, the format of the element will also include its interface identifiers. This is illustrated below.


myProcess (gain, IN mass, length, IN acceleration, OUT velocity) : Process

The arguments between brackets are automatically generated from the process interface section of the custom process. The IN keyword specifies an input-only argument, the OUT keyword specifies an output-only argument, and no keyword specifies a pass-through or bi-directional argument.

The *process interface section* (2) describes the process interface. The process interface with channels can be described using the channel input elements (➔), the channel output elements (←), and the channel pass-through elements (↔). The direction of the channel pass-through element is determined by the connected element. The pass-though channel element is rarely used. However, the possibility of having a pass-through channel element enhances the flexibility of the model.

In principle, processes should communicate through channels. However, in the case of sequentially executed processes, it can be safe to communicate through variables (or non-blocking channels). When sequentially executed processes are communicating with each other through a shared channel, the system will deadlock on the rendezvous synchronization of this channel. Variables are also useful for passing initial parameters to processes. Consequently, variable input elements (➔), variable output elements (←) and variable pass-through elements (↔) are included.

A channel process interface element can be connected to a channel declaration element or a channel process interface element that is in scope. The variable process interface elements can be connected to a variable declaration element or a variable process interface element that is in scope. The scope of the variables, channels, and processes starts at the declaration or interface section of the first parent custom process element and stops at the first child custom process element. Thus, a construct process (section 3.3), like for instance the parallel construct process, does not specify a process interface and does not limit the scope of the channels, variables and processes. One can use channels or variables beyond the scope by defining them in the corresponding interface section. This is illustrated in figure 2c. The scope of the declared channel starts at the custom process `myProcess`, goes through the `parallel` construct process and stops at the `producer` process. At the `producer` process the channel is interfaced through the channel output element. Consequently, the body of the process can use the channel. The same holds for the `consumer` process.

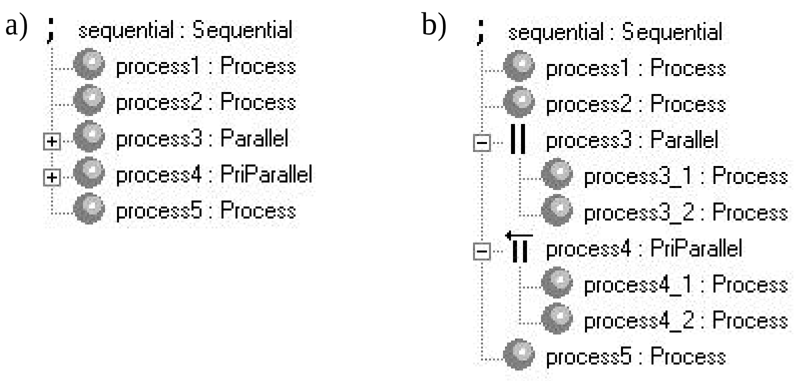
Every channel and variable input/output/pass-through element in the process interface section are references (aliases) to its instance. They are specified by a local identifier name. This aliasing functionality supports the reusability of processes. It makes the used (local) identifier names independent of the context in which the processes are used (see for instance the `producer` process in figure 2c). In addition to this, every channel and variable input/output/pass-through element comprises the type of communication. This is needed for the code generator, and it will enable the tool to prevent mismatches between communication types. Default communication types are: Boolean, Byte, Char, Double, Float, Integer, Long, Short, Object and Any [9].

The *local channel/variable declaration section* (3) declares the local channels and local variables that are used to connect the sub-processes. The channel declaration element (⬌) declares a channel and the variable declaration element ↔) declares a variable. The format of the variable declaration element `size` shows that the initial value of the variable is 0. The default communication types are similar to the communication types mentioned before.

The *process declaration or code segment section* (4) consists of a process element (●) or a code segment element (📝).

3.3 *Construct Processes – (pri)Alternative, (pri)Parallel and Sequential processes*

The construct process is a composition of processes. Consequently, the attached folded icon is ●. The construct process element does not have a process interface section. Hence, a construct process does not limit the scope of a variable, channel or process identifier. The only valid child elements are process elements. When the target language is Java (CTJ), the construct process does not necessary have an identifier. In that case it is also called an *anonymous* process. Because the construct process is a process itself, it is possible to make a hierarchy of processes. Figure 4 illustrates a sequential composition of processes, parallel processes, and priority-ordered parallel processes.

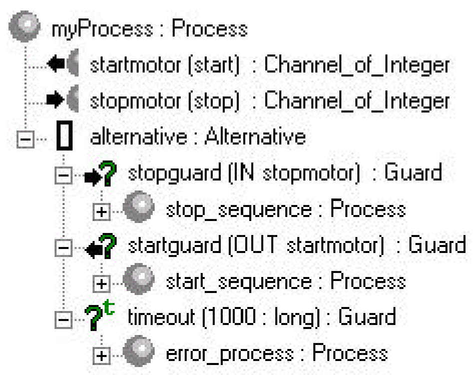


c)
```
SEQ
  process1
  process2
  PAR
    process3_1
    process3_2
  PRIPAR
    process4_1
    process4_2
  process5
```

Figure 4: Composition of processes: (a) all folded, (b) all unfolded, (c) pseudo occam code

