Communicating Process Architectures 2007
Alistair McEwan, Steve Schneider, Wilson Ifill and Peter Welch (Eds.)
IOS Press, 2007

Design Principles of the SystemCSP Software Framework

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

Abstract. SystemCSP is a graphical design specification language aimed to serve as a basis for the specification of formally verifiable component-based designs. This paper defines a mapping from SystemCSP designs to a software implementation. The possibility to reuse existing practical implementations was analyzed. Comparison is given for different types of execution engines usable in implementing concurrent systems. The main part of the text introduces and explains the design principles behind the software implementation. A synchronization mechanism is introduced that can handle CSP kind of events with event ends possibly scattered on different nodes and OS threads, and with any number of participating event ends, possibly guarded by alternative constructs.

Keywords. Concurrency, CSP, SystemCSP, code generation

Introduction

Concurrency is one of the most essential properties of the reality as we know it. In every complex system, it can be perceived that many activities are taking place simultaneously. Better control over concurrency structure should automatically reduce the problem of complexity handling. Thus, a structured way to deal with concurrency is needed.

SystemCSP [1] 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 cover various aspects needed for the design of distributed real-time systems. SystemCSP is based on principles of both component-based design and CSP process algebra. According to [2] "CSP was designed to be a notation and theory for describing and analyzing systems whose primary interest arises from the ways in which different components interact". CSP is a relevant parallel programming model and the SystemCSP design specification method aims to foster its utilization in the practice of component-based design.

Occam was a programming language loosely based on CSP. Nowadays, occam-like libraries exist for modern programming languages. JCSP [4] developed in Kent, and CT libraries [5, 6] developed in our lab, are examples of occam-like libraries. Both approaches rely on OOP principles to implement an API that mimics the syntax of occam.

This paper defines the architecture of a framework for the software implementation of SystemCSP designs. As illustrated in Figure 1, software implementation is one of the possible target domains for a model specified in the SystemCSP design domain. This paper does focus on the infrastructure needed in the target domain to support the implementation of a model specified in SystemCSP (e.g. the one on Figure 2 or Figure 3).

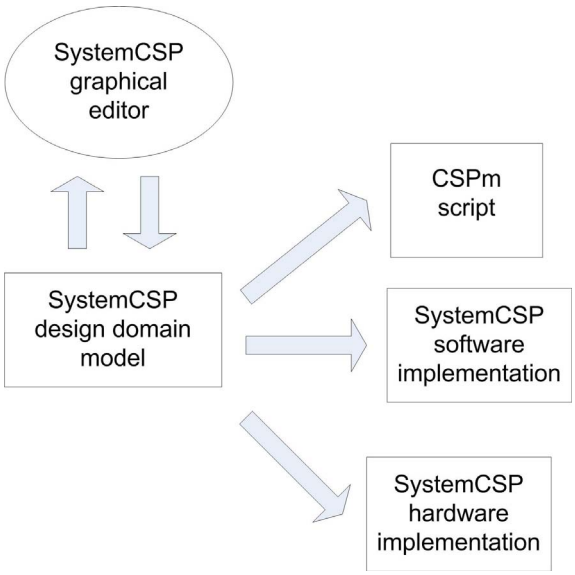

Figure 1 SystemCSP source and target domains

The SystemCSP notation has a control flow oriented part that is more or less a direct visualization of CSP primitives, and an interaction oriented part based on binary compositional relationships. In addition, the primitives for component-based software engineering are introduced.

The following CSP expression

```
P1= ev1→(P;(T;(Q ‖ R)))
```

is in Figure 2 represented using the control flow oriented part of SystemCSP.

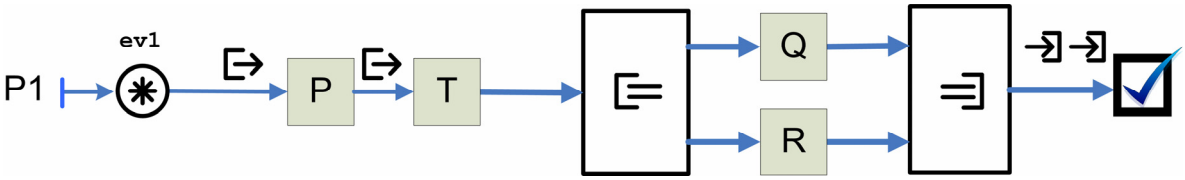

Figure 2 Example of control flow oriented SystemCSP design

In Figure 3, on the right-hand side, a control flow oriented design is visualized, and on the left-hand side two views are shown, each focusing on a part of the interaction between the involved components. Note also that instead of process symbols as used in Figure 2, in Figure 3 symbols for components and interaction contracts are used.

A detailed introduction of SystemCSP elements is out of the scope of this paper. For more details about SystemCSP design domain notation, the reader is referred to [6].

In this paper, in the Section 1, the discussion is focused on the possibility to reuse the CT library, developed at our lab, as a target domain framework for code generation.

After discarding the possibility to reuse the CT library, the discussion about the basic design principles for a new library starts in Section 2 with investigating practical possibilities for implementing concurrency. Possible types of execution engines are listed in Section 2.1. In Section 2.2, a flexible architecture is proposed that allows a designer to make trade-offs regarding the used structure of execution engines. In Section 2.3, a design of component internals is introduced, that allows subprocesses to access variables defined in parent components and offers a way to reuse processes in same way as components.

Section 2.4 explains the way in which function-call based concurrency is applied to structure concurrency inside components. An example is given illustrating how this mechanism actually works.

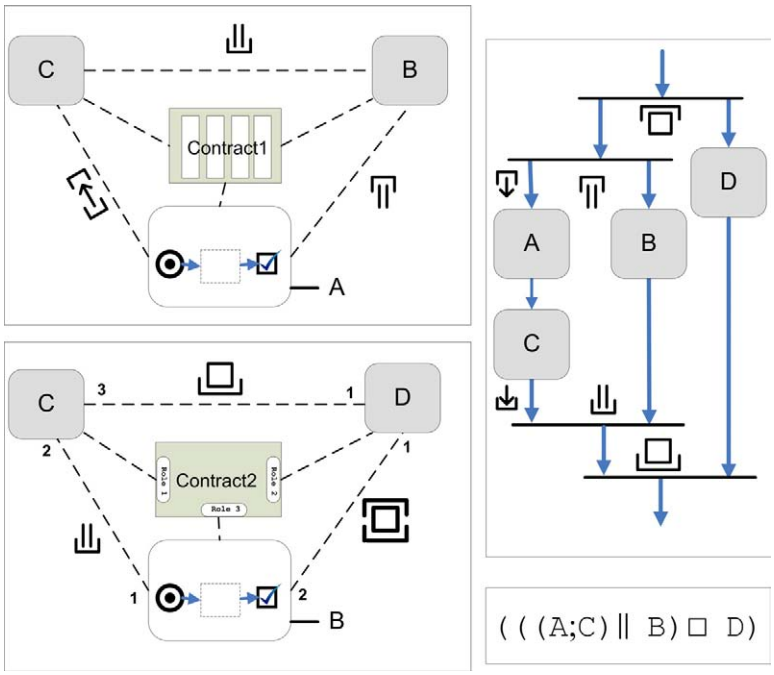

Figure 3 Example illustrating the relation between an interaction-oriented part and a control-based part

Section 3 explains a synchronization mechanism designed to handle CSP kind of events with any number of participants and with some of them possibly participating in several guarded alternative constructs. A special problem that was solved related to this was achieving mutual exclusion when event ends and the associated synchronization points are potentially scattered in different operating system threads or on different nodes.

Section 4 introduces design of a mechanism that implements exception handling and of mechanisms that provide support for logging and tracing.

1. Why Yet Another CSP Library?

In this section we focus on a possibility to reuse the CT library, the occam-like library developed in our lab, as a framework for the software implementation of SystemCSP models. The CT library follows the occam model as far as possible. SystemCSP builds upon the CSP legacy. It does in addition introduce new elements related to the area of component-based engineering. However, those newly introduced elements are: 1) components and interaction contracts that both map to CSP processes and 2) ports that are just event-ends exported by such CSP processes.

In fact, SystemCSP defines auxiliary design time operators like the fork and join control flow elements and binary compositional relationships of FORK, JOIN, WEAK and STRONG types. Those auxiliary operators do exist only during the design process and are therefore after grouping, in mapping to CSPm target domain substituted with CSP operators, and in mapping to software implementations with constructs like the ones existing in occam and CT library.

