7th IEEE International Conference on Industrial Informatics (INDIN 2009), Cardiff, UK, 24-26 June, 2009.

A Two Level Approach to the Design of Software for Cooperating Embedded Systems

Tony Larsson, Member IEEE
Halmstad University, Sweden
tony.larsson@hh.se

Abstract– **This paper proposes an architecture concept to the design of software, for embedded systems and cooperating embedded systems, aimed to simplify not only design but also reuse and reconfigurability. The concept is targeted at software for recurring signal processing and control tasks common in industrial embedded supervision and control applications. An important part of the architecture concept is that the design work is partitioned and described at two major levels. The first level is focused on definition of data structures and reentrant data mapping functions to be supported by a restricted use of an existing programming language. The second level, i.e. the system level, focuses on structural and behavioral concerns including: component encapsulation, connection of actions and components via ports. This level, also handling delays, triggers and actors, is supported by a system modeling language. The thus partitioned approach makes the different design and engineering concerns well separated which simplifies both analysis and verification of functional and time behavioral correctness, compared to if they are mixed and intertwined into each other. Another gain with the proposed architecture concept is that the number of operating system (OS) threads and associated overhead needed to share the processor or a set of processors in a distributed or multi-core case is minimized.**

Keywords: Embedded systems, embedded software, industrial informatics, component oriented design, distribute real-time control systems, software architecture modeling.

I. Introduction and Background

Both engineers and researcher have for a long time been attracted by the prospect to design embedded system software in the same way as many other engineered systems are designed, by help of: standardized and reusable components [1], partitioning the problem in separable concerns [2] and use of “programming in the large” concepts [3].

There are several examples of programming languages and execution models that address the system design problem in different ways. The synchronous program specification language Lustre [4], aimed for embedded signal processing and control systems, exemplifies a class of programming models focusing on synchronously clocked, discrete time step data flow processing. Other models such as communicating sequential processes (CSP) originally presented in [5] and calculus of communicating systems (CCS) presented in [6] are more aimed for coordination of distributed processes and especially their control flow and communication.

Traditional programming languages (including both procedural languages such as C and functional languages such as LISP and their more recent derivatives) are good for description of simple and composed data as well as procedural and functional mappings of data. Theses languages also support composition of individual functions into larger functions intended to be applied iteratively on larger data structures. For embedded systems where concurrent and sometimes distributed physical activities in real-time, with high requirements on dependability, are involved, programs however become harder to overview. Here description of aspects such as triggering conditions, timing, concurrency and distribution needs better and complementary approaches [7]. We argue that traditional programming languages is not suited for description of concurrent behavior and timing related system design issues. Instead a real-time and more system oriented design language should be used, see e.g. GIOTTO [8]. Object oriented and functional programming styles and architectural modeling frameworks like ROOM [9] also address parts of these problems. However, the description of entire software systems in a language mainly aimed for programming of data mappings still has its limitations.

There are many examples of popular programming languages extended with real-time primitives to “RT-languages”. A programmer using such a language needs to give timing related statements without a good view of its influence on the system level characteristics’. The programmer also gets involved in detailed description of how the different program parts shall set up their communication and implement data interaction protocols. We argue that this leads to an inside out way of describing a system where its design team risk lose the big picture.

An aim with our approach is not to design yet another language rather we aim at an architecture design concept enabling the use of conventional existing languages, methods and tools as far as possible. The information needed to implement communication between different parts of a system can be deduced from the connections and delays between actors and the triggered (and period constrained) activation of actors, this with help of a few timing directives, tools and operating system/platform support.

II. Separating Data Mapping and System Levels

Languages used for programming of industrial automation systems, i.e. defined in standards for Programmable Logic Controllers (PLC) such as IEC 61 131 and IEC 61 499 [10]

have many things in common with the concepts to be presented in this paper. A motivation to our study is that there are important semantic matters that need to be solved, this since PLC language based descriptions can be interpreted in several alternative ways or are unclear in existing approaches; for example see [11], [12], [13], [14] and [15]. The potentially severe consequences of small semantic discrepancies are also pointed out in [16]. For these reasons we intend to be as clear as possible regarding the meaning of the different concepts to be discussed in the following.