Processes `process1`, `process2`, `process3`, `process4`, and `process5` are executed sequentially, see figure 4a. Furthermore, `process3` consists of `process3_1` and `process3_2` that are executed in parallel, see figure 4b. In this case the code generator and the runtime kernel will give each process a separate thread of control with the same priority that is inherited from the priority of the parallel construct. Process `process4` consists of `process4_1` and `process4_2` that are executed in parallel with different priorities, see figure 4b. The process `process4_1` has a higher priority than `process4_2`. The processes are executed in order of priority, but they do not have priority by itself. In other words, the user does not assign a priority to a process. The philosophy is that the priority number of a process is an implementation issue and not a design issue. The designer wants to specify a process that must be executed with a higher, equal, or lower priority than another process and not by some number [10]. The code generator and runtime kernel will solve the ordering of the priorities. All the composition processes will terminate when all their child processes are terminated.

The last two compositions of processes are the alternative process and the priority-ordered alternative process. The alternative process is illustrated in figure 5.

a)


```
myProcess : Process
    startmotor (start)  : Channel_of_Integer
    stopmotor (stop)  : Channel_of_Integer
    alternative : Alternative
        stopguard (IN stopmotor)  : Guard
            stop_sequence : Process
        startguard (OUT startmotor)  : Guard
            start_sequence : Process
        timeout (1000 : long) : Guard
            error_process : Process
```

b)

```
ALT
  stopmotor?x            -- stop
     stop_sequence(x)
  startmotor!y           -- start
     start_sequence(y)
  timer? AFTER 1000      -- timer
     error_process
```

Figure 5: Alternative process composition: (a) Tree-based description model, (b) pseudo occam code

This example introduces the alternative process element (▯), the timeout guard element ($?^t$), the input guard element (➡?), and the output guard element (⬅?). Together with the skip guard element (✓?), these guards are the only valid child elements of an alternative.

A guard element itself can only have one process element as a child. That process will be executed when the corresponding guard becomes ready. The alternative process terminates, when one selected guarded process has been terminated successfully.

The input guard and output guard become ready when the corresponding input event respectively output event is initiated. Occam does not allow output guards for safety reasons [13]. However, the tree-based description model does allow output guards, because CSP allows output guards. One must make sure that no input guard is connected to an output guard, because in this situation both guards will never meet, i.e. starvation results. The semantic rule checker in the tool will prevent the user from connecting an input guard element to an output guard element through a channel.

The timeout guard becomes ready after the specified time, when no other guard becomes ready. This means that, in the example illustrated in figure 5, the `error_process` process will be executed after 1000μs, unless the `startmotor` event or the `stopmotor` event took place before this timeout. The skip guard is always ready and lets the alternative process continue when no guard is ready (A skip guard is similar to a timeout guard with a timeout time equal to 0μs). Consequently, a timeout guard in combination with a skip guard will never become ready. Because of this, the skip guard is not illustrated in figure 5.

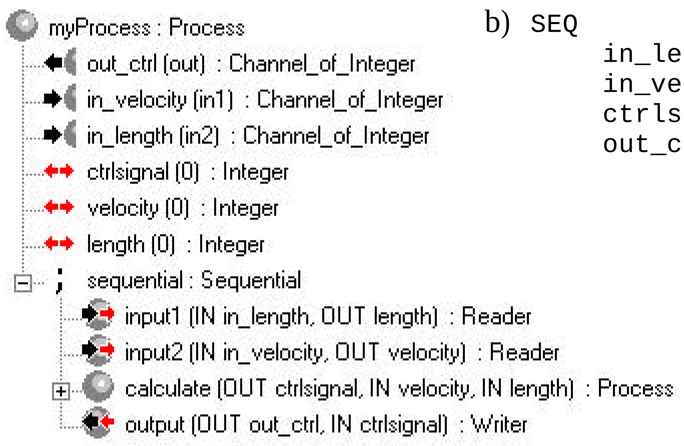
The conventional alternative process is *fair* selective, i.e., the fairness criteria used is that the guards are round-robin prioritised with the guard chosen last time getting the lowest priority next time [13]. In addition to the alternative process element (▯), it is also possible to use the priority-ordered alternative process element (↰▯). If, in the case of a priority-ordered alternative process, more than one guard becomes ready, the guard higher in the list of children will be executed, since this guard has a higher priority.

3.4 Leaf processes – the Skip, Stop, Reader, Writer and Timeslicer processes

There are several leaf process elements, i.e. elements that are not allowed to have children. The skip, stop, reader, writer and timeslicer process elements will be described in this section. The corresponding object of a skip process element (✓) immediately terminates after it is invoked. The corresponding object of a stop process element (✗) does not do anything, but does not terminate either. The object of a reader process element (➡) reads a value from a channel and puts this value in a variable. The object of a writer process (⬅) writes a value from a variable to a channel.

