Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

SystemCSP – Visual Notation

Bojan ORLIC and Jan F. BROENINK
CTIT and Control Engineering,
Faculty of EE-Math-CS, University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands
{B.Orlic, J.F.Broenink}@utwente.nl

Abstract. This paper introduces SystemCSP – a design methodology based on a visual notation that can be mapped onto CSP expressions. SystemCSP is a graphical design specification language aimed to serve as a basis for the specification of formally verifiable component-based designs of distributed real-time systems. It aims to be a graphical formalism that covers various aspects needed for the design of distributed real-time systems in single framework.

Keywords. CSP, Formal methods, Graphical modeling

Preamble

This paper focuses on visual elements of SystemCSP notation. The paper is accompanied with the second paper[1], that puts focus on the component related part of the SystemCSP design methodology.

Introduction

CSP is a relevant parallel programming model and the design specification method introduced in this paper aims to foster its utilization in practice of component-based software development. According to [2], "CSP was designed to be a notation and theory for describing and analyzing systems where primary interest arises from the way in which different components interact".

CSP came into the world of practical software development, via the occam programming language. Our research is situated in a context where several preceding projects were dealing with ways to structure concurrency in complex control systems using occam-like approaches. One of the deliverables of previous projects is GML[3], a visual modeling language for specification of concurrent systems. GML is geared towards producing occam-like programs. It provides a lot of design freedom in early stages of design by relying on idea to relate processes via binary compositional relationships instead of starting immediately with occam-like constructs.

With SystemCSP, we attempt to make a paradigm shift from occam towards CSP. CSP offers more expressiveness for specifying concurrent systems then its occam related subset. In addition, experiences with GML lead to the conclusion that although binary relationships are useful in early stages of design, they tend to clutter readability of even relatively simple diagrams. Instead of binary relations as used in GML, SystemCSP defines visual control flow elements that cover all relevant CSP operators. Still, the benefit obtained by specifying binary relationships in early stages in design, is utilized by allowing certain set of binary

relationships (different than in GML) to be specified among components in special interaction diagrams. In SystemCSP, the same component can appear in many interaction diagrams and all specified binary relationships in such diagrams come together in a single execution diagram. More details on interaction and execution diagrams and other issues concerned with the component framework of SystemCSP is the topic of a related paper[1].

The *SystemCSP* notation is applicable for specifying, documenting, visualizing and formal verification of component-based designs. This includes design of both processes and components. SystemCSP provides a way to visualize architecture, behavioral patterns of components, intra–component interactions and execution relations among components. The notation makes a distinction between components, interaction contracts and processes. CSP is focused on the interaction between processes. A process is viewed as a behavior described via some named pattern of event synchronizations. The term *component* is used as in modern notions of software development: "A *component* is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition" [4]. From a CSP point of view, the behavior of a component is captured as a complex process, described with the help of one or more auxiliary processes. An *interaction contract* is a process or a component that has the responsibility to manage interaction of the involved components.

SystemCSP is based on the principles of both component-based design and CSP process algebra. Such a combination promises to offer a more structured approach and more expressiveness than the one offered by the occam-like approach targeted in GML.

Graphical elements introduced in SystemCSP are related to basic elements of the CSP process algebra. In this way, designs have immediate mapping to CSP expressions. Any CSP description can be mapped to an appropriate graphical representation in SystemCSP.

This paper introduces all graphical elements used in SystemCSP. The related companion paper[1] puts more focus on component-based aspects of software development in SystemCSP.

The first section deals with state of the art in software design with special focus on specifying concurrency and especially CSP-based approaches. The second section introduces elements of SystmCSP that visualize CSP language elements. The third section introduces elements of SystemCSP that are relevant part of the notation, but are not directly related to CSP concepts. The last section attempts to make a comparison between SystemCSP and some other approaches used for visualization of software models.

1. State of the Art in Visualizing Concurrent Systems – Focus on CSP Based Systems

Humans can much better comprehend and communicate visualized behavioural scenarios then the same scenarios given via some mathematical description, e.g. CSP formulas. Still mathematical descriptions are necessary for precise analysis of models. A workaround to using mathematical descriptions directly in software design is to introduce a set of intuitive visual elements that can be automatically mapped onto mathematical descriptions.

1.1 Finite State Machines

In CSP related books[2, 5], often *(state) transition diagrams* are used to illustrate CSP interaction patterns. Those are in fact Finite State Machine (FSM) kind of diagrams. Every node in such FSM represents a state of the process and every edge/transition is associated with some event. Figure 1 represents one CSP description and its associated visualization based on a FSM. In fact, the FSM in Figure 1 is a typical UML-like visual representation of a state machine. State transition diagrams in CSP books differ from UML statecharts in

depicting states as small circles with state names written outside of state circle. In addition, start and exit states bare no special visual difference to other states. The difference is of course that there are no transitions leading to the start state and no transitions leading from the exit state.

One can thus visualize some CSP descriptions using transition diagrams. Looking in another direction, CSP expressions can be seen as an attempt to capture visual FSM specifications in the form of a language-like sequential stream of characters. In order to transform a FSM into CSP expressions, some states are given names and considered to be named CSP processes. Note that the same FSM can be mapped to different CSP descriptions depending on the choice of the states to which names are assigned. However, a CSP representation of a FSM, based on the minimal number of process names is unique (of course, provided that one abstracts away from differences in chosen names). Such a representation is obtained if only the start state and states reachable from more than one other state (states where a join of several control flows is performed) are named. In Figure 1, states 3 and 6 are named respectively Temp1 and Temp2, defining in such a way auxiliary processes needed as recursion entry points in the CSP description.

```
        P1= ev1?x → Temp1 |  ev2!y → ev4!z → Temp2
Temp1 = ev3!x → ( ev7?y → SKIP  |  ev8!y →  ev6?x → Temp2)
                  Temp2 = ev5!x → Temp1
```

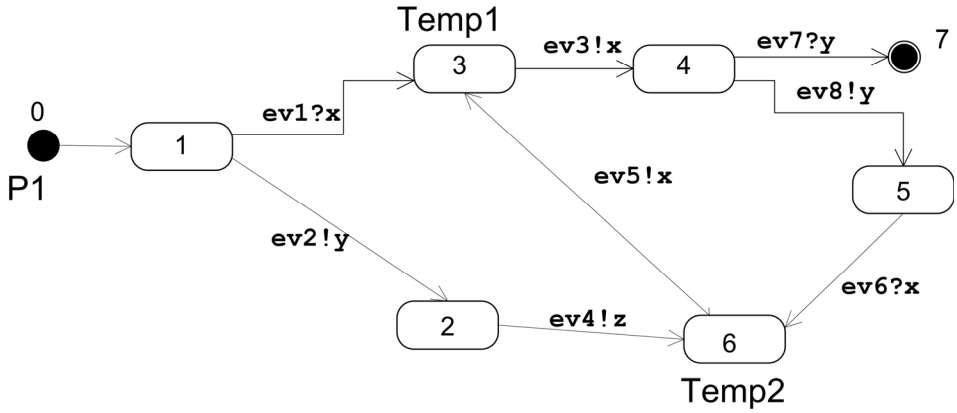

Figure 1. Classical FSM diagram

An obvious question here is why invent something new, if one can use FSM diagrams as they are for specifying CSP processes. If one aims to capture only simple CSP processes that contain only guarded alternatives, prefix operators and event occurrences, then a FSM is good enough abstraction. Those diagrams are however not used to depict examples containing for example parallel, internal choice, hiding and renaming operators. Other issues are that different diagrams can be drawn for the same process (depending on whether recursion is expanded and how many times), and that processes with infinite number of states are impossible to draw.

Processes composed in parallel are sometimes [5] represented as rectangle boxes adorned with ports representing events in the alphabet of the process. Such process boxes are then related via lines that connect ports that synchronize.

One of the approaches to a structured way of specifying concurrency derived from CCS and CSP process algebras and using state transition diagrams is FSP[6]. FSP provides a Java library implementation and a tool for model animation and checking. Compared to CSP books, this approach goes one step further in visualizing process expressions. The

initial (*start*) state is shaded. The letter E inside a state circle denotes the *end* state. A sequential composition is created by concatenating two finite state machines: making a transition from the end of a subprocess to the start of the next one in line and hiding start and end states resolved in this way. A parallel composition is also visualized via creating an equivalent state machine or alternatively again as connecting appropriate ports of boxes representing subprocesses. In addition, the notation allows systematic adding of prefixes to the names of event transitions by altering the process labels associated with states. Hiding is performed by replacing the event name with the keyword `tau` and reducing the state machine by merging states related by the `tau` labeled transitions. Renaming is performed by making a new state machine with names changed according to the replacing function. Time is introduced using `tick` events.

1.2 UML

In software development practice, UML is the de-facto standard for visualizing software design processes.

UML is designed to offer general support that can be used in various development processes [7]. UML contains several kinds of diagrams based on different basic elements and there is no mapping or firm relation between those separate views in which the same entities can participate. UML does not have precise semantics, and as such it is most useful in early stages of design for informal communication. The lack of precise semantics makes complete consistency checks between different diagrams impossible [8]. Informal way of using, based on local conventions, is often creating problems in communicating designs between stakeholders. Another significant problem with UML diagrams is that they are not able to capture in clear and intuitive way the concurrency structure of a program.

According to the survey [9] on UML usage in practice: adherence to standards is loose, there is no objective criteria to verify that a model is complete or satisfies some tangible notion of quality, miscommunication is reported in more than half of the projects, "wrong" product delivered is mentioned, high amount of testing effort is needed. According to this survey, some of the main problems with UML are: design choices scattered in unrelated views, informal use, limited possibility for checking consistency between different views, disproportion between specified architectural details and the needs of implementers…

In UML designs, precise semantics necessary for executable specifications can be obtained only if UML is combined with some formal language (e.g. SDL). Approaches based on a combination of UML and CSP also exist. Crichton and Davis [10] actually proposed a design pattern for specifying concurrency patterns using a subset of UML, in a way that is formally verifiable via CSP. Their approach is based more or less on the combination of statecharts and activity diagrams. But such an approach is fitting existing diagrams into something they were not designed for and it also suffers from not allowing full expressiveness of CSP. The aim of that research was to create formally verifiable concurrency design pattern using existing UML diagrams. For our purpose insisting on usage of UML diagrams is not an issue.