Basic SystemCSP control flow elements and binary relationships do map to the

constructs as it is the case in the CT library. However, since SystemCSP aims to correspond exactly to CSP, it cannot be implemented completely by occam-like approaches that do put only restricted part of CSP into practical use. Following text will explore those differences in more details.

In CT library, like in its role-model occam, a `Parallel` construct spawns separate user-level threads for every subprocess. Synchronization points are defined by channel interconnections. The SystemCSP design domain allows both the CSP way of event synchronization (through a hierarchy of processes), and the occam-way with direct channel interconnections. Thus, a software implementation of SystemCSP designs needs mechanism for the hierarchical CSP-like CSP event synchronization.

In SystemCSP, as in CSP, data communication over a channel can be multidirectional involving any number of data flows. The CT library, as occam, has only unidirectional channels. In addition those channels are strongly typed using the template mechanism of the C++ language and as a consequence, they are not flexible enough to be reused in constructing the support for multidirectional communication. Thus, the channel framework of the CT library is not reusable.

The CT library implements the Alternative construct as a class whose behavior is based on the ideas of the occam ALT construct. The implementation of the Alternative construct [5] allows several different working modes (preference alting, PriAlternative, fair, FIFO), introduced to enable an alternative way to make a deterministic choice in case when more then one alternatives are ready for execution at the same time. The alting in CT library assumes that a channel can be guarded by some alternative construct only from one of the exactly two event-end sides (there can be either an input or an output guard associated with a channel). A guarded channel is just a channel with an associated guard. A guard is an object inside an alternative construct associated with a channel and a process. When a guarded channel is accessed by the peer process, then the guard becomes ready and is added to the alting queue. The way in which guards are ordered in this queue, determines the working mode (preference alting, PriAlternative, fair, FIFO) of the alternative construct. An alternative construct is thus a single point where the decision of a choice is made.

The SystemCSP design domain makes a difference between an external choice and a guarded alternative operators and in that sense adhere strictly to CSP. Thus, an implementation is needed that can support both. Event-ends contained by a guarded alternative or the ones resolving the parent external choice operator need to delegate their roles in the process of CSP event synchronization to the related guarded alternative or external choice operator. In case when, in an event occurrence, any number of guarded event-ends can participate, the whole alting mechanism must be completely different then the one applied in CT library. This means that in fact for CSP event synchronization mechanism completely different implementation of alting needs to be implemented. Thus again in this respect too, the CT library is not useful.

Simple CSP processes, made out of only event synchronization points connected via the prefix and the guarded alternative operator, are often visualized using a *Finite State Machine* (FSM). With the guarded alternative of CSP, no join of branches is assumed, and the branches can lead to any other state. The occam/CT library choice (ALT construct) requires that all alternatives are eventually joined. Thus a natural FSM interpretation is not possible anymore.For SystemCSP, the ability to implement FSM-like designs in a native way is especially important. Thus, implementation of the guarded alternative operator should not assume the join of branches.

In addition, it should be possible to use process labels to mark process entry points and allow recursions other then repetitions as in the SystemCSP design domain. Since in occam and the CT library, processes are structural units like components in SystemCSP, the use of

recursion different then a loop is not natural there. A strict tree hierarchy of processes and constructs as basic architecture design pattern of occam and CT library is a misfit for our purpose. Thus, again the CT library does not meet the requirements imposed by SystemCSP.

In fact, instead of processes as structural units arranged in strict tree hierarchy, flexibility can be introduced by using classes for implementation of some processes and functions and labels for other processes. For instance, a single FSM-like design can contain many named processes that in fact do name the relevant states. Certainly, those processes cannot map to the occam notion of process. They are more convenient to be implemented as labels, while the whole finite state machine is convenient to be a single function.

In addition, SystemCSP is intended to be used as a design methodology for design and implementation of component-based systems. This needs to be supported by introducing appropriate abstractions and also possibilities for easy reconfiguration, interface checking, and so on.

To conclude, the mismatch between the CT library and the needs of SystemCSP is to big to allow reusing the CT library as a framework for the software implementation of SystemCSP designs.

2. Execution Engine Framework

2.1 Brief Overview of Execution Engines

Concurrency in a particular application assumes the potential of parallel existence and parallel progress of the involved processes. If processes are implemented in hardware, or if each of the processes is deployed on a dedicated node, these processes can truly progress concurrently. In practice, multiple processes often share the same processing unit.

Operating systems provide users with the possibility to run multiple OS processes (programs). Every OS process has its own isolated memory space and its own set of allocated resources. Within OS processes it is possible to create multiple OS threads that have their own dedicated workspaces (stack), but share other resources with all threads belonging to the same process. Synchronization in accessing those resources is left to the programmer. OS synchronization and communication primitives (semaphores, locks, mutexes, signals, mailboxes, pipes…)[7] are not safe from concurrency related hazards caused by bad design [4]. OS thread context switch is heavyweight, due to allowing preemption to take place at any moment of time.

User-level threading is an alternative approach that relies on creating a set of own threads in the scope of a single OS thread. Those threads are invisible to the underlying OS-level scheduler and their scheduling is under the control of the application. The main advantages compared to OS threads are the speed of context switching and gaining control over scheduling. The use of Operating System calls from inside any user-level thread is blocking the complete OS thread with all nested user-level threads (operating system call problem).

Another approach is to implement concurrency via function-calls, where the concurrent progress of parallel processes is achieved by dividing every process into little atomic steps. After every atomic step, the scheduler gets back control and executes the function that performs the next atomic step in one of the processes. There is no need to dedicate a separate stack for every process. Steps are executed atomically and cannot be preempted. A function-calls based approach is often used to mimic concurrency in simulation engines. There is even an operating system (Portos [8]) that is based on scheduling prioritized function calls.

2.1.1 Discussion

SystemCSP [1] structures concurrency, communication and synchronization using primitives directly coupled to appropriate CSP operators. To implement concurrent behaviour, it is possible to use any of the approaches described in Section 2.1.

The CT library is based on user-level threading. Every process in the CT library that can be run concurrently (i.e. every subprocess of the (Pri)Parallel construct) has a dedicated user-level thread. A scheduler exists that can choose the next process to execute according to the hierarchy of Parallel/PriParallel constructs. As in occam, rendezvous channels are the basic communication and synchronization primitives. Possible context switching points are hidden in every access to local channels.

The first important issue related to the SystemCSP framework is what type of execution engine is best to choose. Actually, the optimal choice depends on the application at hand and is a compromise between the level of concurrency, the communication overhead and other factors. The best solution is, therefore, to let the designer choose the type(s) of execution engines on which the application will execute. A way to do this is to separate the application from the execution engines, and to let the designer map the components of his application to the underlying architecture of execution engines.

2.2 Four Layer Execution Engine Architecture

An application is in SystemCSP organized as a containment hierarchy of components and processes. A component is the basic unit of composition, allocation, scheduling and reconfiguration. Inside every component, contained components, processes and event-ends are related via CSP control flow elements (sequential, parallel, choice …). While a subprocess is inseparable part of its parent component, a subcomponent is independent and can for example be located on some other node.

As a result of the previous discussion, flexible execution engine architecture is proposed, that allows the user to adjust the level of concurrency to the needs of the application at hand. The execution engine architecture is hierarchical, based on four layers: node/OS Thread/UL thread/component managers. Any component can be assigned to any execution engine on any level in such a hierarchy.

The class diagram given in Figure 4 defines the hierarchy of the execution engines. In the general case, inside an operating-system thread, a user-level scheduler exists, which can switch context between its nested user-level threads. Inside a user-level thread is, in the general case, a component manager that can switch between the contained components. Every component has an internal scheduler that will use a function-call based concurrency approach to schedule nested subprocesses.

Internalizing the scheduler inside every component allows more flexibility in the sense that some levels in the 4-layer architecture can be skipped. The concurrency of the node execution engine can be delegated to operating system threads or to user level threads or to component managers or it can execute a single component directly without providing support for lower-level execution engines. It is even possible to have a single component per node. Similarly operating system threads can execute a set of user level threads, or a component manager or a single component. A user-level thread is able execute just a single component or a set of components via the component manager. The possibility to choose any of those combinations is actually reflected in Figure 4.