The reader and writer process elements are illustrated in figure 6. This figure describes an I/O sequential process.

a)



```
myProcess : Process
  out_ctrl (out) : Channel_of_Integer
  in_velocity (in1) : Channel_of_Integer
  in_length (in2) : Channel_of_Integer
  ctrlsignal (0) : Integer
  velocity (0) : Integer
  length (0) : Integer
  sequential : Sequential
    input1 (IN in_length, OUT length) : Reader
    input2 (IN in_velocity, OUT velocity) : Reader
    calculate (OUT ctrlsignal, IN velocity, IN length) : Process
    output (OUT out_ctrl, IN ctrlsignal) : Writer
```

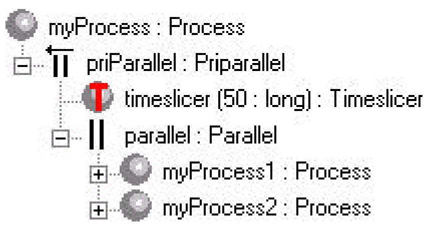
b)

```
SEQ
  in_length ? length
  in_velocity ? velocity
  ctrlsignal = f(velocity,length)
  out_ctrl ! ctrlsignal
```

Figure 6: I/O Sequential process: (a) Tree-based description model, (b) pseudo occam code

The custom process element `calculate` uses the variables `velocity` and `length` that are set by the reader processes of respectively the `input1` element and the `input2` element. The `calculate` process calculates the resulting output and writes it to the variable `ctrlsignal`. The writer process `output` writes the value of this variable to the `out_ctrl` channel.

In certain applications one wants to give every process an equal share of dedicated processor time. This can be described using the timeslicer process element (T). The timeslicer process element, as shown in figure 7, has a quantum time property (50 µs). The timeslicer process wakes-up after the specified quantum time. At that moment it pre-empts the parallel construct. The pre-empted process, `process1` or `process2`, will be placed on the queue and after the timeslicer process sleeps again, the next ready process, `process2` or `process1`, will run. With the timeslicer concept timeslicing is optional and it can be implemented in a CSP manner [12].



```
myProcess : Process
  priParallel : Priparallel
    timeslicer (50 : long) : Timeslicer
    parallel : Parallel
      myProcess1 : Process
      myProcess2 : Process
```

Figure 7: The Timeslicer process

3.5 *Hardware dependency – Systems, Processors, Bios and Link drivers*

The previous sections described how the tree-based description model is able to model the patterns of communication and the hierarchy of the execution framework of the design. When the designed processes are targeted on a distributed heterogeneous hardware architecture [1], this also has consequences for the software. These consequences can also be modelled in the tree-based description model. Consider the Producer-Consumer example again. Imagine the producer and consumer being placed on separate processors. This is, together with the link driver concept, illustrated in figure 8.

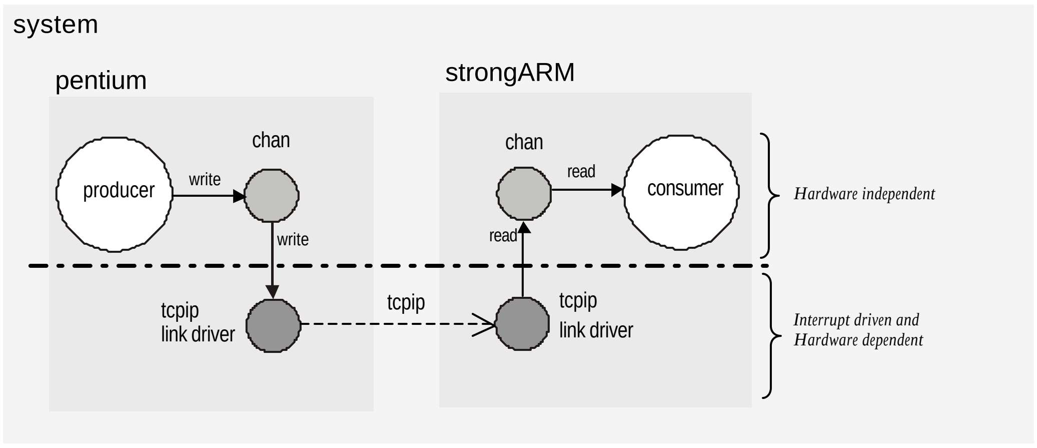
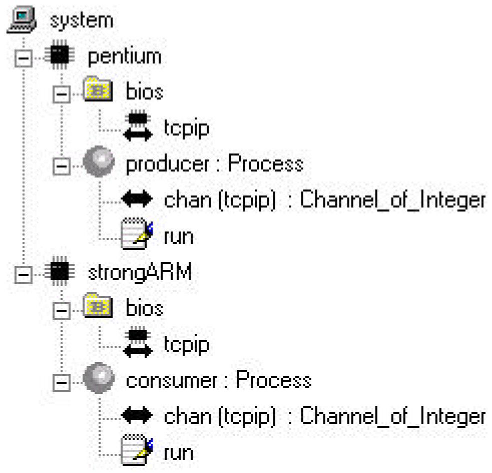


Figure 8: Data-flow graph of the distributed Producer-Consumer example

A link driver reads and writes directly from and to the device's registers, deals with the device's interrupts and has knowledge about (hardware) protocols. A link driver is an object that can be plugged into a channel [9]. Through this link driver, the channel is able to address hardware and to realise the physical data transfer. A process itself should never directly address hardware or link drivers. This way the processes and also the channels remain free from any hardware dependent code. Link drivers are able to address many different kinds of hardware as for example A/D converters, D/A converters, quadrature encoders, PWM modulators, serial links and TCP/IP links.