A. *Partitioning of Concerns*

The base for our proposed architecture concept is that one shall be able to use an ordinary programming language (PL) to define both data and functional mappings of data and restrict its use for this purpose only. Data and program constructs defined in such a way will have well defined structure and interfaces making them easy to replace and also feasible to verify. As a complement to the PL of choice we assume a simple system modeling language (SL) focusing on system architecture issues such as component composition, interconnection, feedback, delay and triggering of data mapping actions (where the data mapping functions are defined in a PL).

Our proposal is to some extent inspired by [17] proposing applicative state transition (AST) systems in which system state transitions occurs once per computation using the function application semantics of the language FP. Our proposal is also in analogy with the partitioning into concerns made when an OS, with its services, is used to handle problems that a PL does not cover itself. However, we argue that the use of OS services and their effect is not easy to keep track of and comprehend since the cooperation between user programs and OS services often are intertwined and distributed all over the code. Other component oriented design and architecture models more or less directly assume or imply some separation in two levels of concerns, such as components and systems, see [18] and [19] as well as kernels and stream composites such as pipelines, see [20].

B. *Programming of Data Mappings*

Since our focus in this paper is on the enhanced handling of system design issues we assume that the PL needed simply can be based on a subset of any functional language such as Lisp (e.g. Scheme or Common Lisp), Haskell or Erlang or the restricted use of a procedural programming language (such as Pascal, C, C++ or Java).

The PL is intended to be used to describe data structures and pure reentrant and thread-safe data mapping functions. Such functions do not own instance unique state or data that need to be maintained from mapping to mapping and thus will not be handled as data holding (object or task) instances.

The intention with this, OS and functional programming inspired, architectural decision is that the resulting approach significantly will simplify the use of formal transformation and verification methods. Another equally important intended gain with the approach is that the number and thus overhead of OS supported execution threads can be kept low if "state-full" objects and tasks are avoided at this architecture level. We thus assume that one can push the use of such state storage to higher levels of the system description, and at this level mainly use it when history reflecting feedback loops are required, see section III *B*.

Data transforming mappings in PL are defined as:

1) *Composition* of other more basic functions;

2) *Mapping table* from inputs to output;

3) *Algorithm* combining more basic functions in an iterative or tail-recursive fashion.

In order to prevent unexpected side effects, such function definitions only make use of its own temporary and local variables and do not have any hidden access to global shared variables.

III. System Modeling

The idea of system modeling and description by the use of a system language SL is to take care of system building concerns not supported in an intentionally restricted PL or in a more complete traditional PL. The basic architecture concepts in SL are ports, actions, components and connections. Other more advanced concepts such as actors activated by triggers and having delays and feedback loops will be discussed later in this section.

A. *Component Definition Composition and Connection*

With a component we mean an encapsulated composed software entity that only interacts with its environment via its input and output ports. A basic component is a named composition and connection of actions and/or components. To define such a component: a component-name and a set of external input and output ports are defined and associated with a description of the components internal design. The components can be described in three ways:

1) *Behaviorally* as an action or a set of actions;

2) *Structurally* as a composition and connection of other more basic components;

3) *Mixed* as a composition and connection of actions and components.

An action exist as a part of a component to perform a job comprising: a causal and time consuming functional mapping of the data valid before the mapping, available at a subset of the component's input ports and/or local (internal) connection variables; followed by a transfer of the mapped data to one of the component's output ports and/or local variables.