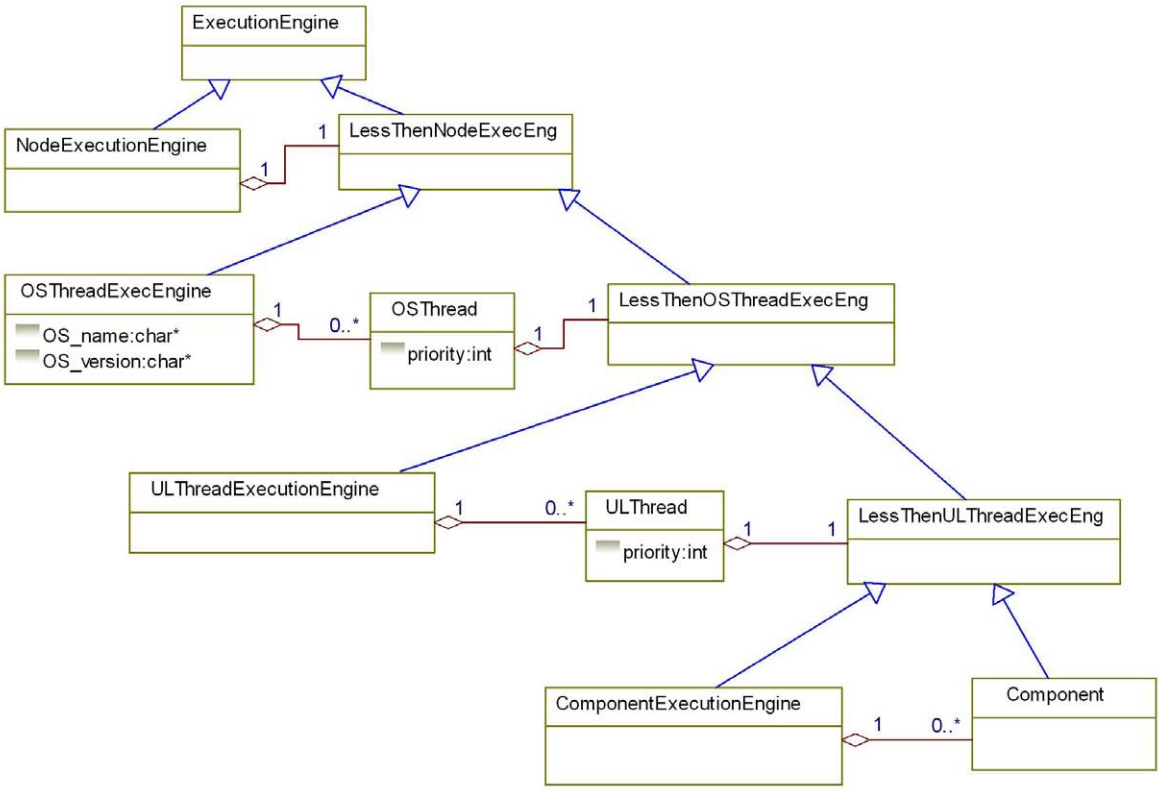

Figure 4 Class diagram of the 4-layer execution engine framework

The OS thread execution engine is in fact representing the scheduling mechanism of the underlying operating system. Therefore, in the design domain this class contains the name and version number of the used operating system as attributes. In software implementation, there is no matching class since implementation is provided by the underlying operating system. The OS thread class in the software implementation domain does have a dedicated subclass for every supported operating system. In that way, the portability is enhanced by isolating platform-specific details in the implementation of subclasses. Auxiliary abstract classes LessThenodeExecEng, LessThenOSThreadExecEng and LessThenUL-ThreadExecEng are.introduced to enable the described flexibility in structuring the hierarchy of execution engines.

2.2.1 Allocation

An allocation procedure as the one depicted in Figure 5 (below here), is a process of mapping components from the application hierarchy of components to the hierarchy of execution engines. The criteria for the choice of the execution framework and for the allocation, is setting the proper level of concurrency while optimizing performance by minimizing overhead. Two components residing on different nodes can execute simultaneously. Two components allocated to the same node, but to different operating system threads can be executed simultaneously only in the case of multi-core or hyper-threading nodes. Communication overhead between two components is directly proportional to the distance between the execution engines that execute them.

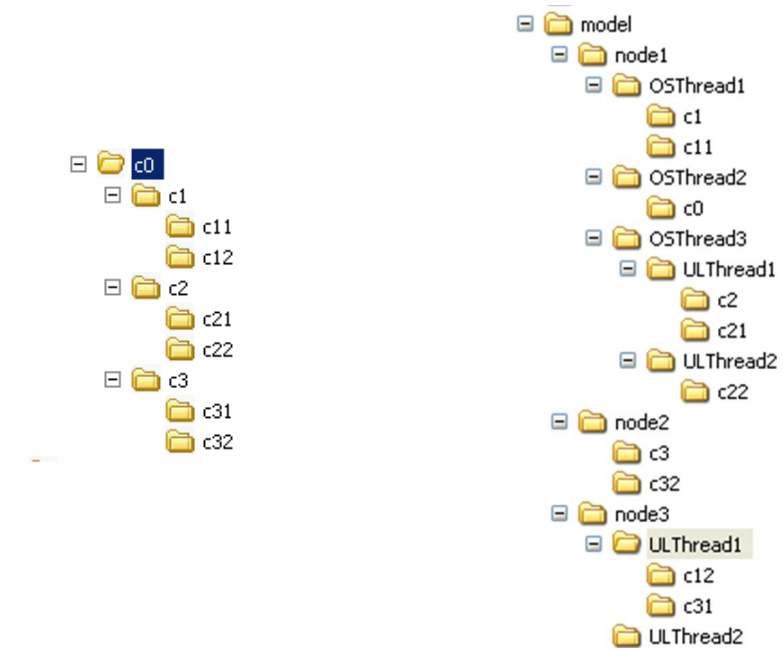

Figure 5 Allocation = mapping components from application hierarchy to hierarchy of execution engines

Control flow (as specified by parallel, sequential and alternative constructs) is decoupled from its execution engines. As a result, components can be reconfigured more easily. A component can be moved from one (node, operating-system thread, user-level thread) execution engine to another. Components can be dynamically created, moved around and connected to interaction contracts. On dynamical reconfiguration, checking compatibility of the interface required by the interaction contract with the interface supported by the component is done.

2.2.2 Priority Assignment

CSP is ignorant of the way concurrency is implemented. Concurrency phenomena involving parallel processes interacting via rendezvous synchronizations are the same regardless whether concurrent processes are executed on dedicated nodes, or sharing CPU time of the same node is done according to some scheduling algorithm. However, temporal characteristics are different in these two cases. The most commonly applied scheduling schemes are based on associating priorities with processes. In real-time systems, achieving proper temporal behavior is of utmost interest. Therefore, in real-time systems priorities are attached to schedulable units according to some scheduling algorithm that can guarantee meeting time requirements.

In addition to the PAR (parallel) construct, in occam a prioritized version of the parallel construct, the PRIPAR construct, was introduced. It specifies parallel execution with priorities assigned according to the order of adding subprocesses to the construct. However, on transputer platforms only two priority levels were supported. Additional priority levels were sometimes implemented in software [9].

Following occam, the CT library introduces a PriParallel construct with the difference that inside one PriParallel up to 8 subprocesses can be placed. While all subprocesses of a Parallel construct have the same priority, priorities of processes inside a PriParallel are based on the order in which they are added to the construct. This allows for a user-friendly priority assignment based on the notion of the, more or less intuitive, relative importance of a process compared to the other processes. The `PriParallel` construct is as any other construct also a kind of process, and as such it can be further nested in a hierarchy of

constructs. This leads to the possibility to use a hierarchy of `PriParallel` and `Parallel` constructs to create a program with an unbounded number of different priority levels. Note however, that priority ordering, of all processes in a system, if defined in this way is not necessarily a strict ordering, but rather a set of partial orderings. If only PriParallel constructs were used, a set of partial orderings results in global strict priority ordering.

As in execution-engine architecture issues, where the conclusion was that flexibility can be achieved by separating hierarchy of components belonging to the application domain, from the hierarchy of execution engines, the similar reasoning applies to specifying priorities. The PriPar construct of the occam-like approaches is hard-coding priorities in the design, where a intuitively priority assignment is related to the execution of processes on the real target architecture. Priority values are in fact the result of a trade-off due to temporal requirements that belong to the application domain and processing time that belongs to the domain of underlying architecture engines. Therefore, the choice is *not* to follow the occam-like approach. Priorities belong to the execution engine framework and not to the application framework. Instead of relative priorities in each Par construct, a component from application hierarchy of components can be mapped to the execution engine of appropriate priority.