1.3 GML

In previous research at our Lab, GML[3, 11] was developed. In GML, the recommended design process starts with process blocks existing in isolation. The first design step is making the communication structure as a standard data flow model. Next, the concurrency structure is added to this model by specifying binary compositional relationships such as sequential, (pri)parallel and (pri)alternative, between involved processes. Some relationships are known in advance and some are subject to various trade-offs. For instance,

instead of specifying immediately that there is parallel composition of processes A, B and C, one might first conclude that processes A and B should be executed in parallel. Only after the same type of relationship is made between for instance B and C, it becomes possible to group A, B and C to one parallel construct. If binary relationships are specified between any two processes, and provided such a complete specification is not illegal, then it implicitly defines grouping of processes into a tree-hierarchy of constructs. Normally, grouping is explicitly defined by using explicit grouping symbols like surrounding rectangles (*box notation*) or indexed bubbles (*parenthesis notation*) on the ends of binary relationships. *Indexed bubbles* at the ends of the relationships are used in a way similar to the way parentheses are used in CSP expressions. A bubble is always placed on the side of the compositional relationship next to the process that is considered to be inside the parenthesis. The index of a bubble is the number of parentheses used on that place. The CSP expression P;(T;(Q || R)) is in GML specified in one of the two ways (parenthesis and box notation) visualized in Figure 2.

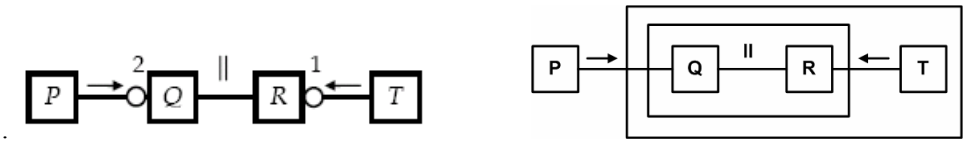

Figure 2. GML models (parenthesis (adopted from [12]) and box notations)

Unfortunately, both grouping notations restrict designs to a single diagram representing the model. This diagram can be split in several views by collapsing/hiding parts of the hierarchy in separate views. It is however not possible that same component participates in several views focused on different aspects of the system, as is the case in UML models.

Another disadvantage of the used grouping notations (especially the preferred parenthesis notation) is that for untrained eye it is quite difficult to spot borders of the constructs and to reconstruct exact control flow, especially in more complex examples.

When a GML model is complete, it is always a tree-like hierarchy of constructs and wrapper processes as branches and user-defined processes as leaves. The resulting executable is always occam-like.

GML is a design methodology and visual notation related to occam. As occam, it fails to use the full expressiveness of CSP. Most notable example is the inability to visualize designs shaped as finite state machine–like diagrams in a clear and intuitive way. The main difference compared to simple visualization of occam-like hierarchies is that design freedom is enhanced by defining constructs as groups of binary relationships between processes. Compared to constructing an occam-like application as a hierarchical-tree, GML models seem to offer more flexibility as a design entering view, especially in early stages of the design, when exact borders of the constructs are not yet quite clear. GML models can express designs that are illegal or underspecified or ambiguous. Additional rules exist to check consistency of designs before translation to CSP is possible.

One of the advantages of GML is that refining the data flow model with compositional relationships expressing the concurrency structure can be done without changing the 2D layout of original data-flow model. This feature makes a prospective tool based on GML suitable for application in a chain of tools, with the preceding tool in the chain producing data flow model e.g. based on some application-specific domain.

2. Visualizing CSP Process Expressions in SystemCSP

In SystemCSP, processes are visualized using diagrams that contain basic process elements, lines representing synchronization/communication events, process labels and various control flow elements.

Processes or process parts can be visualized via transparent rectangles (transparent-box approach) exposing their internals or via solid filled rectangles hiding internals of processes (black-box approach).

2.1 Basic Process Elements

Basic process elements are: START, EXIT, STOP, Tau, EventSync, Writer, Reader, and EventAccept (Figure 3).

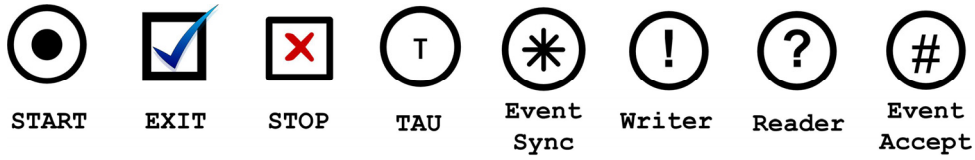

Figure 3. Basic process elements

The existence of a *start* event is implicitly assumed in CSP, but is not written down in CSP expressions. In the visual notation, however, it is very useful to mark the entry point of a process or a component. The START basic process element is marking the entry point of a component. The EXIT process is a point of successful termination (equivalent to the SKIP process in CSP). STOP is, as in CSP, the process that does not engage in any event. Note that the EXIT process is a compound process that communicates a successful termination event to the parent operator and then behaves as STOP. As in CSP, the Tau process represents an event internal to the component, and as such it is never offered to the environment.

EventSync is an elementary process participating in event synchronization with one or more peer *EventSync* elementary processes executing in parallel. An event takes place when all participating *EventSync* processes are ready (*rendezvous synchronization*). Peer *EventSync* elements, participating in the occurrence of the same event, are connected via a dashed line.

An *EventSync* process can in general initiate or accept events. The difference is not important from the CSP point of view, but sometimes in designs, it is handy to know which side initiates the interaction. For a side that can only accept interaction, the *EventAccept* symbol is used. The two other special kinds of *EventSync* symbols are *Writer* and *Reader* symbols that emphasize the direction of a unidirectional communication associated with event occurrence. The Writer and Reader basic processes are alike to channels in the occam approach.

EventSync processes usually have associated an *event label* that specifies the event name and the details of the related data communication if any. In case when the data communication is present, the event label contains, in addition to the event/port name, the description of data communication. Data communication is described via "?" and "!" signs, representing the direction of communication, and the associated names of the local destination or source variables. As source of data, an expression involving multiple variables or a function that evaluates to a value of an appropriate data type can be used. One event occurrence can have multiple data communications associated.

Basic processes belonging to the same component can be interconnected via control flow elements based on CSP operators.

2.2 Prefix - based Control Flow Elements

As prefix operator of CSP, the *prefix* control flow element is shaped as an arrow (see Figure 4). As in CSP, it introduces a constraint into event ordering by specifying the sequential order between an event and the rest of the process executed afterwards. The prefix control flow element can lead to another basic process (event), to another control flow element (with the exception of a prefix operator) or to a component.

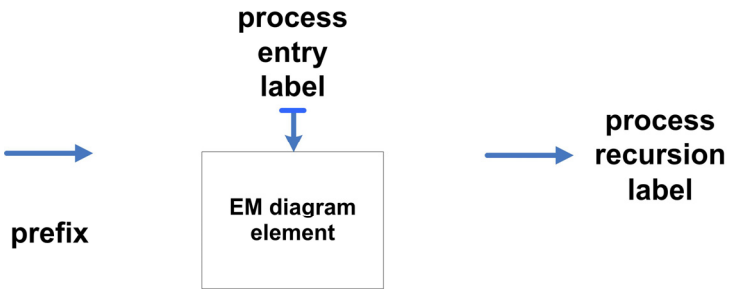

Figure 4. Prefix and related control flow elements

Process labels (Figure 4) are used to visualize process entry and recursion points. Thus, a distinction is made between a *process entry label* and a *process recursion label*. A *process entry label* represents the entry point of a process and is attached via a prefix operator to an element of an SystemCSP diagram (with the exception of a prefix operator). A prefix leading to a *process recursion label* means that after the prefix operator, the process will continue behaving in a way as defined by the process entry label carrying the same name. The combination of process entry labels and process recursion labels allows natural visualization of recursions.

Instead of using process recursion labels one can directly draw prefix arrows to entry points of appropriate processes (provided they are in the same view). However, using recursion labels makes diagrams more readable.

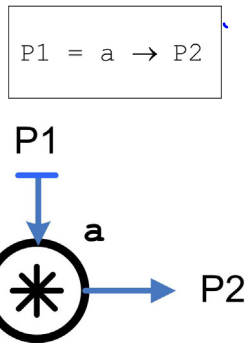

Figure 5. Combining prefix operator and process labels

In, Figure 5, the prefix leads to the description of another process: process P1 will perform event `a` and then behave as process P2.

2.3 Non-interacting Processes

Non-interacting processes are processes that do not interact with their environment. Internally, however they can contain any number of interacting subprocesses and *EventSync* processes. A special kind of non-interacting processes is a *computation process*, which contains only pure computation code.

A non-interacting process is not relevant for the CSP model on the level of abstraction where it is invoked and can thus be omitted in the resulting CSP description. Non-interacting processes are in SystemCSP specified in one of the ways depicted in Figure 6. The first three symbols provide non-interacting process descriptions inside a dedicated rectangle box. The types of description illustrated in Figure 6 are respectively: a textual description or the name of the action or the scenario it represents, a detailed description of internals, or a sequence of functions invoked. Using a brief textual description is especially useful in early stages of the design process, e.g. while specifying use-case scenarios (see Figure 6 for example). The last example illustrates that in case of computation process, it is allowed to skip the box element, and associate the description directly with the prefix operator. This is for instance convenient in some cases where the computation process is not so relevant for understanding the diagrams or in order to reduce number of displayed blocks.