The tree-based description model of the distributed Producer-Consumer example is illustrated in figure 9.



```
system
  pentium
    bios
      tcpip
    producer : Process
      chan (tcpip) : Channel_of_Integer
      run
  strongARM
    bios
      tcpip
    consumer : Process
      chan (tcpip) : Channel_of_Integer
      run
```

Figure 9: The tree-based description model of the distributed Producer-Consumer example

The system element () describes a system. A system can consist of multiple processor elements (), multiple (sub)system elements () and only one child bios element (). This enables the modelling of a hierarchical structure of the distributed heterogeneous hardware architecture [1].

A processor element can consist of one child bios element and one child process element. The bios element describes the basic input-output system of a processor or system. The bios element can have multiple child bios elements and link driver elements. The actual bios description is performed by link driver elements (). The bios elements are only introduced to structure the link driver set, because in actual designs, link driver sets can be inconveniently large.

The format of the channel declaration element `chan` (⬌) shows that the `tcpip` link driver is plugged into the channel. The scope of a link driver element starts at the first parent system element or parent processor element. All the child elements of this processor or system that are deeper in the hierarchy are in scope of the link driver (for example, in figure 9, the channel (`chan`) declared under the `producer` process is in scope of the `tcpip` link driver declared under the bios of the `pentium` processor). A channel that uses a link driver that performs a link between two processes should be plugged into a channel declaration directly under the custom process one level higher in the hierarchy (see in figure 2c the `chan` channel in both the `producer` process and the `consumer` process). Besides that, when there is a link driver that performs a link between two processors, the channels should be declared directly under the main processes of the processors (see figure 9). This is necessary to prevent communication between two processes that is not visible in the process interface section of the processes. It is not necessary from the implementation point of view, only from the design point of view.

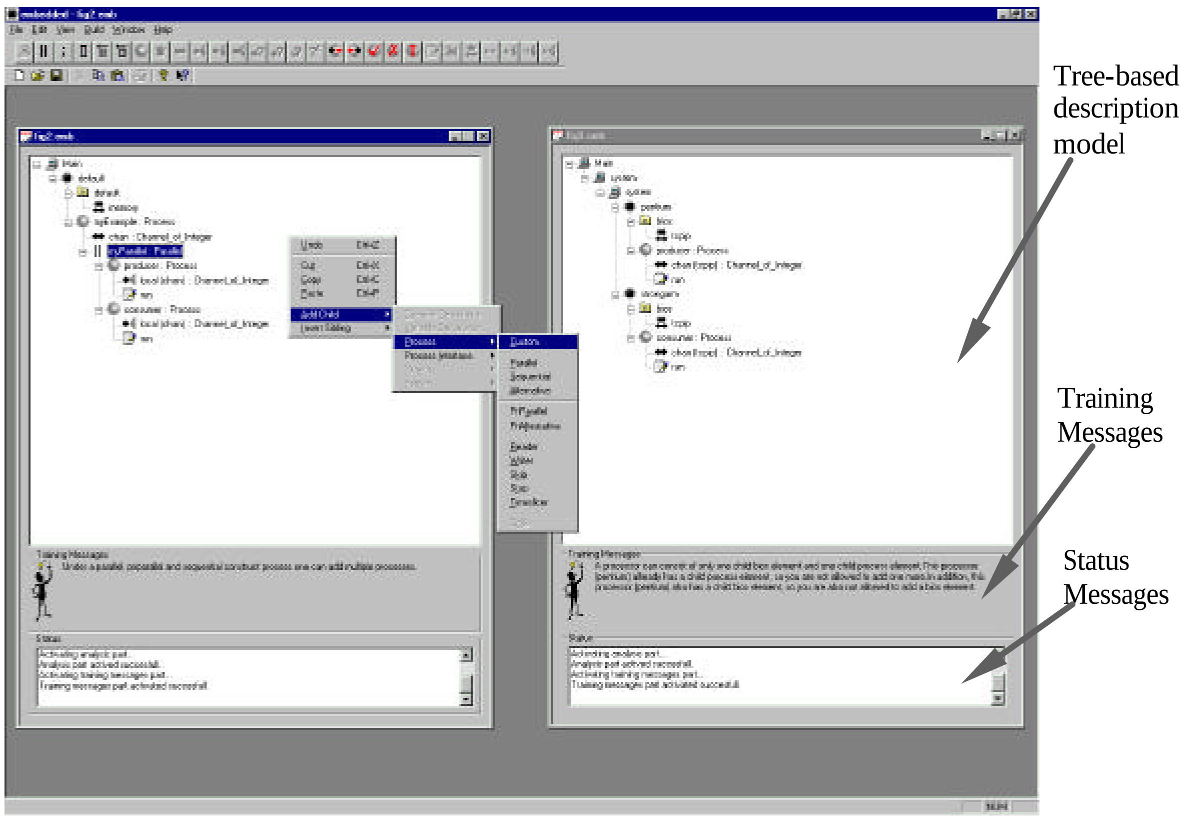
The objective of the tree-based description model is to visualise the software aspects and not the physical hardware aspects. As a consequence, one does not see a physical link between two processors directly. One can see this implicitly because both processors use link driver elements that implement the same communication protocol.