Every operating-system thread has a priority level used by the underlying operating-system scheduler to schedule it. Every user-level thread has its own priority level which defines its importance compared to the other user-level threads belonging to the same operating-system thread. In this way, a 2-level priority system exists and any component can be assigned to the pair of operating-system thread and user-level thread with appropriate priority levels

Note that the priorities specified on higher levels in an execution engine hierarchy overrule the ones specified on lower levels. This is the case because a higher-level execution engine (an operating-system execution engine) is not aware of the lower-level schedulable units (e.g. a user-level thread).

A problematic situation occurs when two components of different user-level thread priorities are allocated to two different operating-system threads of the same operating-system thread priority. In that case, it can happen that advantage is given to the component that has a lower user-level thread priority. In case when such a scenario should be avoided, two components with the same operating-system thread priority should always be in the same operating-system thread. In other words, this problem is avoided when there are no operating-system threads of the same priority on one node.

An additional issue is priority inversion that happens when a component of higher priority interacts with one of lower priority via rendezvous channels. For more details about this problem and possible solutions, the reader is referred to the related paper[10].

2.3 Components, Processes and Variables

The UML class diagram in Figure 6 illustrates the hierarchy of classes related to the internal organization of components. Every component has an internal scheduler that can handle various schedulable units (construct, processes, guarded alternative operators and event ends).

Variables are in SystemCSP defined in the scope of the component they reside in, and should be easily accessible from subprocesses of that component. A subprocess is allowed to access the variables defined in its parent component, but subcomponent cannot – because a subcomponent can be executed in a different operating-system thread or even on a different node. Instead of defining actual variables, the process class does define references to these variables (see Figure 6). Those references are in the constructor of the process

associated with real variables defined in the scope of the component. In this way, subprocesses can access variables defined in components without restrictions; Component definitions are divided into smaller parts that are easier to understand and processes become as reusable as components are.

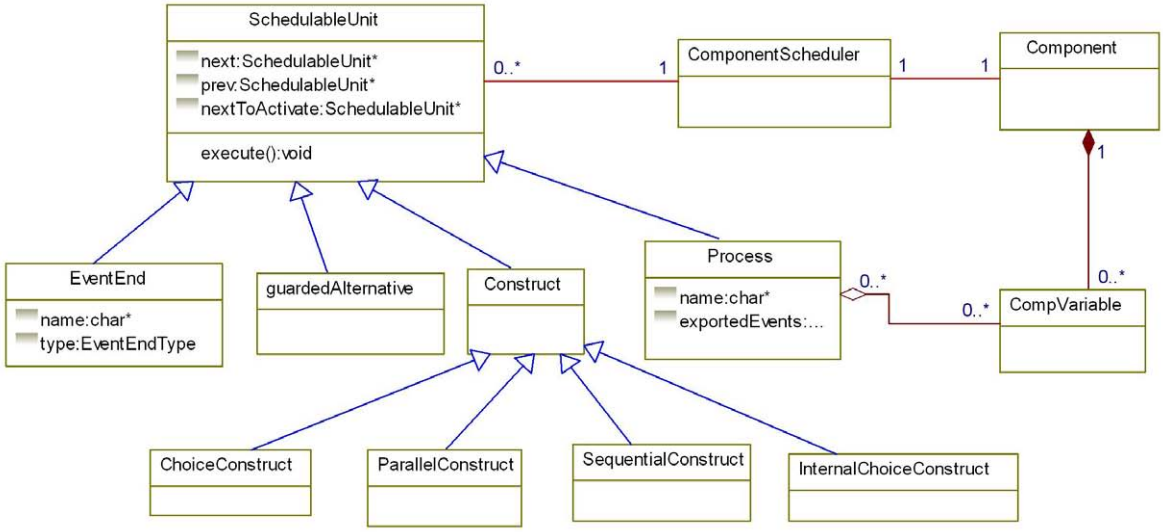

Figure 6 UML class diagram illustrating the relations between components and processes

Subcomponents that are executed in different execution engines do have associated proxy subporocess in their parent component (see Figure 7). In that way, the synchronization between the remote subcomponent and its parent component is done indirectly via that proxy process. The Proxy process and remote subcomponent synchronize on start events and termination events via regular channels.

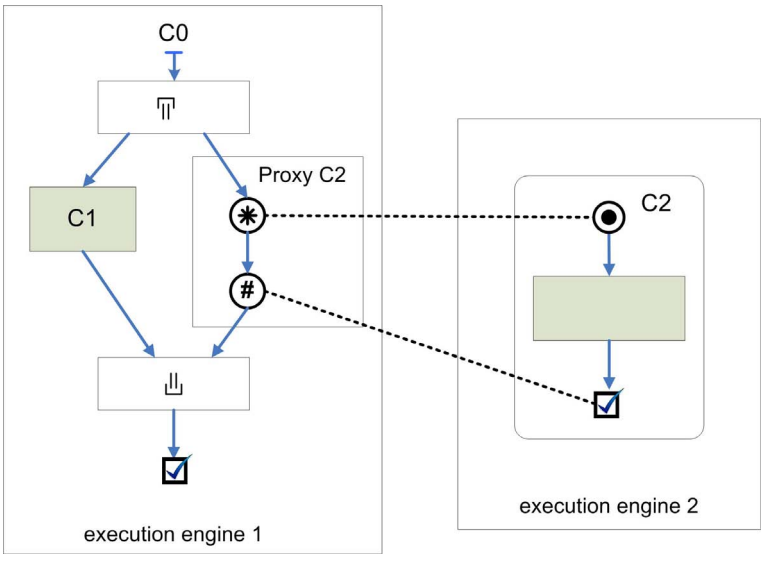

Figure 7 Using proxy processes to relate remote subcomponents to parent constructs

2.4 Function Call Based Concurrency Inside Components

The class diagram in Figure 6 defines that each component contains an internal scheduler. The dispatcher of a component is in its execute() function. It will use a scheduling queue

(FIFO or sorted queue) to obtain the pointer to the next schedulable unit ready to be executed.

Every schedulable unit inside a component is implemented as a finite-state machine that performs one synchronization and computation step per each function call, and subsequently returns control back to the component scheduler. The current place where the schedulable unit stopped with its execution is remembered in its internal state variable. When the schedulable unit is activated a next time, it will use this value to continue from where it had stopped. Every schedulable unit does have associated a pointer to the next schedulable unit to activate when its execution is finished. This is either its parent construct or the next schedulable unit in sequence (if the parent is a sequential construct).

Every construct exists inside some parent component. Constructs (Parallel, Alternative and Sequential) as well as channel/event ends are designed as predefined state-machines that implement behavior expected from them.

For instance, a simplified finite state machine implementing the Parallel construct would have two states: one with forking subprocesses (the FORK state in code snippet bellow), and one waiting for all subprocesses to finish (JOIN state in the code snippet). In reality a mechanism for handling errors and exceptional situations requires one or two additional states.

```
Parallel::run(){
  switch(state){
        case FORK:
              parentComponent->scheduler->add(subprocesses);
              state = JOIN;
              result =0;
              break;
        case JOIN:
              if(finishedCount == size)
              {
                    state = FORK;
                    finishedCount=0;
                    parentComponent->scheduler->add(next);
                    result =1;
              }
              break;
  }
  return result;
}

Parallel::exit() {
  finishedCount++;
  if(finishedCount ==size) parentComponent->scheduler->add(this);
}
```

The subprocesses use the exit() function to notify the Parallel construct that they have finished their execution. Since all subprocesses are in the same component and executed in atomic parts in function-call based concurrency manner, there are no mutual exclusion hazards involved.

When a construct finalizes successfully its execution, it returns a status flag equal to 1 or higher. For its parent it is a sign that it can move to the next phase in its execution by

updating its state variable. In case of a guarded alternative, the returned number is in the parent process understood as the index of the branch to be followed and it is used to determine the next value of the state variable.

Thus, the system works by jumping in a state-machine, making one step (e.g. executing a code block or attempting event synchronization or forking subprocesses), and then jumping out. This might seem inefficient, but actually also in the user-level thread situation, a similar thing is done: testing the need for a context switch is hidden in every event attempt. Only performance testing can show which way is actually more efficient under what conditions. Recursions that are used to define auxiliary, named, process entry points are not implemented in a separate class. Instead they are naturally implemented using labels.