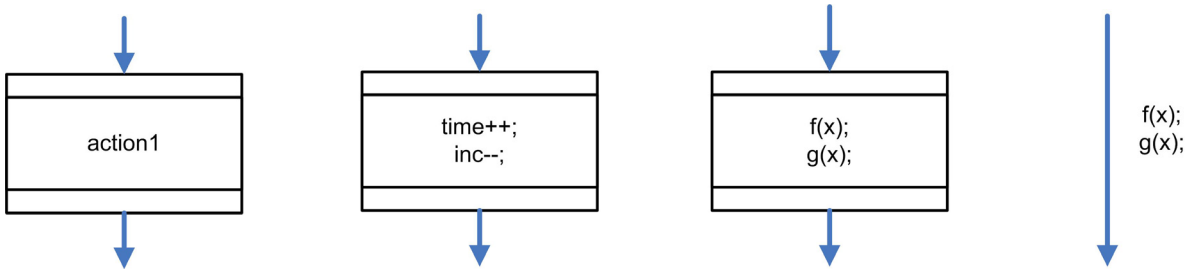

Figure 6. Non-interacting processes

Figure 7 illustrates combining previously introduced elements to describe three processes that synchronize on certain events. Process entry points are marked with *process entry labels* carrying names `P1`, `P2` and `P3`.

Note that in case of event `ev1`, either process `P1` or process `P2` can initiate the interaction and that when both processes are ready to perform the event (rendezvous synchronization), the associated data communication will take place in both directions. The value of variable w of process `P2` will be written into the variable `x` of the process `P1` and the variable `y` from the process `P1` will be written into the variable `z` of the process `P2`. Event `activate_P3` is initiated by the process `P1` and accepted by the process `P3`. This event has no associated data communication. In the third interaction, focus is on emphasizing the direction of unidirectional data communication. Therefore, the basic processes *Writer* and *Reader* are used. Rendezvous synchronization is implied and it is not considered important which side initiates the interaction.

In Figure 7, note the distinction between *dashed lines* representing synchronization points that connect peer *EventSync* processes and *solid, directed lines* used for prefix control flow elements. Such a choice between dashed and full lines makes sense because while the *EventSync* processes are points of discontinuity where waiting for the synchronization with environment is done, the flow of control in prefix lines takes place without delay.

```
P1 = ev1?x!y → activate_P3 → SKIP
P2 = ev1!w?z → ev2?q → SKIP
P3= activate_P3 → ev2!r → SKIP
```

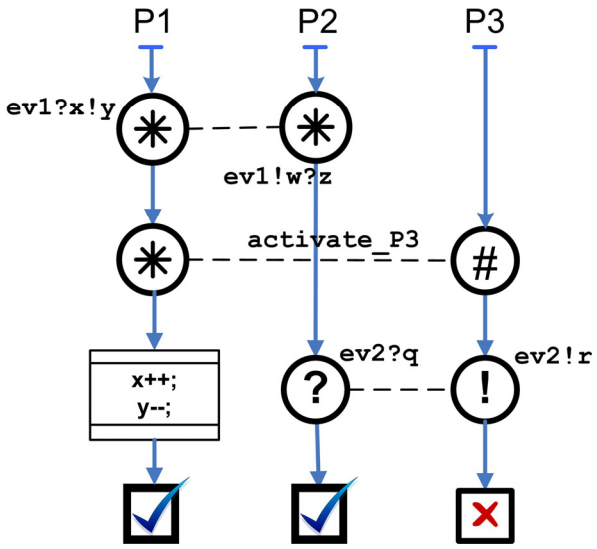

Figure 7. Combining basic processes with prefix control flow elements

2.4 Hiding and Renaming Operators

The hiding and renaming operators are applied on a process and the result is again a process. For this reason, in SystemCSP those two operators are visualized as thin-bordered rectangle elements that relate the process label of the resulting process with the entry point of the process used as the operand.

The environment of some process does not know its CSP description; it sees only the set of offered (ready) events. In CSP, one can apply the *hiding operator* on a process, resulting in hiding the chosen set of events from the environment and making them internal to the process. In SystemCSP, the hiding operator is specifying the set of hidden events after the "\" symbol.

In addition, in a CSP process expression, it is possible to replace all occurrences of event/process names or event/process expressions with some other event/process names or event/process expressions (*renaming operator*). The symbol used in SystemCSP for renaming operator relies on notation that resembles to multiplying with the ratio of the new and old name (expression). This is an intuitively clear way to create the illusion of canceling the old expression and replacing it with the new one.

The hiding and renaming operators are illustrated in Figure 8. The renaming element specifies that events `on` and `off` are renamed into `light_on` and `light_off`. The process created in this way is named ELECTRIC_LIGHT_SWITCH. In same example the hiding operator is applied on same SWITCH process in order to hide event `off` from some users. The process name under which such users can see SWITCH process is TURN_ON. This process will offer to its environment either event `on` or no event.

```
                    SWITCH= on → off → SWITCH
                     TURN_ON = SWITCH \ {off}
ELECTRIC_LIGHT_SWITCH = light_on → light_off → ELECTRIC_LIGHT_SWITCH
```

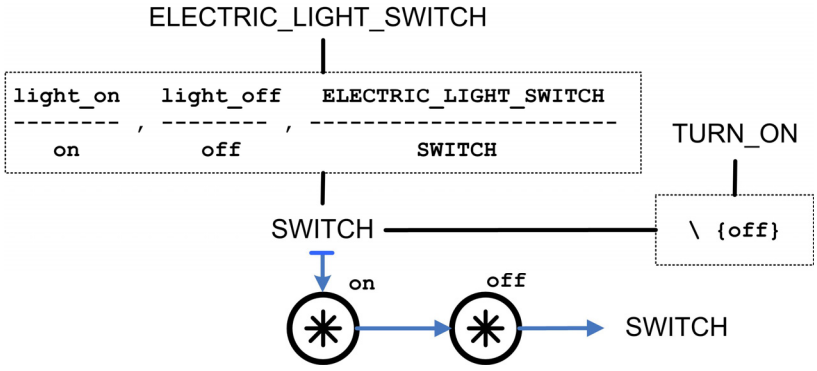

Figure 8. Example illustrating the usage of the renaming operator

2.5 IF Choice and Guarded Alternative

An IF element (see Figure 9) specifies, inside square brackets, a Boolean expression that represents a condition. It has two *prefix* arrows leading from it: the TRUE and the FALSE paths. Generalization of the IF control flow element is the SWITCH control flow element. Its symbol is as the symbol for an IF element, but it can have more then two outgoing prefix control flow elements, each one with a different constant value associated. In CSP, the SWITCH element can always be represented by several nested IF choice operators.

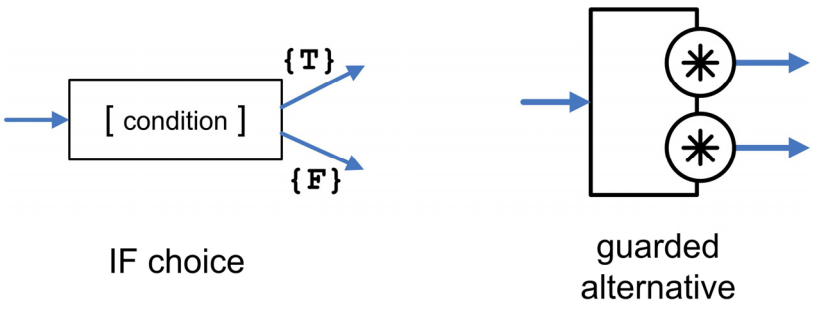

Figure 9. IF conditional choice and guarded alternative choice

The guarded alternative is a process that offers to its relevant environment, a choice between several events. The branch starting with the chosen event will be followed. In SystemCSP, this element (see Figure 9) is depicted as a rectangle in which SyncEvent processes are half-emerged. From the outer side of the SyncEvent circles, prefix control flow elements lead to communication patterns representing the alternative control flow branches.

Figure 10 illustrates the usage of an IF construct for specifying a *recovery block* fault tolerance mechanism. The essence of the mechanism is providing several implementations of the same functionality, possibly with different QoS (Quality of Service) levels. If the results obtained by executing block F(x) fail to pass the acceptance test, then the block G(x) is performed, and so on. In [12], a distributed recovery block mechanism is described in CSP. The example here is a visualization of a non-distributed version. If all recovery blocks

fail associated acceptance tests, then the `error` event is used to notify the client that the recovery block has failed to provide the required service.

```
Recovery block = input?x →
                  (Success ≮ acc.test 1 ≯
                        (Success ≮ acc.test 2 ≯ error!status → SKIP))
Success = result!res → SKIP
```

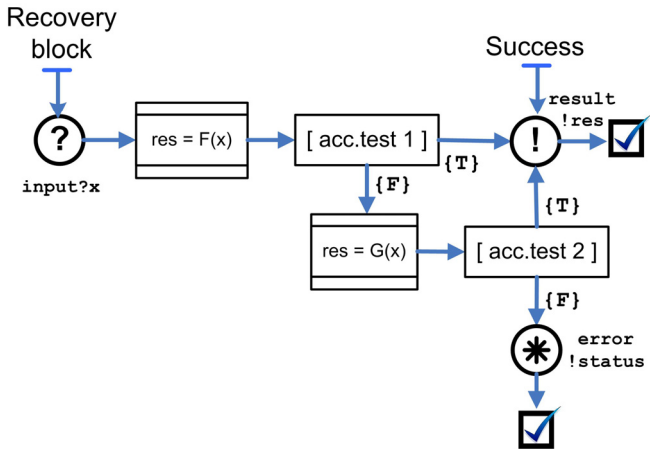

Figure 10. Example with IF conditional branching element

Figure 11 illustrates the use of parameterized process labels and the use of logical conditions associated with a *guarded alternative* on the example of a non-negative counter. A counter is specified as an array of parameterized `COUNT` processes with parameter `i` being used as the counter value. After accepting the `inc` event, the `COUNT(i)` process will behave as `COUNT(i+1)`. After accepting the `dec` event, it behaves as the `COUNT(i-1)` process. The event `dec is` guarded with the logical condition set to allow the counter value to be decremented only if the value of `i` is greater than zero.

```
Count(i)= inc!(i+1) → Count(i+1)
                    |
          ((i>0)& dec!(i-1)) → Count(i-1)
```