4. The tree-based description model prototype tool

The previously used figures of the tree-based description model in this paper, are actually snapshots of the tree view of the developed prototype tool.

The prototype tool comprises five parts: (1) the *analysis part*, (2) the *code generator part*, (3) the *database part*, (4) the *information message engine* and (5) the *user interface with the design entry part*. The *analysis part* comprises the semantic rule checker and the syntax rule checker. A subset of the rule checks will be performed immediately when elements are added or modified. The remaining rule checks cannot be applied immediately, but must be temporarily suspended because the design passes through an inconsistent state. The *code generator part* supports the Java target language using the CTJ package [9]. By generating machine-readable CSP [21] more advanced analysis can be done using FDR [5]. The *database* is based on the structure of the specification language, see appendix A. There is only one central repository [6,7]. The tree-based description model can be seen as a view on the database. The *information message engine* generates messages appropriate for the current design state. To perform this functionality, the help engine uses the analysis part. The *user interface part* is implemented using the Microsoft Foundation Class (MFC). A snapshot of the user interface is given in figure 10.

The implementation is based on the multiple document interface (MDI). This enables the possibility of drag & drop operations between designs. It uses the document/view architecture [18]. This programming model separates a program's data from the display of that data and from most user interaction with the data. In this model, a MFC document object reads and writes data to persistent storage. A separate view object manages data display. The view obtains display data from the document and communicates back to the document any data changes. To modify and add the tree elements, the design tool comprises unique dialog boxes for every tree element type. The menu bars, pop-up menus and toolbars are configured, using the analysis part. All the options that do not meet a subset of the semantic and syntactic rules are disabled.



Training Messages
Under a parallel, prparallel and sequential construct process one can add multiple processes.

Status
Activating analysis part...
Analysis part activated successfull.
Activating training messages part...
Training messages part activated successfull.

Training Messages
A processor can consist of only one child box element and one child process element. The processor (pentium) already has a child process element, so you are not allowed to add one more. In addition, the processor (pentium) also has a child box element, so you are also not allowed to add a box element.

Status
Activating analysis part...
Analysis part activated successfull.
Activating training messages part...
Training messages part activated successfull.

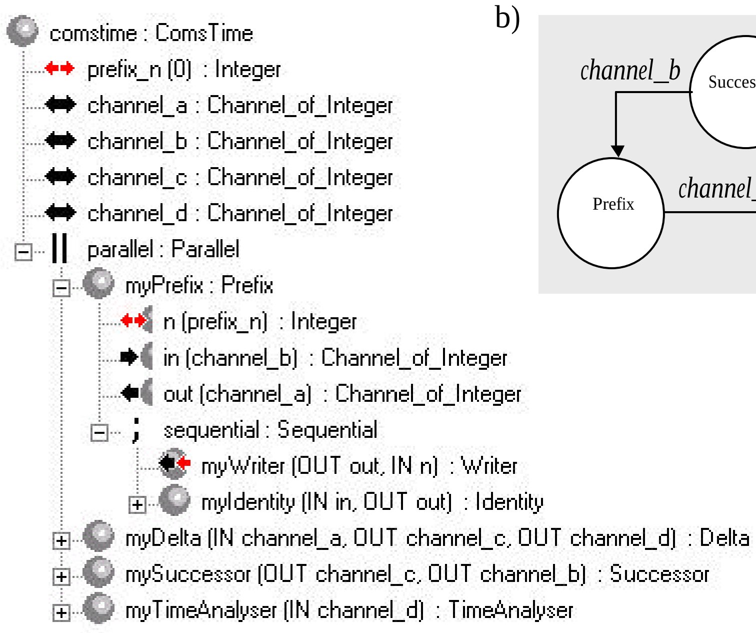
Figure 10: Snapshot of the design tool

Some general features of the tool are:

- The design tool is documented in UML using the Rhapsody software development environment [3].
- The complexity of a design can be controlled by folding and unfolding tree elements. By browsing through the tree it is easy to get an overview of the design. One is able to look at detailed information at lower hierarchical levels, while keeping an overview of abstract information at higher hierarchical levels.
- The tree-based description model enables almost a complete enforcement of the syntax. And, in relation to this, the tool is able to support the user with extra information appropriate for the specific design state. This makes the tool useful for educating CSP concepts.
- The tree-based description model allows easy reasoning about CSP designs and the syntax and semantics of the model can be explained easily.
- The prototype tool and tree-based description model is able to assist modern workbenches, like Borland Builder, Microsoft Visual C++ or 20-SIM, with coding concurrency based on CSP.
- The model can be modified in a flexible way. This can be useful when optimising for instance the throughput or reliability of a design or when one wants to make a reverse engineering tool. The browser concept in combination with the link driver concept enables a flexible mapping and user-friendly navigation between the software processes (the software dimension) and the distributed heterogeneous hardware elements (the hardware dimension). This enables concurrent engineering and reusability of both dimensions, that comes across with component-based engineering [14].
- The model is both close to the design and close to the implementation, enabling stepwise refinement. After the source code is generated, the model could be used to browse through the source code.