The behaviour of a component can thus be given a name and be defined as a set of actions; where an action is a composition of one or more functions applied directly or via each other to a set of input and output ports; and where actions can be series connected via internal connection variables. This is illustrated by the following example where a component `A`, is defined with an internal action consisting of two functional mappings f and g as follows:

```
Component A(a, b, c, d) =
    d = g(f(a, b), c).
```

Alternatively the component A can be described by two actions connected via an internal connection named x, as in the following description:

```
Component A(a, b, c, d) =
    Internal x,
    x = f(a, b),
    d = g(x, c).
```

In the last of the above two definitions parallel composition and series connection of two actions is expressed by two equations, sharing an internal connection variable x, and separated by "," to be read "and". The punctuation mark "." is used to end the definition. This component called A is illustrated graphically in figure 1.

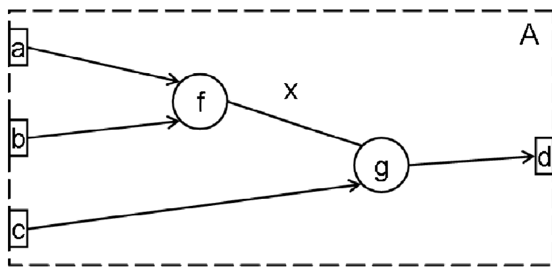

Fig. 1. Component called A defined by its internal action structure.

A component can alternatively be described as a composition and connection of instances of already defined or intended to be defined components and thus form a hierarchical composition structure, see figure 2 below.

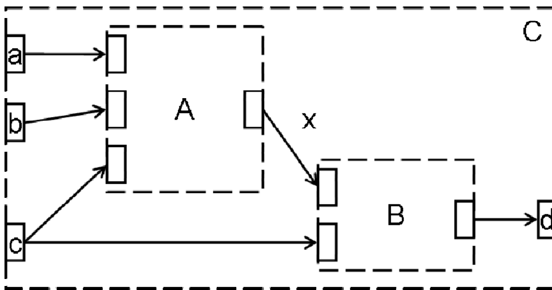

Fig. 2. Component called C, defined by its internal composition and connection structure.

Using textual description the component C, illustrated above, can in a declarative fashion be defined as follows:

```
Component C(a, b, c, d) =
    Internal x,
    A(a, b, c, x),
    B(x, c, d).
```

In order to run on a sequential machine and provide correct semantics such a declarative component description need to be expanded and translated into a data flow dependent and "scan ordered" sequential code. The correct semantics requires that the evaluation order sees to that all inputs to a following function or component are evaluated first to eliminate the need for storage between executions. For example: in figure 1 function f is executed before function g and in figure 2 component A is executed before component B.

To get more specific knowledge about the behaviour of component C, one can unroll the "black box" encapsulation and expand the two components called A and B to see their internal definitions. Depending on the behaviour of components such as A and B, component C may be behaviourally equivalent with some other component X, defined as in the first example, as an action structure.

For more complex structures, such component unrolling may have to be performed in several steps until all internal behaviour defining action structures are exposed. A consequence and drawback with too many levels in a hierarchical description, and thus of component reuse, is that the deduced composed behaviour may be hard to grasp.

The input and output ports of component instances and actions within a composed component can be connected to each other via internal connection variables and to external input and output ports. The connections can be one-to-one from an output to an input port, or one-to-many from an output to several input ports. Output ports can not be connected to each other. However, explicit multiplexing or data fusion components can be inserted between the original sources and sinks to enable for multiple actions or component instances to act as one joined source and share a producer port in order to collectively deliver their contributions to a consumer or a set of consumer ports.

The data that is exchanged via ports and connections are treated as messages containing:

Single scalar values such as simple data samples;

Complex data structures such as aggregates of samples packed in frames, vectors or matrices;

Formatted packages (e.g. type-length-value (TLV) tagged) assembling several different data elements.

For correct system composition the data types of the ports and connections that are connected must be compatible. We assume that the techniques used for type declarations can be inherited from the PL. We also assume that the type information needed can be deduced from the declarations made in the PL.