Let us use the example given in SystemCSP (Figure 8), and also in CSPm code above the figure to display how its software implementation would look like in this framework.

```
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
```

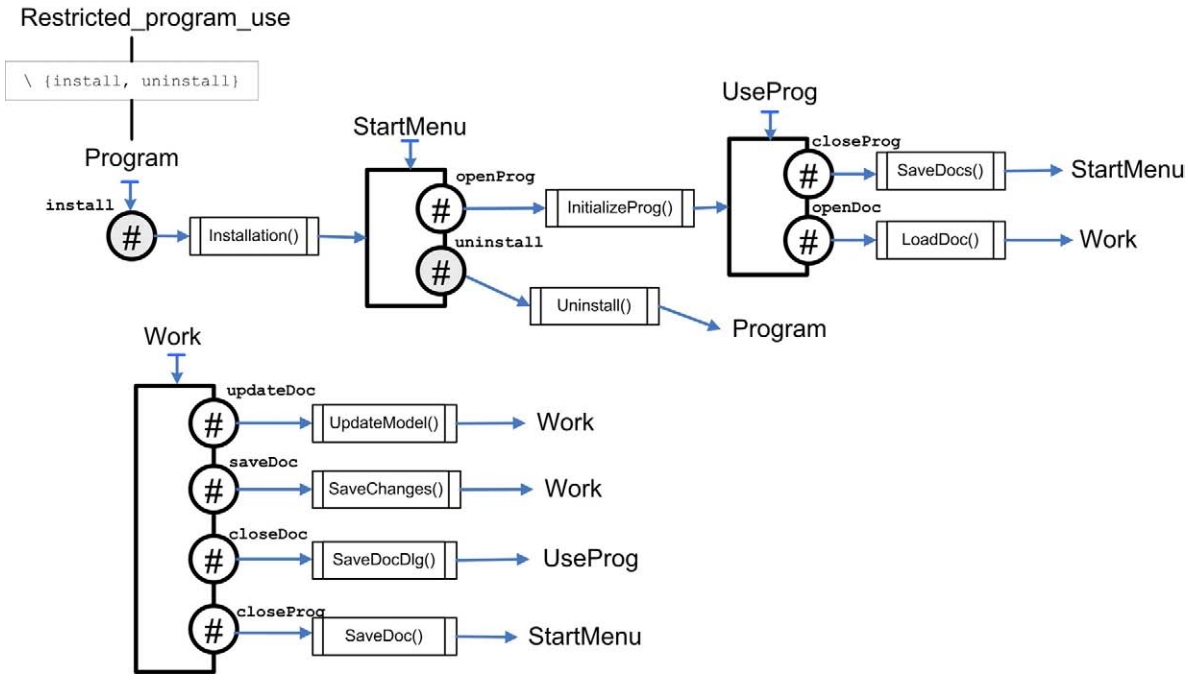

Figure 8 SystemCSP design used as an example for software implementation

The code is as follows:

```
Program(){
 switch (state){
  case START:
     status = install->sync();
     if(status == 0) return;
     elseif(status == 1){
        Installation();
        state = START_MENU;
```

```
    }
    else state = ERROR;
   break;
 case START_MENU:
     status = guardedAlt_StartMenu->select();
     if(status == 0) return;
     elseif(status == 1) {
         InitializeProg();
         state = USE_PROG;
    }
  else if(status== 2) {
      UninstallProg();
      state = START;
  }
  else state = ERROR;
  break;
case USE_PROG:
  status = guardedAlt_UseProg ->select();
  if(status == 0) return;
  elseif (status == 1) {
      SaveDocs();
      state = START_MENU:
  }
  else if (status == 2) {
      LoadModel();
      state = WORK;
  }
  else state = ERROR;
  break;
case WORK:
  status = guardedAlt_Work->select();
  if(status == 0) return;
  elseif(status == 1) {
      UpdateModel();
      state = WORK;
  }
  elseif(status == 2) {
      SaveChanges();
      state = WORK;
  }
  elseif(status == 3) {
      SaveDocDlg();
      state = USE_PROG;
  }
  elseif(status == 4) {
      SaveDocs();
      state = USE_PROG;
  }
  else state= ERROR;
  break;
```

```
case ERROR:
     printf(" process P got invalid status ");
  break;
}
```

In the constructor of the class defining this process, objects for the contained event ends and constructs are instantiated. For instance, the guarded alternative named StartMenu is on creation initiated using the offered event ends (openProg and uninstall) as arguments:

```
guardedAlt* StartMenu = new guardedAlt(openProg, uninstall);

EventEnd* openProg = new EventEnd(parentESP);
```

Code blocks are defined as member functions of a class that represent the process in which they are used. Code blocks that are used in more then one subprocess are usually defined as functions on the level of the component. Note that all code blocks (even a fairly complex sequential OOP subsystem that contains no channels, events and constructs) will be executed without interruption. Their execution can only be preempted by the operating-system thread of higher priority. As explained, user-level scheduling and function-call based execution engines are not fully preemptive. Thus, the events that need immediate reaction should be handled by operating-system threads of higher priorities.

3. Implementing CSP Events and Channels

Event ends are schedulable units implemented as state machines. They participate in the synchronization related to the occurrence of the associated event. This includes communicating their readiness to upper layers and waiting till the event is accepted by all participating event ends. This section describes in more detail how precisely this synchronization is performed.

3.1 Event synchronization mechanism

CSP events use the hierarchy of constructs for synchronization. An event end can be nested in any construct and it has to notify its parent construct of its activation.

In Figure 9, component C0 contains a parallel composition of components C1, C2 and C3 that synchronize on events a and b. Component C2 contains a parallel composition of C11 and C12 that synchronize on event a. The guarded alternative located in component C21 offers to its environment both events a and b.

Every process needs to export not-hidden events further to its environment, that is to a higher level synchronization mechanism. Every construct in the hierarchy must provide support for synchronizing events specified in its synchronization alphabet. This synchronization is done by dedicated objects – instances of the ESP (EventSynchronizationPoint) class (see Figure 10). The event-end will actually notify the ESP object of its parent construct about its readiness. A guarded alternative offers a set of possible event ends and thus instead of signaling its readiness to its parent construct, it can only signal conditional readiness.

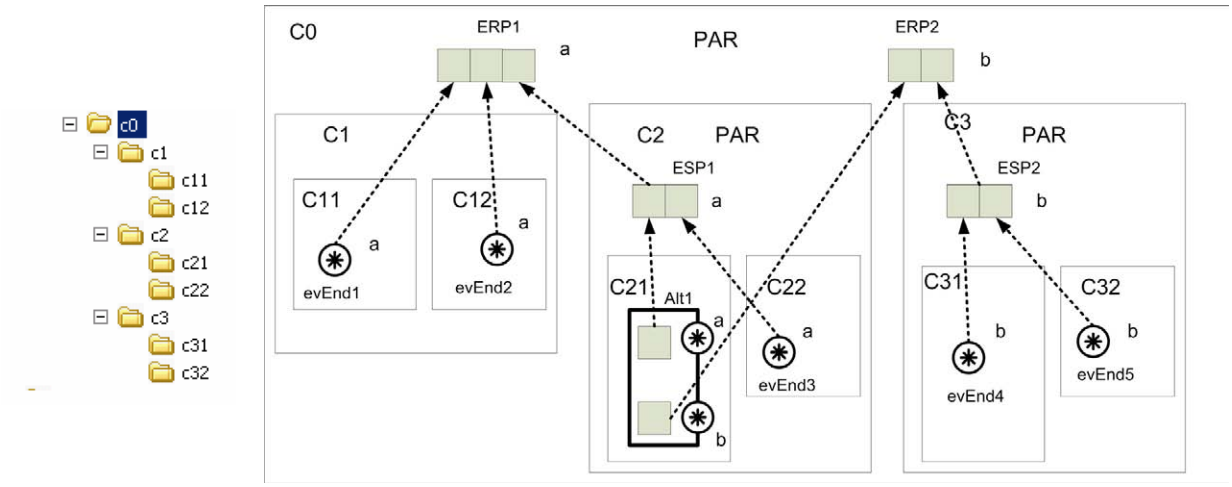

Figure 9 Hierarchical synchronization of CSP events

An ESP will, when all branches under its control are ready (conditionally or unconditionally) to synchronize on the related event, forward the readiness signal further to its parent ESP. When an event is not exported further, that construct is the level where the event occurrence is resolved. In that case, instead of an ordinary ESP object, a special kind of it exist (Event Resolution Point or ERP class) that performs the event resolution process. If some event ends are only conditionally ready, the ERP object will initiate a process of negotiation with the nested guarded alternative elements willing to participate in that event. When all event ends agree on accepting the event, ERP will notify all of them about the event occurrence.