5. Application: simple Communication Timer benchmark

The building block library, as described in Hilderink [9], is implemented in the prototype tool. Using this building block library, a simple Communication Timer test program is designed [10]. This section describes the results.

a)


```
comstime : ComsTime
    prefix_n (0) : Integer
    channel_a : Channel_of_Integer
    channel_b : Channel_of_Integer
    channel_c : Channel_of_Integer
    channel_d : Channel_of_Integer
  ⊟ parallel : Parallel
    ⊟ myPrefix : Prefix
        n (prefix_n) : Integer
        in (channel_b) : Channel_of_Integer
        out (channel_a) : Channel_of_Integer
      ⊟ sequential : Sequential
          myWriter (OUT out, IN n) : Writer
        ⊞ myIdentity (IN in, OUT out) : Identity
    ⊞ myDelta (IN channel_a, OUT channel_c, OUT channel_d) : Delta
    ⊞ mySuccessor (OUT channel_c, OUT channel_b) : Successor
    ⊞ myTimeAnalyser (IN channel_d) : TimeAnalyser
```

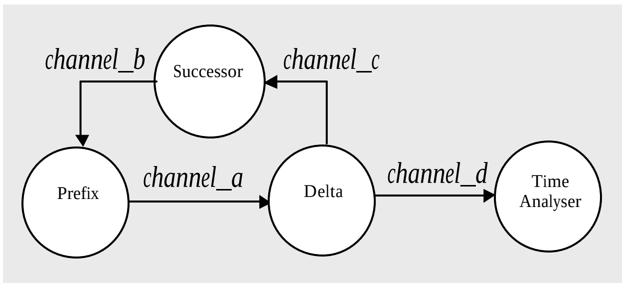
b)


Figure 11: ComsTime: (a) tree-based description model; (b) data-flow graph

The Communication Timer test program is a benchmark for measuring process (i.e. thread) context-switch time. Basically, it is an up counter that starts from zero. The tree-based description model is illustrated in figure 11a and the data-flow graph is shown in figure 11b. Note that the data-flow graph shows insufficient information to model patterns of communication and the hierarchy of the execution framework of the design. The prefix process is unfolded to illustrate its content. The benchmark consists of four processes (`Prefix`, `Delta`, `Successor`, and `TimeAnalyser`) connected by channels [10]. The `prefix` process kicks off by outputting the `prefix_n` variable on its output channel and then going into an infinite *input-and-forward* cycle. The format of the `prefix_n` variable declaration element shows that the initial value of the variable is 0. The `delta` process building block just cycles by waiting for input and then forwarding two copies (in parallel) to each of its output channels. The `Successor` process cycles through waiting for input, incrementing the number that arrives and forwarding the result to its output channel. Finally, the custom `TimeAnalyser` process simply consumes the arriving numbers (which will be the sequence of natural numbers) and times how fast they arrive.

The `Prefix` process and `Successor` process synchronise with other threads twice per cycle, the `Delta` process three times and the `TimeAnalyser` process once per cycle. This makes 8 synchronisation events (context switches) per number that is consumed by the `TimeAnalyser` process. Dividing the cycle time reported by the `TimeAnalyser` by 8 gives us the basic overhead for thread synchronisation. The time spent by the processes themselves doing other things being negligible [10]. The Java code that is generated by the tool is shown in figure 12. The `Delta`, the `Successor`, and the `TimeAnalyser` classes are omitted, only the `Prefix` class is shown.

```
public class ComsTime
{
  public static void main(String args[]) {
    int prefix_n = 0;
    Channel channel_a = new Channel();
    Channel channel_b = new Channel();
    Channel channel_c = new Channel();
    Channel channel_d = new Channel();

    Prefix Prefix = new Prefix(prefix_n,channel_b,channel_a);
    Delta myDelta = new Delta(channel_a, channel_c, channel_d);
    Successor mySuccessor = new Successor(channel_c,channel_b);
    TimeAnalyser timeranalyser = new TimeAnalyser(channel_d);
  }
}

public class Prefix implements Process
{
  private Integer n;
  private ChannelInput_of_Integer  in;
  private ChannelOutput_of_Integer out;
  private Sequential sequential;

  public Prefix(Integer n, ChannelInput_of_Integer in,
                ChannelOutput_of_Integer out) {
    this.n   = n;
    this.in  = in;
    this.out = out;

    sequential = new Sequential(new Process[] {
      new Writer(out, n),
      new Identity(in, out)
    });
  }

  public void run() {
    sequential.run();
  }
}
```

Figure 12: The automatic generated Java code

6. Conclusions

A tree-based description model with a specification language was developed. A prototype tool that uses the model was implemented. The tool comprises several syntactic and semantic rule checks and context dependent support messages. A subset of the rule checks is performed immediately, preventing the user from making errors. Several cases with the prototype tool show that it enables a user-friendly way to design CSP based software. The information messages make the tool useful for learning CSP concepts. Using the tool, the mechatronic control engineer is able to develop concurrent software, while enabling a formal proof for analysing, verifying and eliminating among others race hazards, deadlocks, livelock and starvation, resulting in reliable software.