Both input and output ports act merely as reference symbols until connected. Semantically, a connection driven by an output port can be thought of (and implemented) as a variable, a register or a memory position. Thus a connection is assumed to keep its data stable and unchanged until updated by a new action. The execution of an action causes a basic execution delay dependent latency. In addition a synchronization delay is caused by the period between the events that trigger the action to execute new incoming data samples or packages. These timing related issues are discussed further in the next two subsections: the first

subsection deals with actor and trigger concepts while the following focus on the handling of synchronous delays and feedback loops.

B. Actors and Triggers

An actor is a task, encapsulated and named as a component, to be used in one or more instances at the system level. An actor is associated with an activation condition called a *trigger* that defines a time baseline shared by all internal actions and component instances enclosed within an actor. Triggers are activating time or event[1] conditions described as logical or conditional expressions on data (available via one or more input ports) which become true after having been false or vise versa. Triggers are assumed to be handled by an OS and used to control the timing and coordination of the specified actors' execution and output flow and may occur in periodic, aperiodic or sporadic[2] (optionally phase delayed) time intervals. To reduce dependencies between different actors, simplify timing analysis, and enable proper scheduling of actors as well as communication between actors; each actor has to provide stable and deterministic synchronously sampled input ports and delayed updating of its output ports.

The following are examples of simple trigger attributes, stated within curly brackets: {`Actor G On Y Periodic 100 Phase 20`}, {`Actor H On Z Sporadic 100`} or {`Actor I On Y Aperiodic`}. Periodic and Sporadic execution shall be performed within their specified real-time requirements while Aperiodic execution requests can be accepted only when the overall utilization and schedule permits. In addition to trigger attributes one can in a similar way envision an actor to be associated with other advising attributes such as configuration *Mode* and *Processor* allocation attributes.

Functional data mappings described in PL are to be associated with their estimated worst case execution time (WCET) before proper timing assignments can be made. Assuming allocation to one single processor the total sum of WCET of the mappings to be performed within an actor needs to be shorter than the shortest period between the actors' triggering events. Running more than one actor on one processor requires each actor to execute within its utilization share of the processor's full capacity.

C. Delay and Feedback Loops

All components and actions that are part of an actor, performing actions, i.e. functional mappings[3] of input data to output data available at its ports, are intended to be executed once at each activation and work as atomic transactions. When activated they will start and execute to completion, while their input data remain unchanged (meaning that they are sampled just before the start of the work). To guarantee that all possible consuming actors, acting as receiving data sinks, triggered with different periods or other activation events will see the same result, all output ports of a producing actor are assumed to be synchronized and the results delivered at the end of the producing actor's period (or its guarantied deadline). An actor working as receiver of such output results will thus get access to these results synchronized with its next triggering event, "reshaping" the otherwise undeterministic (WCET limited) execution delay into a synchronized and deterministic delay.

We thus do not consider output data to be valid just because its execution has finished. Many signal processing and control applications can be robust to small differences in delay and thus also to different implementation of the intended semantics. However, when it matters the difference must be observed. The dissimilarity also relates to the important difference between the Mealy and the Moore state machine models. Control state information should be delayed according to the second case above. In the case of a Mealy machine model, all other output values of a component are allowed to depend directly on the input values according to the first case above.

To enable deterministic execution behavior within given deadlines, all feedback connections between output ports and input ports must include a synchronizing delay. As disused above and to simplify reasoning about the resulting behavior when several actors' cooperate all their output connections also must include such a delay, in effect enforcing Moore machine design, an example of this is illustrated in figure 4.

Since we have, as stated previously in subsection II.*B*., restricted actions and components used at lower levels to performing reentrant tread-safe, state less, functional data mappings; the state holding capacity of connections and feedback coupled connections as well as the handling of delays at the actor level instead becomes very important. Our purpose with this approach is to make explicit important sequential behavior influencing state information and feedback loops that otherwise would be hidden deep inside functions, actions and components; and often also at several design levels.