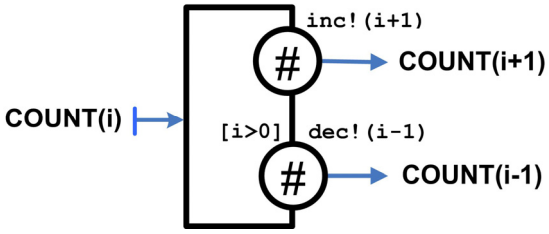

Figure 11. Parameterized counter process

Figure 12 illustrates the SystemCSP visualization of a CSP example, which uses a *guarded alternative* control flow element. The example in Figure 12 also illustrates how the hiding operator is used to hide `install/uninstall` events from the program user that has restricted access rights. When the process `Program` is at its entry point, it is ready to

be installed; those users that see the program only via `Restricted_program_use` cannot engage in any interaction with program. After program is installed by some process from the environment that can see the process under name `Program` and thus can initiate `install` event, the user can use `Start_Menu` to open the program. A user that has no access restrictions can at this point also decide to uninstall the program (`uninstall` event). If the program is opened (`openProg` event), then it can be used (`UseProg` menu). Using the program initially offers two options: closing the program (`closeProg` event) or opening some document (`openDoc` event). Upon opening a document, one can work with it (`Work` menu). Working with the document includes making choices between several actions: updating the document (`updateDoc` event) saving the document (`saveDoc` event), closing the document (`closeDoc` event) or closing the entire program (`closeProg` event). Note that in Figure 12, the assumption is that the process selected for showing in the figure is `Restricted_program_use` and that because of that, the events `install` and `uninstall` are shaded.

```
Restricted_program_use = Program \ {install, uninstall}
Program = install  → StartMenu
StartMenu = openProg  → UseProg | uninstall  → Program
UseProg = closeProg  → StartMenu | openDoc  → Work
Work = UpdateDoc  → Work | saveDoc  → Work
                | closeDoc  → UseProg | closeProg  → StartMenu
```

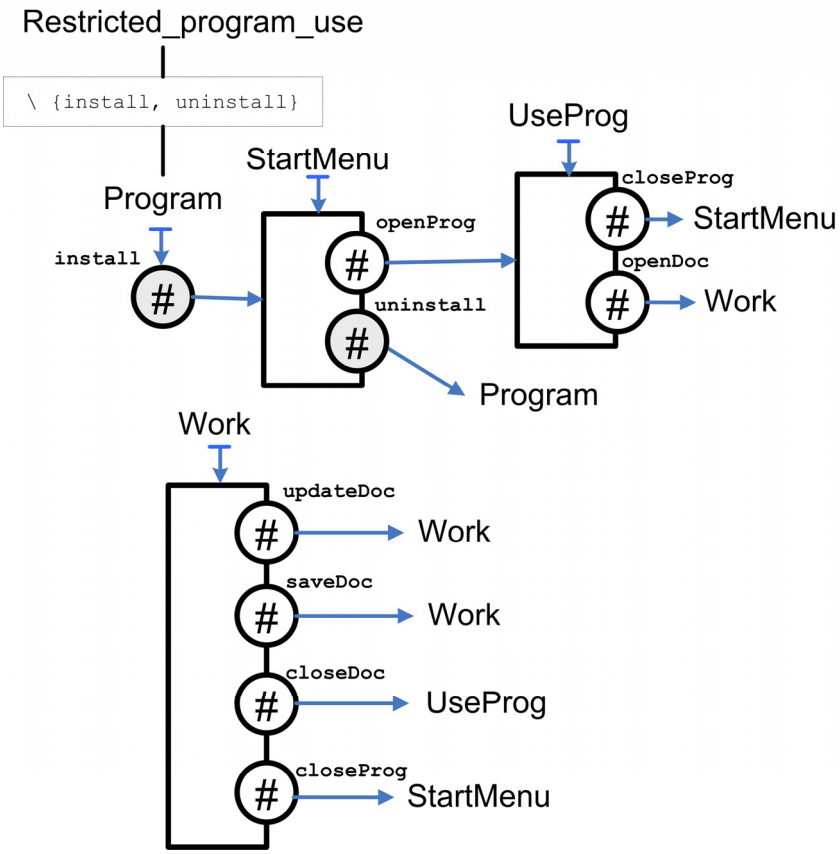

Figure 12. Example illustrating guarded alternative operator

2.6 Start and Exit Control Flow Elements

In CSP, operators like sequential, parallel, external choice and internal choice are used to combine two or more processes into a new process. In SystemCSP, control flow elements are introduced to represent those operators.

The operators and theirs operands are in CSP grouped via parentheses. Instead of using explicit grouping symbols (like parenthesis bubbles or boxes in GML), SystemCSP chooses to merge the START and EXIT events of the composition with CSP operators, creating in that way an extended set of control flow elements as illustrated in Figure 13. Every process composed via one of the CSP operators has an implicit START event and either a termination (EXIT) event or a process recursion label leading to the entry point of some other process. In a sequential combination of processes, the EXIT of one process is triggering the START of the next one in line. START event of a sequential composition corresponds to the START event of the first element in sequence and the EXIT event of the composition corresponds to the EXIT event of the last subprocess in sequence. In case of Parallel and Choice operators, a START event is a point of forking control flow to branches and an EXIT event is a point of joining control flows of branches. In a parallel combination, all involved processes synchronize on both START and EXIT events. In case of a choice, only one branch is executed.

Thus, the pair of open and close parentheses, bounding the scope of a CSP operator, actually corresponds to the pair of control flow elements representing synchronization on START and EXIT events.

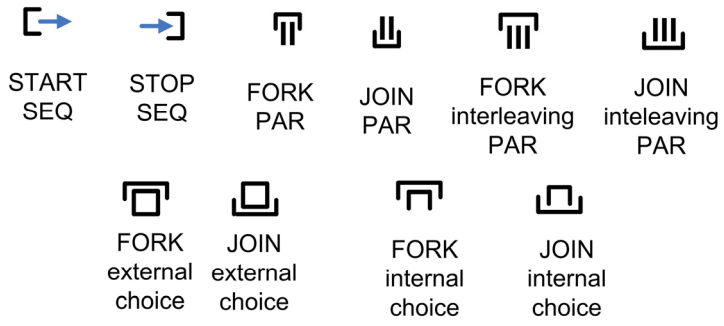

Figure 13. START and EXIT grouping symbols

A special kind of a START grouping symbol is the FORK symbol that branches control flow on two or more branches. A special kind of EXIT grouping symbol is the JOIN symbol, where control flow branches are joined. Often, but not always a FORK element is paired with an appropriate JOIN element.

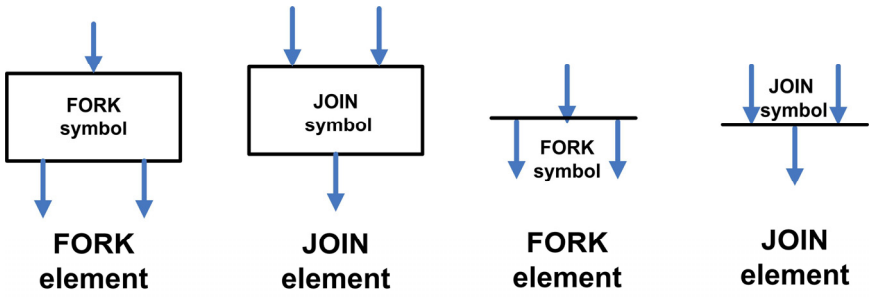

Figure 14. FORK and JOIN control flow elements

FORK and JOIN symbols are specified in association with branching/joining control flow elements in one of the two styles depicted in Figure 14. The look based on rectangles is more convenient when additional details need to be specified (e.g. synchronizing alphabets related to a Parallel operator). The look based on lines is more convenient to produce a UML-like activity-diagram look-and-feel.

One or more SEQ START and SEQ EXIT symbols can be associated with a single prefix control flow element. Comparing the CSP expression and its SystemCSP representation in Figure 15, one can see that all brackets used to group CSP operators with operands are present in symbols associated with control flow elements. Process P1 will perform event ev1 and then behave as a sequence of process P and a process constructed as a sequence of process T and parallel composition of processes Q and R.

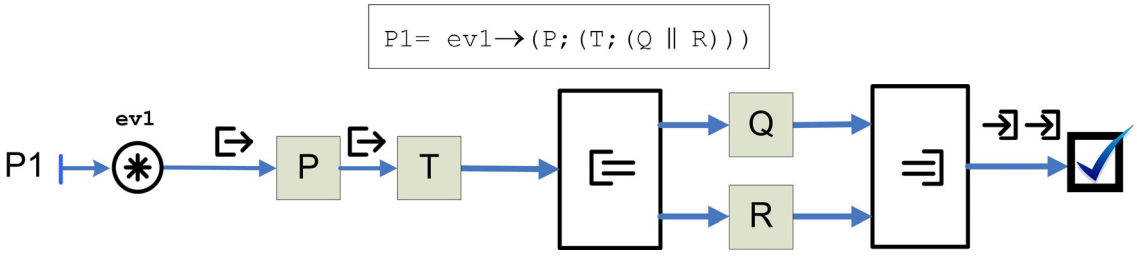

Figure 15. Grouping in SystemCSP

In the example in Figure 16, the FORK interleaving PAR is not paired with a JOIN interleaving PAR. In this example, the message forwarder process P needs always to be ready to accept new messages and can send received message at some later moment of time. Therefore, after receiving a message it spawns a copy of itself and puts it in a parallel with process Out(x) that is forwarding the already received message. This example requires in fact *dynamic process creation*, because in each recursion a fresh copy of process P needs to be created. This recursion is the reason why the black-box notation is used and not a process recursion label. A JOIN element can be drawn, but in reality, it will never be reached because new instances of P will always be created.