The CSP paradigm scales well with complexity and so does the tree-based description model. One of the benefits of the tree-based description model is that it shows hierarchical aspects well. When a hierarchical level is unfolded, it is still possible to visualise for instance higher hierarchical levels. The user has complete freedom of what level of hierarchy he wants to see. This freedom is especially useful for the aspects related to the execution of processes. It enables the possibility to look at more processes at once, while retaining the possibility of visualising the execution relation between the processes. Besides folding, the overview can become clearer by opening multiple windows next to each other with each window showing a fragment of the tree one is interested in. This is particularly useful when tree fragments, one is interested in, are far apart.

The disadvantage, however, is that the tree-based description model does not show topological aspects well. Future research will be focused on developing a topology-based description model that enhances the tree-based description model, such that it is able to describe both the hierarchical aspects as well as the topological aspects well.

References

[1] Broenink, J., Hilderink, G.H., et al. (1998), *Conceptual Design for Controller software of mechatronic systems*, Computer Aided Conceptual Design,University of Twente.

[2] Broenink, J.F. (1999). "20-SIM software for hierarchical bond-graph/block-diagram models." Simulation Practice and Theory **7**: 481-492.

[3] Douglass, B.P. (1998), *Real-Time UML: developing efficient objects for embedded systems*, Addison Wesley Longman.

[4] Estier, T. About the BNF notation. Geneva, University of Geneva.

[5] FDR (1994), *FDR User Manual and Tutorial Formal Systems (Europe), Version 1.4*, Oxford.

[6] Gee, D.M., Worral, B.P., et al. (1991), *Development methods and occam*, Occam and transputer - current developments, Loughborough, UK,111- 122,16-18th September 1991,IOS Press.

[7] Gray, J.P. (1994), *PARSE-EDIT: a Parallel Program Design Tool*, Parallel Computing and Transputers PCAT-93, Brisbane, Australia,343-349,3-4th November 1993,IOS Press.

[8] Hilderink, G.H. Communicating Threads for Java (CTJ) home page, http://www.rt.el.utwente.nl/javapp.

[9] Hilderink, G.H. (1997), *Communicating Java Threads Reference Manual*, Proc. WoTUG-20 on Parallel programming and Java, Enschede, Netherlands,283-325.

[10] Hilderink, G.H. (1997). Concurrent programming for Real-Time and Embedded Systems. *Faculty of Electrical Engineering, Control Laboratory*. Enschede, Netherlands, University of Twente.

[11] Hilderink, G.H. (2000). CTC and CTCPP Reference Manual, internal document.

[12] Hilderink, G.H., Bakkers, A.W.P., et al. (2000), *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,400-407,IEEE.

[13] Hilderink, G.H. and Broenink, J.F. (2000), *Conditional Communication under the presence of Priority*, Communicating Process Architectures, Canterbury, UK.

[14] Hissam, S.A. (2000). CBS Overview, Carnegie Mellon, Software Engineering Institute. 2000, http://www.sei.cmu.edu/cbs/overview.html.

[15] Hoare, C.A.R. (1985), *Communicating Sequential Processes*, Prentice Hall.

[16] Hoare, C.A.R. (1988), *INMOS Limited Occam 2 Reference Manual*, Prentice Hall International Series in

Computer Science.

[17] Jackson, M.A. (1983), *System Development*.

[18] Microsoft (2000). Document/View Architecture Topics. 2000, http://msdn.microsoft.com/library/devprods/vs6/visualc/vccore/_core_document.2f.view_architecture_topics.htm.

[19] Moores, J. (1999), *CCSP - A Portable CSP-Based Run-Time System supporting C and occam*, WoTUG-22. Architectures, Languages and Techniques, Keele,IOS.

[20] RSRE (1986), *The Official Handbook of MASCOT*, RSRE.

[21] Scattergood, B. (1998). The Semantics and Implementation of Machine-Readable CSP. *Computing Laboratory*. Oxford,UK, Oxford University.

[22] Ward, P.T. and Mellor, S.J. (1986), *Structured Development for Real-Time Systems*, Yourdon Press.

[23] Welch, P.H. (2000), *Process Oriented Design for Java: Concurrency for All*, The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'2000, Monte Carlo Resort, Las Vegas, Nevada, USA,June 26 - 29.

[24] Welch, P.H. and Austin, P.D. The JCSP home page, http://www.cs.ukc.ac.uk/projects/ofa/jcsp.