To handle the description of time delay relations, both in text and in graphical notation, we introduce two temporal offset operators written {!, ?} where:

`!x` stands for next value at `x,` after triggering delay

`?x` stands for previous value at `x,` before triggering delay

These temporal operations neutralize each other as defined by the following two axioms, stating:

```
?!x = x                                   (1)
!?x = x                                   (2)
```

Observe that one can think about these temporal offset relations either in terms of individual values or as applied to streams of values (it is the referred to values available at ports that are shift back or forth in time). Delay relations, defining temporal offsets are, in graphical form, symbolized with a black bar as in the figure 3:

---

[1] An event is a transition or change of data from one value to another.

[2] Aperiodic events with a least inter-arrival time, are called sporadic.

[3] Mappings are implemented as a function composition, a table lock up or as an algorithm, running on a sequential CPU or even on multi-core or parallel CPUs.

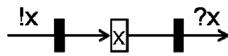

Fig. 3. Delay relations with references both to previous values at x written ?x and next values at x written !x, before and after the delays for a port x.

For an example of its use, see figure 4 illustrating a component F forming an actor composed by three components B, E and D connected in a mutual feedback loop. The actor also includes synchronizing delays, illustrated as above. The length of such synchronizing delays is defined by the actual periodic time interval between their associated triggers or by the deadline after sporadic or aperiodic triggering events.

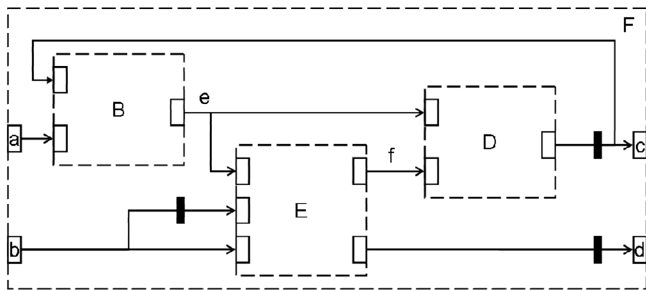

Fig. 4. Component described on actor level with trigger attributes and synchronizing delays, shown explicitly.

```
Component F(a, b, c, d)(e, f) =
    Internal e, f,
    B(c, a, e),
    E(e, ?b, b, f, !d),
    D(e, f, !c).
```

The component E is using both the current and the previous values available at a port b and produces the next value at port d. The component D is producing the next value at port c while component B uses the current value at port c.

Data validity can be guarantied at each occasion when the actor is triggered if the activation is periodic or sporadic. Activation conditions and other run-time attributes are included in a list of declarations associated with actors, given as follows: {Actor F On X Sporadic 100 Mode A Processor P}. Such a list of advising declaration attributes can be added late to a component specification, to make it an actor, or even latter at system design time, but must of course be possible to satisfy with respect WCET characteristics. Observe that attributes values can be dynamically set by an output port and read via an input port. Available processors, sensing, actuating and communication resources can be specified in a similar attribute list.

To simplify formal reasoning we can as we have shown above explicitly state synchronizing delays, similar to the synchronous approach [4], using a delay operator. An alternative would be to rely only on an implicit and assumed scheduling semantics, e.g. based on that the different sub components of the actor component and their internal functional mappings are executed only once per triggering event in the correct causal order and that values at input and output ports are generated simultaneously and synchronously at the end of an actor's period or deadline. The use of explicit delay expressions, as we propose, enables to describe more elaborate timing difference relations and also allow an output to control an external physical resource directly, immediately after its execution delay, i.e. without the enforcing an implicit triggering delay. But of course, such an added flexibility and expression power also can give rise to problems if not used with intended meaning and care.

There are several attempts to define unclear parts of the semantics for industrial control languages, recent examples are the references' [11], [12] , [14] and [15] that proposes a sequential semantics for IEC 61 499 function blocks based on applying a sequential execution order, where each action of each function block is executed once per triggered activation and with direct, once executed, delivery of their resulting output. Other similar approaches to define the semantics of industrial control languages are discussed in [21] and [22].