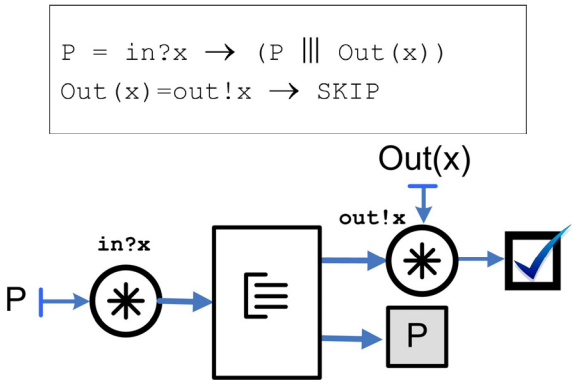

Figure 16. Message forwarder as an example for dynamic process creation

The same non-negative counter example of Figure 11 is implemented in Figure 17, using a FORK choice element instead of a guarded alternative element. The only difference compared to the guarded alternative based design, is that events offered to the environment (`inc` and `dec` events) are not considered to be a part of the control flow element but instead are considered part of branches to which the FORK choice control flow elements lead. The guarded alternative of CSP is thus in fact a special kind of external choice

operator and can always be replaced with a FORK external choice. The opposite is not the case. However, a guarded alternative is more convenient way of specifying finite state machine – like designs.

```
Count(i) = P1 □ P2
P1 = (i>0)dec!(i-1) → Count(i-1)
P2 = inc!(i+1) → Count(i+1)
```

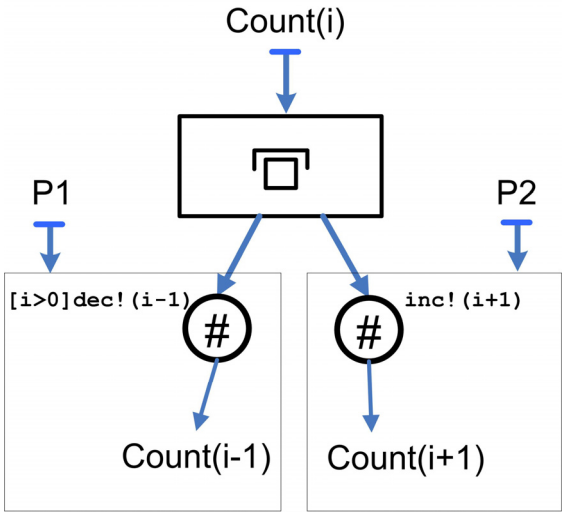

Figure 17. Counter example using FORK choice

Figure 18 represents a complex process named `Race` that contains both PAR and CHOICE control flow elements. `Race` is a parallel combination of two processes: `RaceCtrl` managing the race and `Runners` being a parallel composition of two runners participating in the race. Both runners are described via the same process description specifying that they will engage in the `start` event, and in the events `100m`, `200m` and `finish`. The parallel control flow element named `Runners` specifies that its subprocesses - two runners - actually synchronize only on the `start` event. This `Start` event is initiated by the race control mechanism (`RaceCtrl` process). The race control mechanism will on request of the runners deliver them the current time on the milestone events `100m`, `200m` and `finish`. When both runners have engaged in the event `finish`, the `count` variable becomes equal to the number of runners and the `Race` process is finished.

```
Race   = Runners ‖ RaceCtrl
Runners= Runner1 {start} ‖ {start} Runner2
Runner1 = start → 100m?time11 → 200m?time12 → finish!time13 → SKIP
Runner2 = start → 100m?time21 → 200m?time22 → finish!time23 → SKIP
RaceCtrl = start → Milestones
Milestones = (100m!time → SKIP)
                    □ (200m!time → SKIP)
                          □ (finish!(time, count++)→ SKIP)
              →(SKIP ≮ count == #Runners ≯ Milestones)
```

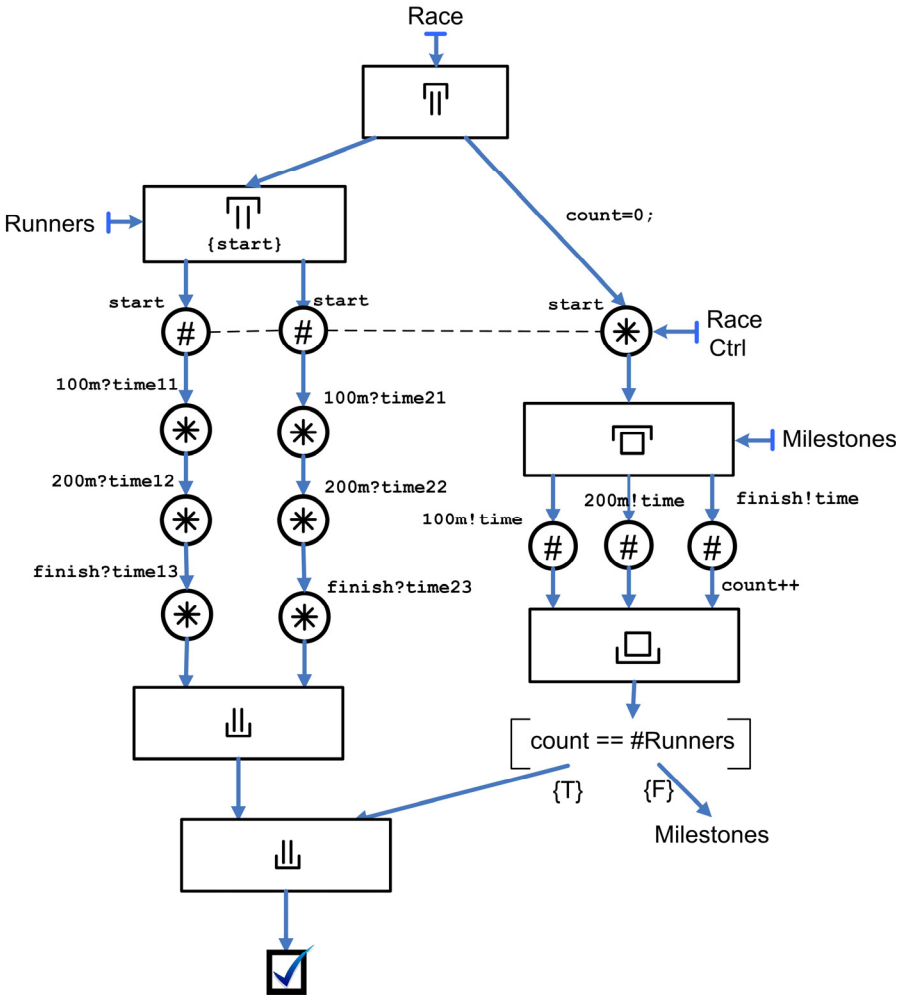

Figure 18. Example combining Par and Choice elements

The *internal choice* (symbol given in Figure 13) is different from the external choice in the sense that it will first internally make a choice and then offer only the chosen branch to the environment. The internal choice is not so useful for implementation, but it represents a powerful abstraction mechanism; It is used when one needs to specify that some process will in some, still unknown way choose one of several branches.

2.7 Exception Handling Layer

Handling exceptional situations that occur during the execution of a process is an important issue for well-designed programs. In the software development world, the implicit agreement exists that in some way the part of code/design, which specifies/implements handling exceptional situations, should be visually isolated as far as possible from the design/code specifying normal execution. In general, exceptional situations in some process can be handled by attempting recovery within that process (recovery model) or by aborting the process and executing some other process instead (termination model).

CSP defines the *interrupt operator* that covers the termination model. Process P Δ_i Q is a process that behaves as process P until either P terminates successfully or until an event occurs that activates process Q. In the latter case, further execution of process P is aborted and process Q is executed instead. Despite its name, the semantics of the Δ operator is much closer to the termination model of exception handling than to *interrupt handling*, since it implies termination of the left hand-side operand.

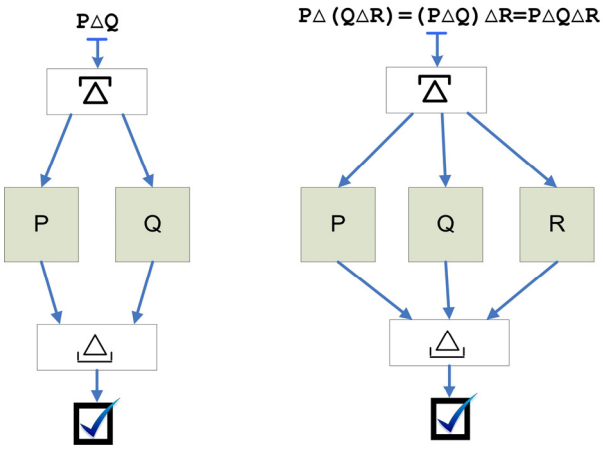

Figure 19. Take-over operator

SystemCSP adopts the interrupt operator, but under the name *take-over* operator. In Figure 19, in the first case the process Q can take over process P. The second example in Figure 19, illustrates the fact that the take-over interrupt is associative by definition. Process Q can take over process P. Process R can take over both P and Q.

2.8 FSM-like Diagrams in SystemCSP

If only events, prefix and guarded alternative elements are used, any CSP process can be mapped to both SystemCSP diagram and FSM and easily converted from one to another. In the example below, a one-to-one mapping between the SystemCSP diagram in Figure 20 and the FSM diagram in Figure 1 is obvious. The states of the FSM map to *waiting done on events* and *guarded alternative* elements in the Figure 20. This is illustrated by enumerating states in Figure 1 and the corresponding *EventSync* and guarded alternative elements in Figure 20 with numbers 0 to 7.