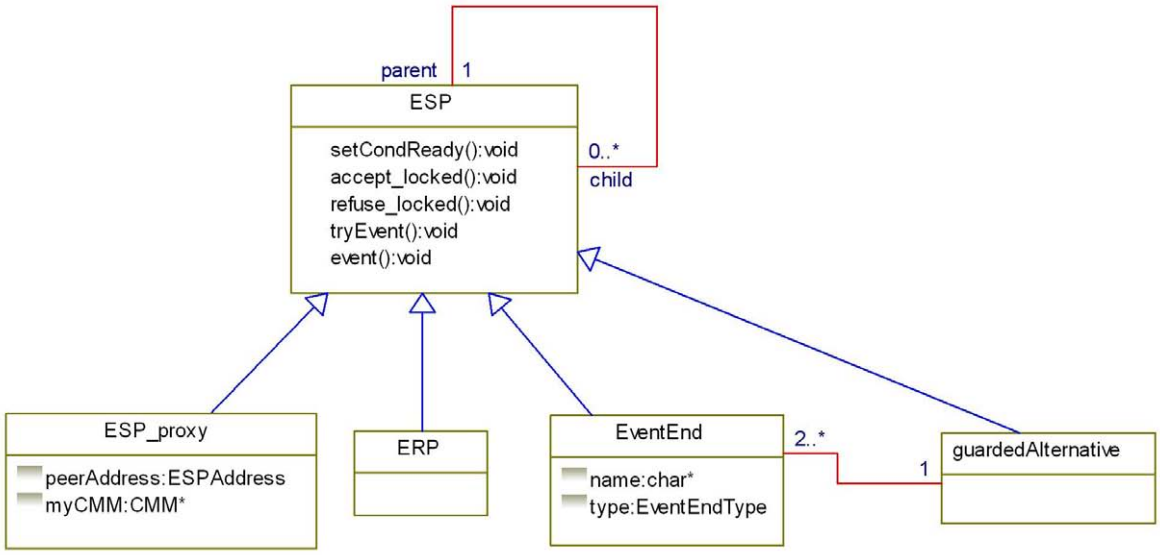

Figure 10 Event synchronization point classes

When on the top-level, in ERP, all fields, representing readiness of the associated branches, are ready or conditionally ready, a procedure of negotiation with sources of conditional readiness starts. This action results in every participating guarded alternative being asked to accept the event. If not previously locked by accepting negotiation with some other ERP, the queried guarded alternative will respond by accepting the event conditionally and locking till the end of the negotiation process. The attempt to start negotiation with already locked guarded alternative results in a rejection. In that case, the

conditional readiness of the guarded alternative is canceled for that event and the negotiation process stops. When all guarded alternative constructs participating in the negotiation process have accepted the event (and are locked - rejecting other relevant events attempts), the ERP declares that the event is accepted by notifying all participating event ends (including the guarded alternatives) about the event occurrence. However, after one of the involved guarded alternatives has rejected the event acceptance, the event attempt did not succeed and all involved guarded alternatives are unlocked. Guarded alternatives unlocked in this way do again state conditional readiness for those event ends for which it might have been canceled during the negotiation procedure.

The class hierarchy defining types and relationships between event synchronization points is illustrated in Figure 10. For every type of the negotiation message, the ESP class declares a dedicated function. In case of local synchronization, a parent and the related children ESPs communicate via function calls. In case that synchronizing parent/child ESPs are residing in different OS threads or nodes, the ESP_proxy abstraction is used.

In the table below, the list of exchanged messages is specified as an illustration of an attempt to synchronize participating event-ends in a scenario based upon the example from Figure 9.

Table 1 One synchronization scenario

source	destination	message
evEnd1, evEnd2	ERP1	Ready
ALT1	ESP1	Conditionally Ready
ALT1	ERP2	Conditionally Ready
evEnd3	ESP1	Ready
ESP1	ERP1	Conditionally Ready
evEnd4	ESP2	Ready
ERP1	ESP1	Try event
ESP1	ALT1	Try event
evEnd5	ESP2	Ready
ALT1	ESP1	Accept_locked
ESP2	ERP2	Ready
ERP2	Alt1	Try event
ALT1	ERP2	Refuse_locked
ESP1	ERP1	Accept_locked
ERP1	ESP1, evEnd1, evEnd2	event
ESP1	ALT1, evEnd3	event

3.2 Solving the Mutual Exclusion Problem

Let us assume that allocation of the application hierarchy from Figure 9 to the hierarchy of execution engines is performed as in Figure 11. Clearly, simultaneous access to variables, which is possible in the case of distributed systems and operating-system thread based concurrency, must be prevented while implementing the previously explained event synchronization mechanism.

Event synchronization is more or less a generalization of the synchronization process

used for channels. Let us therefore use channel synchronization as an example to show where the simultaneous access can cause problems.

In CT, a channel is a passive object. The process that first accesses the rendezvous channel will be blocked (taken out of the scheduler) and the pointer to that process thread is preserved in the channel. The process thread that arrives secondly will then copy the data and add the blocked process (one that has arrived first) to the scheduler. In CT, there is no problem of simultaneous access because the whole application is located in single OS thread.

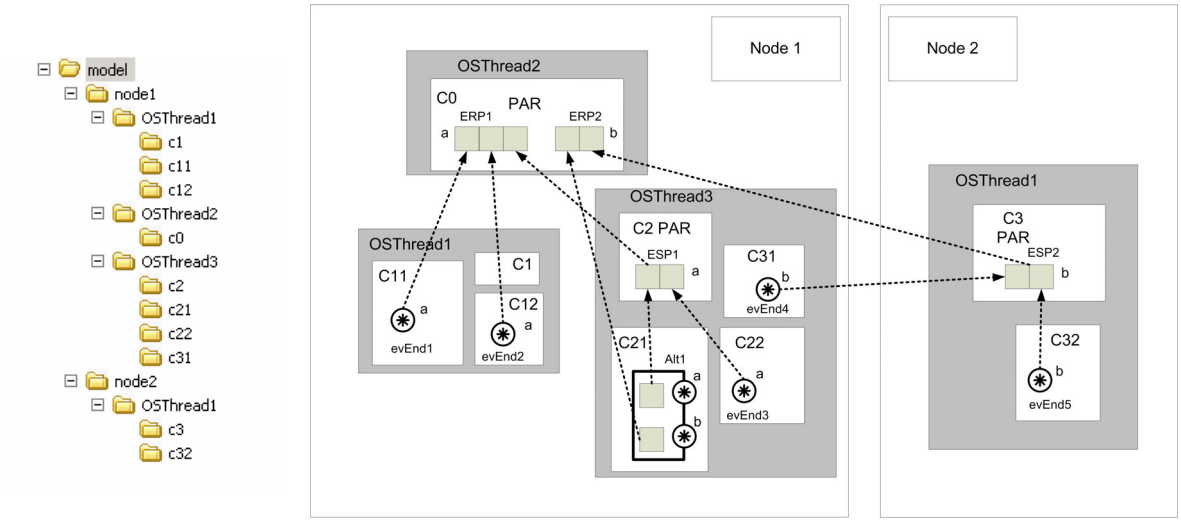

Figure 11 Synchronization of event ends allocated to different execution engines

In the SystemCSP framework, due to the possibility of using several OS threads as execution engines, protection from simultaneous access needs to be taken into account in order to make safe design.

Problematic points for channel communication when truly simultaneous access is possible are: (1) making the decision who arrived first to the channel and (2) adding the blocked process/component/user-level thread to its parent scheduler that can be accessed simultaneously from many OS threads.

Constructing a custom synchronization mechanism using flag variables is complex and error-prone. Besides, it is highly likely that such mechanism will fail to be adequate in case of hyperthreading and multi-core processors.

Using blocking synchronization primitives provided by the underlying operating systems causes the earlier mentioned problem of blocking all components nested in an operating-system thread that makes the blocking call. Besides unpredictable delay, this introduces additional dependency that can result in unexpected deadlock situations. It also does not provide a solution for an event synchronization procedure in case the participating components are located on different nodes.

If non-blocking calls, to test whether critical sections can be entered, are used, the operating-system thread that comes first can do other things and poll occasionally whether a critical section is unlocked. However, this approach makes things really complicated. For instance, the higher priority operating-system thread needs to be blocked so that the lower priority one can get access to the CPU and be able to access the channel. To block only the component, which accessed the channel and not the whole operating-system thread, one needs later to be able to reschedule it. For safe access to the scheduler from the context of another operating-system thread, another critical section is needed.