An important motivation mentioned in the introduction of our approach is that it makes important timing related information, such as triggers and delays, explicit and transparent and that this in effect also simplifies the process needed to translate a component expression to a form that enables symbolic substitutions, transformation and reasoning, for example the values at output port c at time t is some function f of the values available at input ports a, b, and c at the previous (trigger) time t-1, expressed as $c_t = f(a_{t-1}, b_{t-1}, c_{t-1})$. The actual definition of such a function f can be deduced by expansion of the enclosed components.

Our proposed approach is, as presented so far, mainly oriented towards streaming signal processing and industrial control applications where data is passed by value between actors and the components within an actor. However, there is, also in this application domain, a need to pass not only values but also reference and index information to be able to deal with large information amounts handled as vectors, matrices or database like structures.

The functional mappings' performed within an actor, specified in actions and/or components, can for example get access to a state holding database, via a message containing a reference variable and difference or query information that may change or request state information kept in the database, see fig. 5.

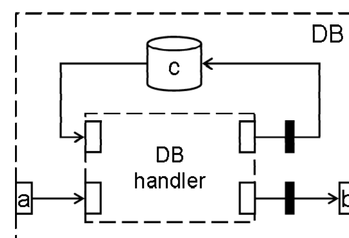

Fig. 5. Actor component with an internal feedback loop connection acting as a database.

A system can be described as a set of cooperating actors much in the same way as lower level components, i.e. as a composition and connection structure, see figure 6. Observe, synchronizing delays are hidden encapsulated within the actors and thus not visible at the system level. This is motivated by the relation between delays and the trigger conditions for the actors. For this reason we do not push delays to the outermost system level.

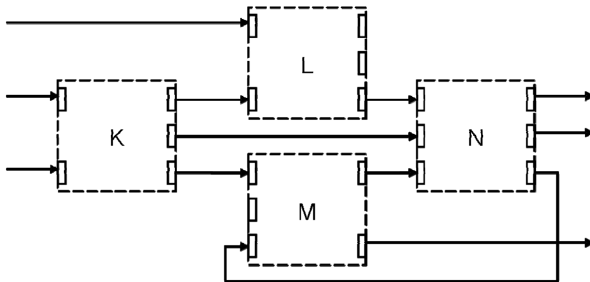

Fig. 6. System described as four connected actors, with all delays and possibly also some more feedback loops encapsulated within the actors.

D. *Execution Control and Priority*

The approach proposed and presented in this paper relies on the use of a traditional OS to handle the activation of actors. Where actors when required are partitioned and distributed as several task instances executing on different processors. Once activated by its OS an actor's or task's behavior can be influenced and controlled by events, states and conditions for example handled by a state machine, described as a feedback loop. Very control state dependent and sparse activation of an actor can be achieved by a guard part of an actor or by a guard actor working like an interrupt service routine creating a triggering event for the actual actor.

This more traditional OS supported approach is somewhat different to the IEC 61499 idea of a more distributed control handled by interconnected event/condition interfaces and execution control charts (ECC), for example see [11] and [13]. A first drawback with the IEC 61499 idea of distributed control is that it can lead to intricate dependencies between control and data mapping behaviors and thus verification problems; a second is that most implementations still relies on an OS and the thus following dual responsibilities between ECC and OS must be well defined and comprehended by all involved designers. Our idea is to have the overall timing behavior defined at the actor level, this in order to reduce and make temporal control dependencies as transparent as possible to designers.

All components and actions part of the same actor has the same priority and can be handled by an OS as one process, task or thread per processor on which they are allocated. Some parts of an actor can be allocated to one processor and other parts to other available processors, each handled as an individual process, task or thread.

An actor can in principle also be partitioned in several instances to deal with different segments of a large data structure on different processors that each run as loosely coordinated processes, tasks or threads, depending on the supporting OS.

The priority of an actor can be based on an assigned application specific value or be deduced (in a static or dynamic fashion) from: required execution period, least laxity (slack) or least time left to deadline.

The preemption or interrupt of a lower priority actor caused by the triggering of a higher priority actor shall from a pure computational point of view, not change its effect. A higher priority actor is thus allowed to preempt a lower priority actor, given that specified timing constraints still can be satisfied.