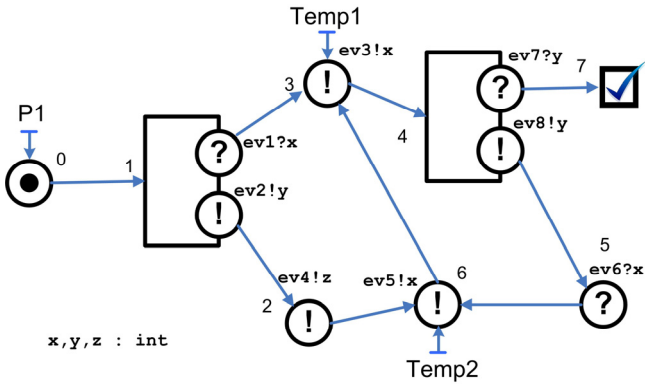

Figure 20. SystemCSP diagram

The diagram in Figure 20 made using a subset of SystemCSP has similarities to the FSM diagram of Figure 1. The main paradigm shift is emphasizing events instead of states. Events are not shown as transitions (as in the FSM), but as *EventSync* processes, which allow for direct line connections to peers in the environment or to ports of the component. Mapping from SystemCSP to CSP is more direct then it is the case for a FSM, where such a mapping requires distinguishing between states with *exactly one* outgoing transition and states where more then one outgoing transition is present.

SystemCSP makes difference between internal and external choice, while FSMs imply external choice.

3. Non-CSP Elements of Notation

3.1 Representing Sequential Designs

Computation processes are allowed to specify arbitrary complex sequential OOP designs. In principle they can be designed using UML or some other type of diagrams. We however aim to provide visual programming covering all parts of the design. For specifying computation processes, special diagrams inspired by UML sequence diagrams are proposed.

In addition to elements from UML sequence diagrams, the diagrams representing sequential designs in SystemCSP introduce notation elements for specifying grouping statements in blocks, conditional branching and loops. Conditional branching is represented by a condition specified in square brackets. At any moment, for every condition either the `true` or `false` branch is depicted. The borders of the control flow blocks are visualized by means of large square brackets at the most left-hand side of the diagram (see Figure 21). Every function that is specified via a sequential design diagram is expected to have defined properties like: description of service that it provides, input/output parameters, preconditions, postconditions, and exceptions it may raise.

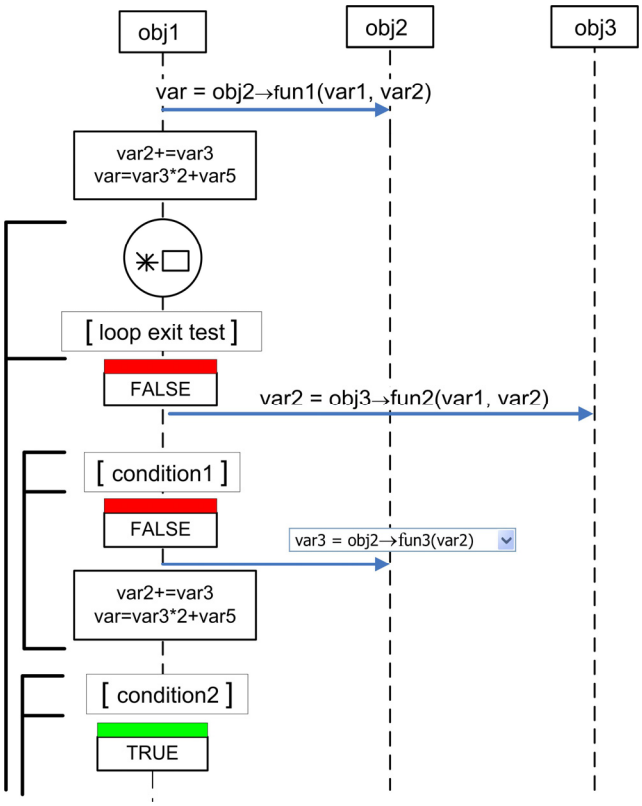

Figure 21. Design of sequential code

UML sequence diagrams can depict only one scenario. SystemCSP sequential design diagrams capture the complete control flow of the function that is visualized. Going through different scenarios takes place by toggling true/false values for the conditions, which results in the diagram being updated by displaying the relevant branch.

UML sequence diagrams display nested function calls. In SystemCSP, every sequential design diagram focuses on modeling the internals of exactly *one* function (either a global function or a member function of some object) and thus only the function calls made directly from the internals of the visualized function are specified. Displaying nested

function calls is not relevant for the current abstraction level, especially because the used function calls are assumed to provide certain services regardless of their internals.

A side benefit of the decision not to display nested function calls is that a source code model needed for construction of the diagram can easily be reconstructed from code (*reverse engineering*), since the contents of only *one* function needs to be parsed. When editing of the function is finished, code is generated and no additional data about the visualized function needs to be preserved.

Introduction of special kind of diagrams for representing sequential designs allows visualizing the coding process, which results in a completely visual design process. Entering the design by switching continuously between mouse and keyboard slows down the design process, especially when one applies it in lowest design level where most of source code is. For really efficient coding, it is expected that the prospective tool allows users to completely enter a diagram using keyboard shortcuts (e.g. arrows…). The presented form of the diagram, with predefined placement areas for visual elements, provides a form that enables design entering via keyboard only.

3.2 Components

In SystemCSP, in order to enhance reusability, a distinction is made between component types and component instances. In further text, the term component will be used to denote a component instance.

A *component* is a structural unit containing variables, code blocks, subcomponents and glue code between subcomponents. It is a reconfigurable and reusable unit, which can be used in different contexts and different interaction scenarios. In the CSP sense the behavior of the component is described as a process.

In our notation, components are enclosed in a rounded rectangle (see Figure 22) specifying the boundaries of the component. The entry point of component is always marked with a `start` event. Variables are defined in the component scope and represented via labels floating somewhere inside the component. A *variable label* contains the variable declaration consisting of its name and its type separated by a colon. Using variable labels allows omitting variable types in *event labels*. *Ports* are depicted as little rounded rectangles attached to the outer side of the component border. Inside a port is a *port label* carrying the externally visible name of the event or role associated with the port (see Figure 22).

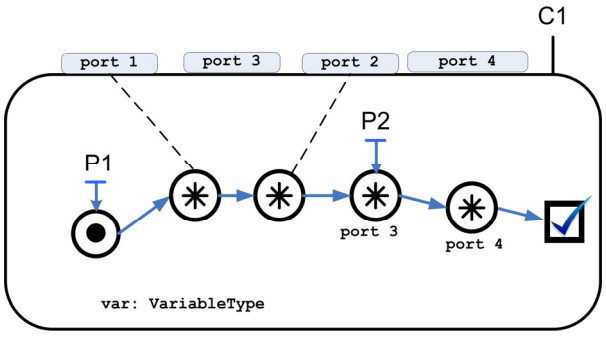

Figure 22. Component, ports and interfaces

In Figure 22, the behavior of the component C1 is represented via process P1. In case of port1 and port2, the relation between the *EventSync* processes and the associated port is done via a dashed line and in case of port 3 and port 4 via using the same name.

Let us consider a simple example of a Printer device driver component. Figure 23 shows a SystemCSP component specifying the same behaviour.

```
Printer = startup → Init
Init    = init → PrintMenu
PrintMenu = ( accept?job  → print!job  → PrintMenu)
            □ restart → Init
            □ shutdown → Printer
```

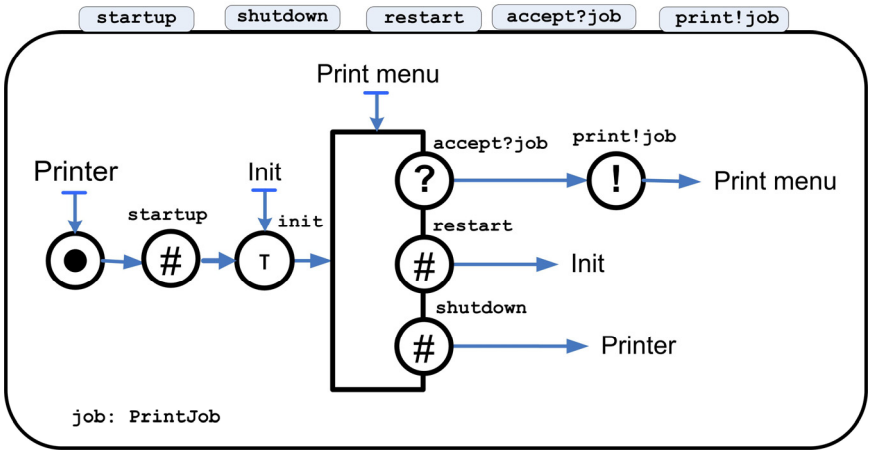

Figure 23. Printer process

`Printer` is both process and component, while `Print menu` and `Init` are only processes and not components. The `Printer` process is a component, because it is bounded via a rounded rectangle and has a specified set of ports.

Components can also be depicted using black-box approach: for example, components P, Q and R in Figure 24.

3.3 Interaction Contracts

An *Interaction diagram* is a diagram that specifies the way in which components interact. It usually contains a set of components (in black-box or transparent-box notation) centred around an interaction contract.

An interaction contract, in addition to interfaces of processes involved in the interaction, captures the complete set of *all* possible interaction scenarios. This is possible due to the fact that both contract and roles of participating processes are in fact CSP processes. An interaction contract is actually an entity that exists as components do, with a distinction that the basic purpose of contract is providing a management support for the interaction between components. Components participating in interaction contracts must provide an implementation that is the refinement (in CSP sense) of the role specified in the interaction contract.

Contracts usually have an internal component dedicated to managing the contract. For instance, let us imagine that the process `Race` from Figure 18 is specifying an interaction contract. In that case, the processes `Runner1` and `Runner2` could be considered to be roles implemented by some external components and all the rest would actually be an internal component managing the `Race` interaction contract.

The simplest interaction contracts are: event, shared memory, buffered channel, Any2One, One2Any, Any2Any channels. Interaction contracts can be made for any kind of application specific scenario and can also be reused in same way as components.

In this paper we focus on ways to visualize interaction contracts. A more elaborate discussion on interaction contracts and related concepts is provided in the companion paper.