Appendix A syntax of the specification language

```
<CSPSPEC language>    ::= CSPSPEC <heading> <body> END
<heading>             ::= <identifier> [<description>]
<body>                ::= <system>
<system>              ::= SYSTEM <identifier> [<description>] [<bios>] {<system>
                          | <processor>} END
<processor>           ::= PROCESSOR <identifier> [<description>] [<bios>]
                          <process> END
<process>             ::= <default process> | <skip process> | <stop process> |
                          <reader process> | <writer process> |
                          <timeslicer process>
<default process>     ::= CUSTOM [<identifier>]1[<description>] <class> <package>
                          [{<processinterface>}] [{<channel>}] [{<variable>}]
                          (<process> | <code>) END
<skip process>        ::= SKIP <identifier> [<description>] <class> <package>
<stop process>        ::= STOP <identifier> [<description>] <class> <package>
<reader process>      ::= READER <identifier> [<description>] <class> <package>
                          IN (<channelinput.identifier> |
                          <channelinputoutput.identifier> | <channel.identifier>
                          ) OUT ( <variable.identifier> |
                          <variableoutput.identifier> |
                          <variableinputoutput.identifier> )
<writer process>      ::= WRITER <identifier> [<description>] <class> <package>
                          OUT (<channeloutput.identifier> |
                          <channelinputoutput.identifier> | <channel.identifier>
                          ) IN ( <variable.identifier> |
                          <variableinput.identifier> |
                          <variableinputoutput.identifier> )
<timeslicer process>  ::= TIMESLICER <identifier> [<description>] <class>
                          <package> <time>
<code>                ::= CODE <identifier> [<description>] <target>
                          <codesegment>
<composition>         ::= <sequence> | <parallel> | <alternation> |
                          <priparallel> | <prialternation>
<sequence>            ::= SEQ [<identifier>]1 [<description>] <class> <package>
                          {<process>} END
```

[1] Anonymous for Java, but must be specified for C and C++.

```
<parallel>             ::= PAR [<identifier>][1] [<description>] <class> <package>
                           {<process>} END
<priparallel>          ::= PRIPAR [<identifier>][1] [<description>] <class>
                           <package> {<process>} END
<alternation>          ::= ALT [<identifier>][1] [<description>] <class> <package>
                           {<guard> | <alternation> | <prialternation>} END
<prialternation>       ::= PRIALT [<identifier>][1] [<description>] <class>
                           <package> {<guard> | <alternation> | <prialternation>}
                           END
<guard>                ::= <input guard> | <output guard> | <skip guard> |
                           <timeout guard>
<input guard>          ::= GUARDINPUT [<identifier>][1] [<description>] <class>
                           <package> ( <channelinput.identifier> |
                           <channelinputoutput.identifier> | <channel.identifier>
                           ) [<condition>] <process> END
<output guard>         ::= GUARDOUTPUT [<identifier>][1] [<description>] <class>
                           <package> ( <channeloutput.identifier> |
                           <channelinputoutput.identifier> | <channel.identifier>)
                           [<condition>] <process> END
<skip guard>           ::= GUARDSKIP [<identifier>][1] [<description>] <class>
                           <package> [<condition>] [<process>] END
<timeout guard>        ::= GUARDTIMEOUT [<identifier>][1] [<description>] <class>
                           <package> [<condition>] <time> [<process>]
<processinterface>      ::= <channelinput> | <channeloutput> |
                           <channelinputoutput> | <variableinput> |
                           <variableoutput> | <variableinputourput>
<channelinput>         ::= CHANNELINPUT <identifier> [<description>] [<type>]
                           ( <channel.identifier> | <channelinput.identifier>|
                           <channelinputoutput.identifier> )
<channeloutput>        ::= CHANNELOUTPUT <identifier> [<description>] [<type>] (
                           <channel.identifier> | <channeloutput.identifier>|
                           <channelinputoutput.identifier> )
<channelinputoutput>   ::= CHANNELIO <identifier> [<description>] [<type>]
                           <channel.identifier>
<channel>              ::= CHANNEL <identifier> [<description>] <type> LINK
                           <linkdriver.identifier>
<bios>                 ::= BIOS [<identifier>] [<description>] {<linkdriver> |
                           <bios>} END
<linkdriver>           ::= LINKDRIVER <identifier> [<description>] <class>
                           <package>
<variable>             ::= VARIABLE <identifier> [<description>] <type>
<variableinput>        ::= VARIABLEINPUT <identifier> [<description>] [<type>]
                           ( <variable.identifier> | <variableinput.identifier>|
                           <variableinputoutput.identifier> )
<variableoutput>       ::= VARIABLEOUTPUT <identifier> [<description>] [<type>]
                           (<variable.identifier> | <variableoutput.identifier> |
                           <variableinputoutput.identifier> )
<variableinputoutput>  ::= VARIABLEIO <identifier> [<description>] [<type>]
                           (<variable.identifier> | <variableinput.identifier> |
                           <variableoutput.identifier> |
                           <variableinputoutput.identifier> )
<type>                 ::= TYPE Boolean | Character | Byte | Short | Integer |
                           Long | Float | Double | Proxy | Any | <class>
<target>               ::= LANGUAGE JAVA | C++ | C | CSP | NOTSUPPORTED
<time>                 ::= TIME {digit} | <variable.identifier>
<condition>            ::= CONDITION <identifier> true | false
<identifier>           ::= <lowercase letter> { <letter> | <digit> }
<package>              ::= PACKAGE <uppercase letter> { <letter> | <digit> | "*" |
                           "." }
<class>                ::= CLASS <uppercase letter> { <letter> | <digit> }
<description>          ::= DESCRIPTION {<ASCII>} END
<codesegment>          ::= CODESEGMENT {<ASCII>}[2] CODEEND
<letter>               ::= <lowercase letter> | <uppercase letter>
<lowercase letter>     ::= "abcdefghijklmnopqrstuvwxyz"
<uppercase letter>     ::= "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
<digit>                ::= "0123456789"
```

[2] The syntax of the chosen language (Java, C++, C) is valid here