The previously discussed attempts to solve the mutual exclusion problem do apply

only for processes located in different OS threads, but on the same node. In essence, from the point of view of the mutual exclusion problem, an operating system thread is equally problematic as synchronization with parts of a program on another node. Thus, it is convenient if the solution for both problems relies on the same mechanism.

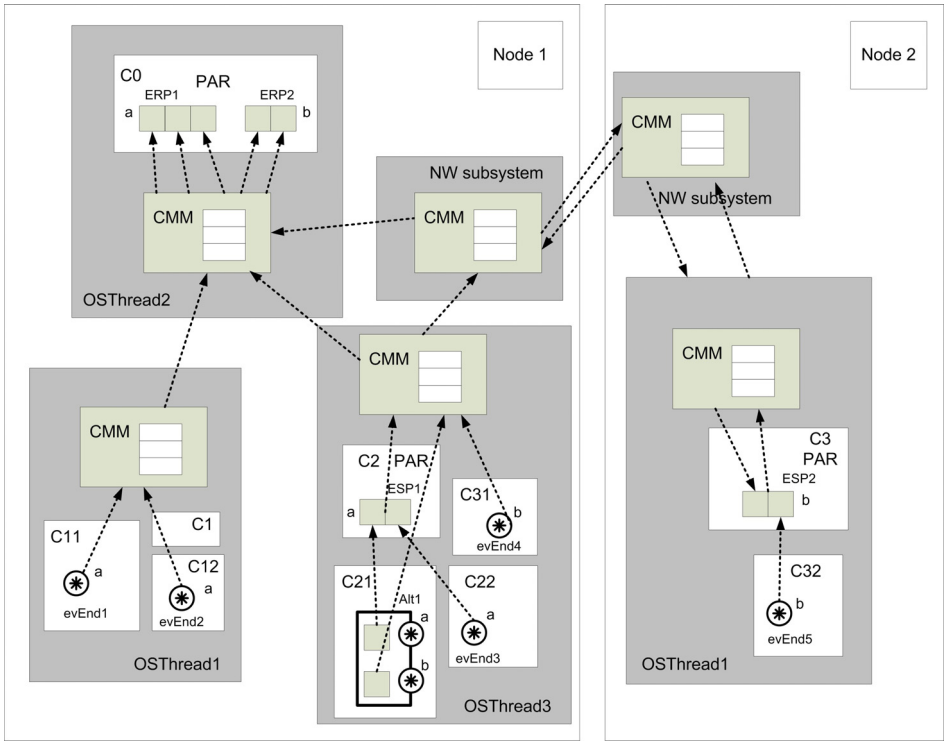

Figure 12 Using message queue based CMM to provide safe usage of concurrency

We propose that every operating-system thread has an associated message queue (operating systems provide message queues as a way to have non-blocking communication between operating-system threads). Thus, every OS thread, that interacts with other OS threads, will contain a control message manager (CMM) component that dispatches control messages (like event ready, event conditionally ready, try event, event accepted and similar) to message queues of other operating-system threads and transforms the received control messages to the appropriate function calls. For synchronization between nodes, networking subsystem can be located in a dedicated operating system thread that has a similar CMM component. This CMM will use the networking system to dispatch control messages to other nodes and will dispatch control messages received from other nodes to the message queues associated with CMMs of appropriate operating-system threads.

ESP_proxy (see Figure 10) communicates messages and addresses to local CMM, which further transfer it to the peer's CMM. The peer's CMM will then deliver the message by invoking direct function calls of appropriate ESP objects.

3.3 Channels Capable of Multidirectional Communication

Channels are special types of events where only two sides participate and in addition data communication is performed. As such, channels can be implemented in a more optimized way then events by avoiding the synchronization through hierarchy. Similar optimizations can be done for barriers with always fixed participating event ends, shared channels (any2One, One2Any) and simple guarded alternatives where all participating events are channels that are guarded only on one side.

One of the requirements (imposed by CSP as opposed to occam) for channels is that data communication can contain a sequence of several communications in either direction. A design choice made here is to separate synchronization from communication. To achieve flexible multidirectional communication, the part dealing with communication is further decomposed to pairs of sender and receiver communication objects (TxBuffer and RxBuffer) instead of using the template C++ language mechanism to parameterize complete channels with parameters specifying transferred data types, only RxBuffers and TxBuffers are parameterized. In this way flexibility is enhanced. Every channel end will contain an array consisting of one or more TX/RxBuffer objects connected to their pairs in the other end of the channel.

Since TxBuffers and RxBuffers contain pointers to the peer TxBuffer<T>/RxBuffer<T> objects, checking type compatibility of connected channel ends is done automatically at the moment of making the channel connection. This is convenient in case when connections between components are made dynamically during run-time. Otherwise, design time checks would be sufficient. Decoupling communication and synchronization via Tx./RxBuffers is also convenient for distribution.

3.4 Distribution/Networking

The CMM based design with control messages is straightforwardly extendable to distributed systems. In a distributed system, compared to operating-system thread based concurrency, besides control messages, also data messages are sent. Every node has a network subsystem with a role to exchange data and control messages with other nodes. The network subsystem takes control over RxBuffer and TxBuffer objects of a channel-end from the moment when the event is attempted, and returns control to the OS thread where the channel end is located after the data transfer is finished. This is done by exchanging (via the CMM mechanism) control messages related to location, locking and unlocking of data.

Of course, distributed event resolution comes with a price of increased communication overhead due to network layer usage. But, the task of the execution framework is to create conditions for this distribution to take place and the task of the designer of a concrete application is to optimize its performance by choosing to distribute on different nodes only those events whose time constraints allow for this imposed overhead.

4. Other Relevant Parts of the Software Implementation

4.1 Exception Handling

In SystemCSP, exception handling is specified by the take-over operator related to the interrupt operator of CSP. The take-over operator specifies that when an event offered to the environment by the process specified as second operand (exception handler) is accepted, the further execution of the process specified as the first operand (interrupted process) is aborted.

Upon the abort event (see Figure 13), the exception handler process is added to the scheduling queue of its parent component. Since the exception handler is a special kind of process recognizable as such by the scheduler, it is not added to the end of FIFO queue as other, ‘normal’ processes, but at its head. The preempt flag of the component manager is set to initiate preemption of the currently executing process. In that way, the situation where the exception handler needs to wait, while the interrupted process might continue executing, is avoided as much as possible.

As illustrated in Figure 13, the preempted process is appended to the end of FIFO queue of the component scheduler. If the preempted process is in fact the interrupted one then it will be taken out from the FIFO queue later during the abort procedure.

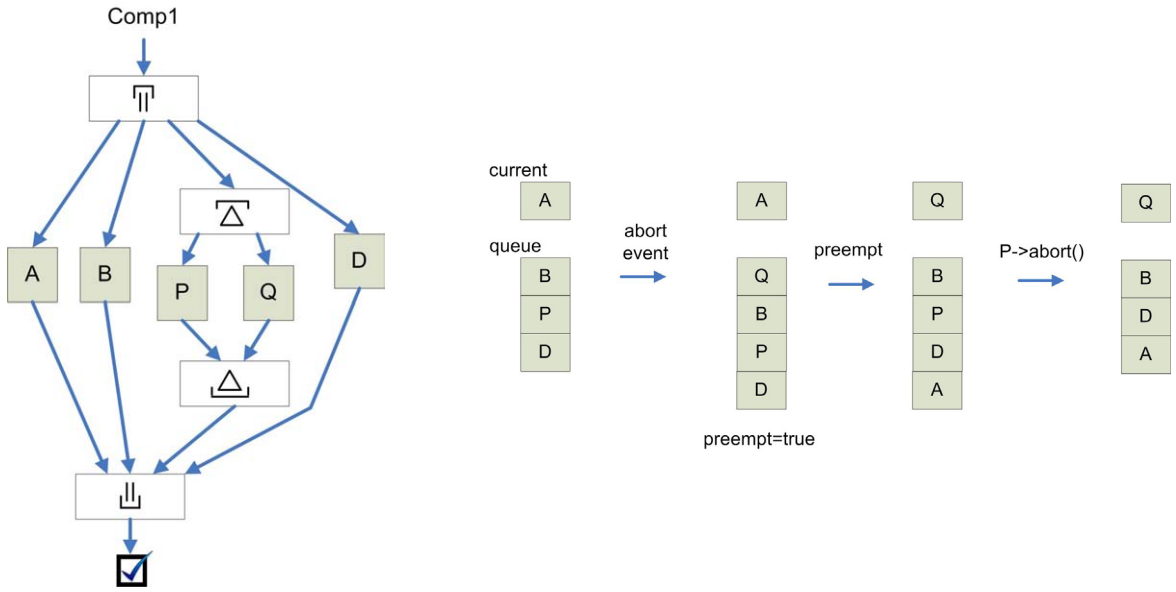

Figure 13 Example used to explain the implementation of take-over operator

The first step in the interrupt handler process is calling the abort() function of the interrupted process. The default version of abort() will cancel the readiness of all event ends for which the aborted process has declared readiness or conditional readiness. If the process is in the scheduling queue, it will be removed from there. Further, if the process is a construct, abort() will be invoked for all its subprocesses.