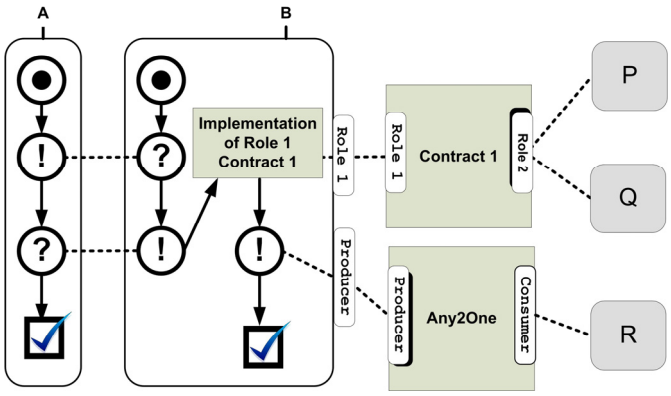

Figure 24. Interaction - direct and via contract

In Figure 24, an interaction view is given with several components cooperating directly or via simple interaction contracts. Component A and component B interact directly and for simplicity, even the port elements are omitted in the diagram. Component B participates in interaction with components P and Q in a way specified in the interaction contract named Contract 1. It implements Role 1 of that contract. Component B also participates in an Any2One contract as one of several possible Producers. In same contract, component R plays the role of the Consumer. The ports displayed in Figure 24 are associated with provided and required roles. Since role implementation is provided by the component and required in the contract, the ports associated with roles are depicted at the outside of the components and as plug-in sockets on the contract side. Both Contract1 and Any2One contract in Figure 24 are visualized using the black-box approach.

The internals of an interaction contract can also be visualized in an expanded (transparent-box) approach. Details of the Any2One interaction contract from Figure 24 are displayed using transparent-box approach in Figure 25.

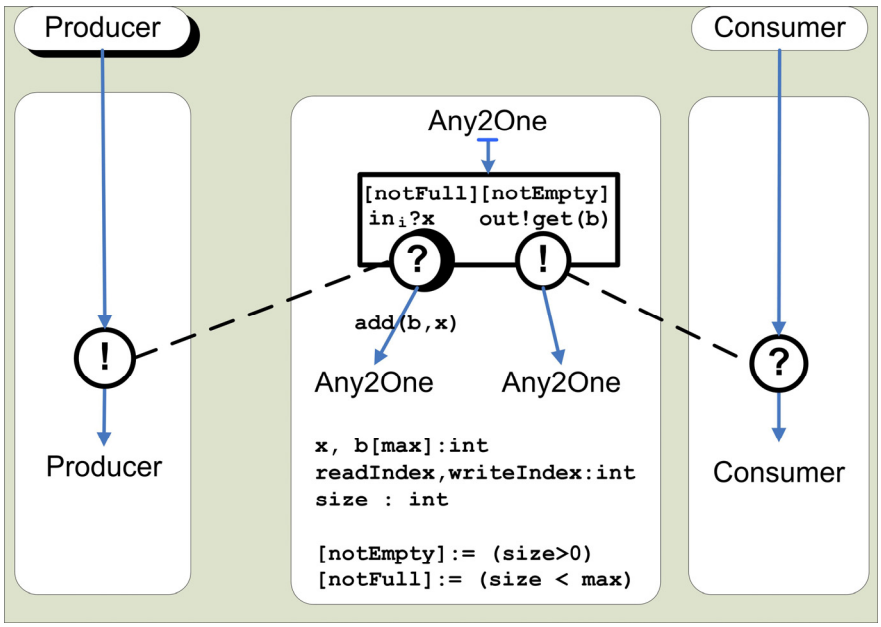

Figure 25. Any2One interaction contract

Note that in the transparent-box approach, roles are depicted in separate areas and associated with appropriate ports. The contract specifies at same time a CSP description of

the behaviour required by the roles (Producer and Consumer roles in Figure 25) and the CSP process description of the contract management layer (Any2One area in Figure 25).

The Any2One interaction contract in Figure 25 can buffer data arriving from several producers into the array `b` of size `max` used as circular buffer. In order to fulfill its task, it keeps track of the current size and positions of reader and writers using variables `size`, `readIndex` and `writeIndex`. The shaded Producer port means that more then one interleaving Producer can be attached. The same goes on for the shaded reader element in the guarded alternative inside the Any2One contract manager.

3.4 Distributed Systems – Allocation

This research is focused on systems that execute on distributed computer platforms and need to interact with processes from the physical world (plant in the further text). Interaction between computing nodes takes place over network interconnections and interaction between application and the plant takes place via I/O interfaces. In our approach, plants and computing nodes are considered components, and networks and I/O interconnections are considered system-level interaction contracts. The system level interaction diagram specifies interconnections between the chosen set of nodes and networks. Some components can be replicated on several nodes. Alternative network routes are visible in such a diagram.

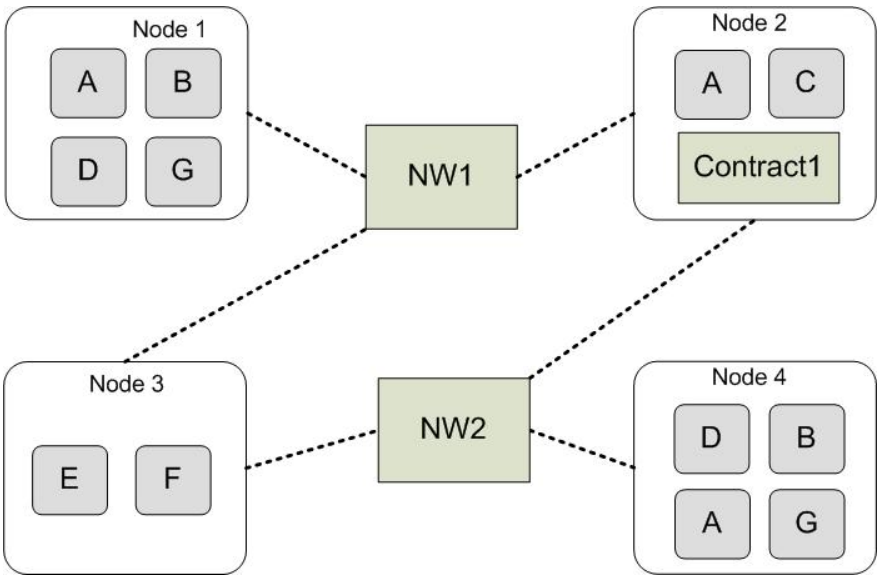

Figure 26. System-level interaction diagram

System-level interaction diagrams are usually used to illustrate the allocation of components and contracts participating in same interaction. Figure 26 illustrates a system-level interaction diagram concerned with the interaction of components A, B, C, D, E, F and G via interaction contract Contract1. In this configuration, Node 4 and Node 1 actually contain the same components. The connectivity of Node 1 to Nodes 2 and 3 is via network NW1 and Node 4 is connected to Nodes 2 and 3 via network NW2. This network topology suggests that this might be a design of a fault tolerant system that can survive a failure of either network NW1 or NW2. It can also survive the failure of either Node 1 or Node 2.

Software components and contracts must always belong to some node. One can express this visually by assigning a special port to every software component. This port needs to be attached to some node kind of component. This is comparable to power supply ports in electric circuits as opposed to data signal ports. Thus, this port is also depicted in a

somewhat different way, namely as a port inside the component/contract giving a visual impression of a plug-in socket for putting the software component on top of the node.

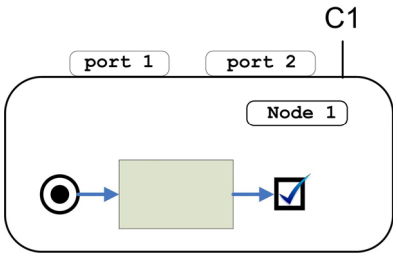

Figure 27. Node ports

3.5 Specifying Time Properties

Time properties can roughly be divided into two groups: time constraints and execution times. Time constraints specify that certain events take place precisely at some time or before some deadline. A deadline can be set relative to global time or as the maximally allowed distance in time between occurrences of two events. The deadline constraints are independent of the platform on which they are executed.

SystemCSP specifies time constraints in square brackets, next to the element they are associated with. The keyword `time` is used to denote the current time in the system. In Figure 28, the `start` event of process P1 is scheduled to be triggered periodically at precise moment in time. Event `ev1` is a point when time is observed and written to variable `time1`. Event `ev2` specifies that its occurrence should take place at most `d` time units after `time1`, or in other words at most `d` time units after the occurrence of the event `ev1`.

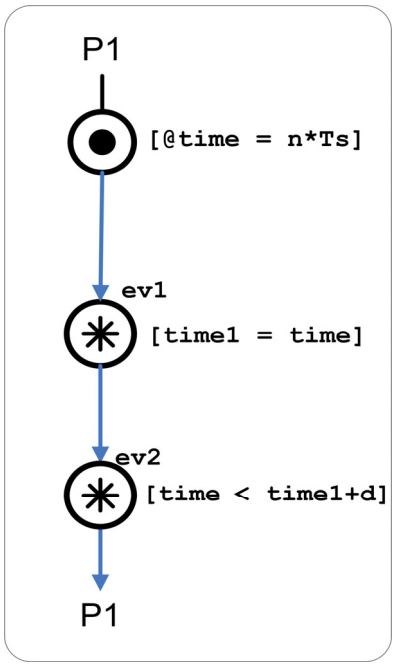

Figure 28. Specifying time requirements

The execution times exist only when a software component is associated with the node on which it is executed. Execution times can optionally be visualized on SystemCSP diagrams. More often, an execution time or a time interval is considered to be a property of the diagram element (e.g. computation process block).

4. Related Work

This section attempts to compare SystemCSP with several different approaches for specifying software systems. First, a comparison is made to UML, which is the de-facto industry standard for software development. Then a comparison is made to GML, as a predecessor of SystemCSP based on an occam-like approach and the concept of using binary compositional relationships.

4.1 SystemCSP vs. UML

SystemCSP gives precise semantics based on a formal method. It attempts to make a single type of diagram with elements that can cover all different aspects relevant for component-based design of real-time systems. SystemCSP basic elements are based on CSP operators and that gives possibility for formal verification.

A UML design starts with designing use case scenarios, that is with defining actors – users of the (sub)system and use cases – services offered by subsystem. Use cases are entities that can be related via inheritance, and via include and extend dependency relationships. In use case scenarios, the focus is on what should be done and not on how it is done. Actual use-case scenarios are usually specified via collaboration or activity diagrams.

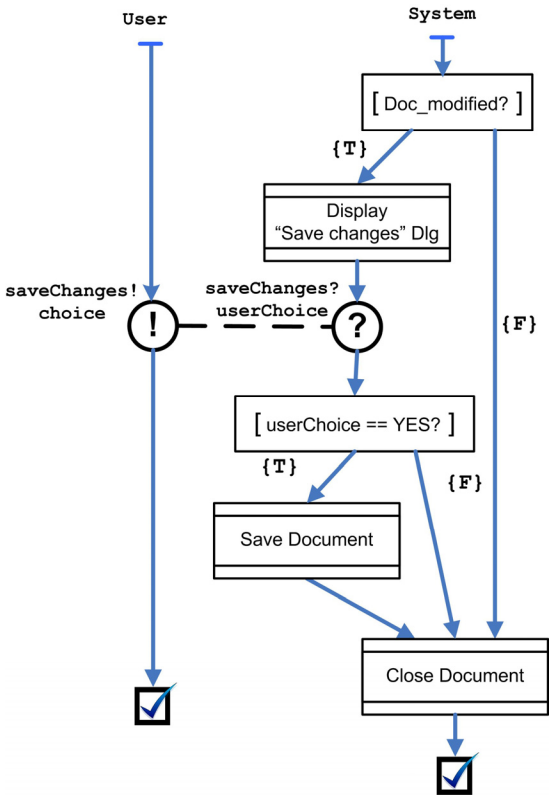

Figure 29. Closing document use case scenario

In System CSP, one can use interaction views to specify use-case scenarios. Black-box components of SystemCSP can in that case be used in similar way as actors of UML and interaction contracts can be used as use-case elements of UML. In principle inheritance, containment and dependency relationships between interaction contracts are naturally possible. Figure 29 illustrates how SystemCSP fits in the purpose of specifying use-case scenarios. Note that activities like displaying "`Save changes`" dialog, and performing

`save document` and `close document` activities are actually depicted as non-interacting processes. The reason is that on the observed level of abstraction, those activities do not communicate to the environment. However, inside they can contain events and processes.

State machine diagrams of UML origin from the StateChart approach. Comparison between statecharts and SystemCSP is given in section 2.8. As it was concluded, the main paradigm shift is in putting focus on events instead of on states. Otherwise, statechart diagrams can be considered directly translatable to SystemCSP. SystemCSP contains elements that do not have counterparts in FSM diagrams.

Activity diagrams of UML are also similar to SystemCSP diagrams. Actually, one can see activity diagram of UML as a subset of SystemCSP diagrams. An *activity* element of an activity diagram is essentially a black-box process or a component in SystemCSP. Fork and join of activity diagrams are PAR FORK and PAR JOIN in SystemCSP. A control flow arrow in an activity diagram is a prefix element in a SystemCSP diagram. A branch in an activity diagram is an IF choice in a SystemCSP diagram and the data flow in an activity diagram has in SystemCSP the more precise semantics of *EventSync* peer synchronization and data exchange.

Finally, interaction diagrams of UML are not directly comparable to SystemCSP because the basic paradigm is different. While UML is tied to OOP, SystemCSP is related to the CSP parallel programming model. As control flow in UML is a sequential invocation of operation or signaling events from parallel threads, CSP is based on rendezvous message passing. Still one can find some similarities in the purpose of those kinds of diagrams.

Collaboration diagrams emphasize structural aspects and the associations between objects via which messages are transferred. Timeline ordering is specified by attaching numbers to operations. Sequence diagrams forget about structural aspects and emphasize time ordering of messages by associating vertical dimension of the diagram with time. Both collaboration and interaction diagrams have a limitation that basically they can capture only a single timely ordered trace of messages – thus they are most useful as illustrations of part of some interaction.

SystemCSP focuses on expressing synchronization points between peer processes and not on displaying only one timeline trace of messages as a diagram. This principle covers all possible traces in a single diagram.

Computation processes are on a lower level where there is no possibility for concurrency and they can be designed using any kind of UML diagram. However, in order to utilize maximum of it and be able to perform automatic transformations from graphical representation to code and vice versa, the special kind of diagrams loosely based on ideas from UML sequence diagrams is proposed in section 3.1.

4.2 SystemCSP vs. GML

SystemCSP started as an attempt to enhance the expressiveness of GML, especially to include state machine behavior. However, SystemCSP diverged to a completely different notation based on CSP.

SystemCSP is capable of expressing any CSP behavior. This is a consequence of the different paradigm where a process is, namely the same as in CSP, considered to be just the behavior and not a tangible entity as it de facto is in occam and GML.

Instead of using binary compositional relationships, SystemCSP introduces control flow elements. This approach seems to give somewhat better readability concerning observing control flow patterns. START and EXIT pair of control flow elements determine boundaries of the constructs.

Besides cluttered readability, caused by using only binary relationships and no control flow elements, two main deficiencies exist in GML: lack of explicit support for component based design and restriction to the occam subset of CSP.

Control flow elements augmented with grouping symbols as introduced in SystemCSP, also create the opportunity to scatter same components in different views concerned with different aspects of the system. More details on how is that actually done can be found in the companion paper[1].

Compared to CSP, the occam subset of CSP offers only limited expressiveness for specifying and designing concurrent systems. As a consequence, GML as occam-oriented approach suffers from same limited expressiveness.

The introduction of process entry labels and recursion process labels in SystemCSP is one of the elements that enable full expressiveness of CSP. While GML does not allow recursions other then loops, SystemCSP allows mutual recursions to be specified. Process labels also help to make diagrams more readable by omitting the prefix lines connecting recursion invocation to recursion entry point.

In SystemCSP, it is also possible to use a FORK kind of element (e.g. guarded alternative element) without the corresponding JOIN element. This lays foundation for constructing finite state machine-like diagrams. GML has, as occam, only the ALT kind of choice. ALT is a choice between two processes with control flow continuing on same place after a chosen alternative regardless of what was the choice. SystemCSP follows strictly CSP semantics and introduces a guarded alternative control flow element that essentially forks alternatives without forcing the existence of common join place. This allows building diagrams alike to FSM diagrams.

GML has advantage that for visualizing given design, total number of used elements can be less, because control flow elements are present as relationships, and not as explicit elements. But this advantage comes with already mentioned consequence that exact control flow is more difficult to observe. In that sense a choice between SystemCSP and GML is a tradeoff between the efficiency of notation (measured in the number of used elements) and its readability.

Finally, SystemCSP does not have to be complete substitute for GML designs. Parts of designs can still be entered in GML. Since GML is based on occam which is subset of CSP and SystemCSP is based on CSP, designs from GML can be expressed in SystemCSP or made understandable to a prospective SystemCSP based tool. Both kinds of designs can, according to their own set of rules, be used for code generation of both source code and CSPm scripts, allowing mixed SystemCSP and GML design. The only restriction is that while GML designs can be nested in SystemCSP designs, the other way around is not possible.

5. Conclusions

A novel way of component-based system description founded on the CSP process algebra is proposed. This notation can represent visually any CSP description and any system constructed via the presented notation can be expressed as a set of CSP descriptions. Components are distinguished from interaction contracts. Finally, a way to specify exception handling, distribution properties and time properties is introduced. The notation is compared to relevant related methodologies.

A library and a tool that will offer support for SystemCSP are under development. To fully exploit benefits of the SystemCSP notation, a tool should be capable to perform automatic code generation both in a form of executable code and CSP scripts. The tool should also provide support for a simulation framework with animation capabilities.

Simulation can be performed either by interpreting the model inside the tool or by obtaining feedback from running the generated executable.

Further work is to test the SystemCSP notation on more complex examples. We use our Production Cell [13] setup as a study case.

More information about the developments concerning the SystemCSP tool and library will be available on www.ce.utwente.nl web site.

Acknowledgments

The authors are grateful to Job van Amerongen, Marcel Groothuis, Matthijs ten Berge, Dusko Jovanovic and Peter Visser for their valuable comments and suggestions regarding the SystemCSP theory and contents of this paper.

References

[1] B. Orlic and J. F. Broenink, "Interacting components", presented at CPA 2006.
[2] A. W. Roscoe, *The Theory and Practice of Concurrency*: Prentice Hall, 1997.
[3] G. H. Hilderink, "Graphical modelling language for specifying concurrency based on CSP," *IEE Proceedings Software*, vol. 150, pp. 108-120, 2003.
[4] C. Szyperski, "Component Software: Beyond Object-Oriented Programming," 1998.
[5] S. Schneider, *Concurrent and Real-Time Systems: The CSP approach*: Wiley, 2000.
[6] J. Magee and J. Kramer, *Concurrency: state models & Java programs*: Wiley, 1999.
[7] G. Booch, J. Rumbaugh, and I. Jacobson, *The UML Reference Guide*: AddisonWesley, 1999.
[8] P. Marwedel, *Embedded system design*: Kluwer Academic Pubilshers, Dordrecht, Nethelands, 2003.
[9] C. F. J. Lange and M. R. V. Chaudron, "In practice: UML software architecture and design description," *IEEE software*, vol. 23, pp. 40, 2006.
[10] C. Crichton, J. Davies, and A. Cavarra, "A Pattern For Concurrency in UML," presented at FASE, 2002.
[11] G. H. Hilderink, "Managing Complexity of Control Software through Concurrency," vol. PhD: University of Twente, 2005.
[12] W. L. Yeung, S. A. Schneider, and F. Tam, "Design and verication of distributed recovery blocks in CSP", University of London 1998.
[13] L. S. van den Berg, "Design of a Production Cell setup", University of Twente, 2006.