This exception handling mechanism does not influence the execution of other components that might have higher priority than the component where interrupted process resides.

4.2 Support for Development and Run Time Supervision

4.2.1 Logging

Logging is the activity of collecting data about the changes in values of certain chosen set of variables during some time interval. Not every change needs to be logged, but one should be able to use the obtained values to get insight in what was/is going on in some process/component. In this framework, the design choice is to allow logging only for the variables defined on the component-level. The main reason is obtaining a very structured and flexible way of logging that allows on-line reconfiguration of logging parameters. Thus all data constituting the state of the component should be maintained in the shape of component level variable. Every component can have a bit field identifying which of its variables are currently chosen for logging. The interface is defined that allows human operators to update this bit field at any time and thus change the set of logged variables.

Logging points are predetermined in design. In control flow diagram of SystemCSP, symbol used for logging point (a circle with big L inside) is associated with a prefix arrow as its property. The reason for this is a choice to treat a set of logging points as an optionally visualized layer added on top of the design. In implementation however prefix arrows do not exist, while logging points are inserted to the appropriate location in execution flow, as defined by the position of prefix arrow in the design.

Any logging point, either uses set of variables set for logging on component level using the described bit field mechanism, or defines its own bit field with set of variables to log. The operator is via the NodeManager allowed to inspect logging points and update their bit fields. Every logging point has a tag (or ID) unique in scope of its parent component, that is used to uniquely identify it. On the target side of the application, this tag can be a pointer to the object implementing the logging point. On the operator side of the application this tag is mapped to the unique ID of the logging point as specified in the system design.

The reason to opt for this kind of logging is predictability. The logging activity is considered to be part of the design and all the needed resources (e.g. CPU time, memory, network bandwidth and storage capacity) can be preallocated. Logging points can in design be inserted in such a way that it is possible to reconstruct change of every variable during the time. This approach to logging is considered here to be more structured and predictable then tracking every change for a chosen set of variables.

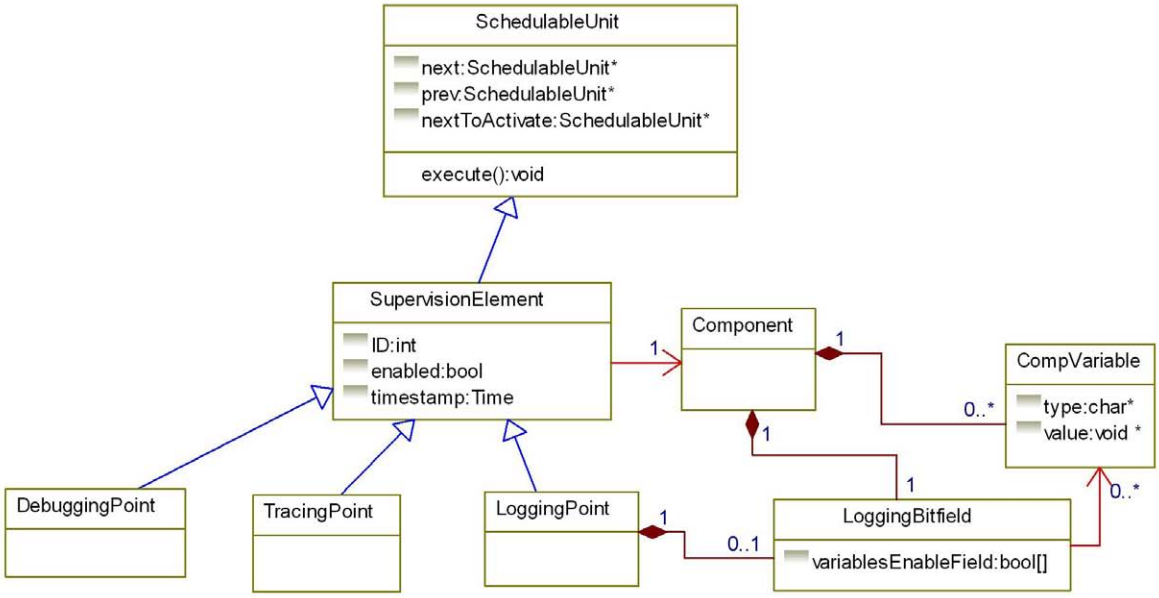

Figure 14 Supervision elements

4.2.2 Tracing

Tracing is an activity similar to logging. The difference is that instead of data, the information communicated to the human operator is the current position in execution flow of the application. Control flows leading to error states are always traced. Errors that are not fatal for the functionality of the system are logged as warnings. Other tracing points can be used for debugging or for supervising control. As it is the case for logging, the tracing is here considered to be part of the design and as such performed in predefined points of the execution flow.

SystemCSP defines a circle with a big T inside as a symbol of tracing point. Again it is associated with prefix arrow element, defining in that way the precise position of a tracing point. Every tracing point has a tag (or ID) that is unique per component and communicated to the operator to notify the occurrence of control flow passing over a tracing point. In addition, every function entry/exit is a potential tracing point.

5. Conclusions

This paper introduces design principles for the implementation of a software architecture that will support SystemCSP designs. The paper starts with explaining the reasons to discard the possibility to reuse the CT library as a framework for software implementation of SystemCSP models. The rest of the paper introduces the design principles for the implementation of the framework infrastructure needed in the software domain to support the implementation of a models specified in SystemCSP.

One of the main contributions of this paper is the decoupling application domain hierarchy of the components (related via CSP control flow elements and parent-children relationship) from the execution engine framework. In addition, this framework is constructed to allow maximal flexibility in choosing and combining execution engines of different types. In this way, flexible and reconfigurable component-based system is obtained. The priority specification is related to the hierarchy of execution engines and has thus become part of the deployment and not application design process.

Another significant contribution is solving the problem of implementing the mechanism for synchronizing CSP events in a way that is safe from mutual exclusion problems and is naturally suited for distribution. Besides that, the paper describes and documents the most important design choices in the architecture of the SystemCSP software framework.

Recommendation for future work is to fully implement everything presented in this paper. Furthermore, a graphical development tool is needed that will be capable to generate code. The described software framework would be used as a basic infrastructure that supports the proper execution of generated code.

References

[1] Orlic, B. and J.F. Broenink. SystemCSP - visual notation. in CPA. 2006: IOS Press.
[2] Roscoe, A.W., The Theory and Practice of Concurrency. Prentice Hall International Series in Computer Science. 1997: Prentice Hall.
[3] Welch, P.H. and D.C. Wood, The Kent Retargetable occam Compiler, in Parallel Processing Developments -- Proceedings of WoTUG 19. 1996, IOS Press: Nottingham, UK. p. 143 -166.
[4] Welch, P.H. The JCSP Homepage. 2007, http://www.cs.kent.ac.uk/projects/ofa/jcsp/.
[5] Hilderink, G.H., Managing Complexity of Control Software through Concurrency. 2005, University of Twente.
[6] Orlic, B. and J.F. Broenink, Redesign of the C++ Communicating Threads Library for Embedded Control Systems, in 5th PROGRESS Symposium on Embedded Systems, F. Karelse, Editor. 2004, STW: Nieuwegein, NL. p. 141-156.
[7] Tanenbaum, A., Modern Operating Systems. 2001.
[8] Chrabieh, R., Operating System with Priority Functions and Priority Objects. 2005.
[9] Sunter, J.P.E., Allocation, Scheduling and Interfacing in Real-time Parallel Control Systems, in Faculty of Electrical Engineering. 1994, University of Twente: Enschede, Netherlands.
[10] Orlic, B. and J.F.Broenink. CSP and real-time – reality or an illusion? in CPA. 2007: IOS Press.