E. *Reuse and Reconfiguration*

We have decided to allow components to be defined as a combination of both (named) components and (unnamed) actions for the following reasons:

1) Designers want to reuse previous designs and therefore create or get access to a library of data mappings defined in a PL and a library of components defined in a SL;

2) Designers will not name and encapsulate action structures as components *unless they intend to reuse* them.

A system requiring its behavior to adapt to different applications or missions can be described as a set of alternative *mode dependent configurations* or structures that are setup with different composition and connection of its actors and/or their internal component instances and actions. Such a system can thus reconfigure and change its behavior, including the timing or triggering attributes of actors in a controlled way between different modes of operation. This of course requires that all configuration modes are analyzed and verified as viable from a system timing point of view.

During the switch from one operation mode to another, data is assumed to remain unchanged at all actors' output ports. After reconfiguration a system may utilize the old data held on these output ports as new input or disregard it as no longer relevant and instead initialize its inputs with new mode or state dependent data values. Input ports are thus sampled after the reconfiguration that thus is performed as an atomic transaction between the sampling of output and input ports.

Mode changes effected by reconfigurations are assumed to be made within realistic system boundaries with a fixed set of physical input (sensor and communication channels) and output (actuator and communication channel) devices. The connections, data formats and other interface defining characteristics between major subsystems implemented as actors are also assumed to be fixed.

Typically what is supposed to be changed between configuration modes is only the functional or timing related behavior. The functional parts are typically changed to provide for a richer set of services and the time related parts to provide for different quality of service characteristics or just to get the interwork between involved actors feasible.

We assume that our approach can be extended and made more adaptive, if needed, but so far our ambition is mainly to give support for reconfiguration and thus we have not set the

goal to the more advanced level of adaptation to dynamically changing needs and environments that require intelligent agent controlled distributed system reconfiguration, discussed for example in [23] and [24]. However, the actors' constituting an application can, as we see it, be made as intelligent as needed also with our current ambition level.

IV. Implementation Considerations

A. *Design-Time Tools and Execution Platform Support*

To create an executable system from an SL description requires some analysis and engineering efforts, expected to be supported by generally available analysis and design tools. For example, one must as in other approaches, estimate and accumulate the WCET of an actor component's internal structure and parts to be able to decide feasible execution periods and deadlines and related priority assignments. In a distributed case further also allocate communication resources and time in relation to the amount of data that need to be transferred between actors (optionally partitioned into task instances allocated to different processors).

The execution of a system described in SL, in addition to such design-time analysis tools, requires support from an execution platform (EP) that provide memory for storing of data at the system's ports' and/or connections', and also perform the actors' with specified, functional mapping, ordering and timing behavior. Such an EP, for example based on an embedded RT OS, must in case of use in an cooperating embedded system also support distribution and communication over processor boarders, this by utilizing inter process communication (IPC) services. If the EP is a distributed system platform the IPC services can be provided as part of a middleware layer also providing fault tolerance services (e.g. for failover and data replication).

The EP is supposed to ensure that the data produced by an actor component allocated to one specific processor and connected to one or more actors acting as consumers allocated to other processors will be transferred to and buffered on the receiving processors in a preconceived way. This is viable since all communication resource needs can be estimated, allocated and controlled based on information deduced from connection and timing constraints specified in a declarative fashion.

Further, in order to deal with communication disturbances and resulting data correctness or timing faults, we assume that:

1) The IPC mechanism with very high probability will eliminate any effect of such problems;

2) The system designer will provide solutions for fault tolerance (for example by replication of important data and use of failover and recovery mechanisms).

High requirements on dependability can be satisfied by replication of data on duplicated connection structures and internal connection variables, handled by replicated actors. Other more declarative alternatives, aimed for automatic synthesis, are to:

1) Define important connections as replicated and solve the replication issue in a standardized way using design tools and support from the EP;

2) Use aspect oriented design techniques and define important connections as join points where replication functionality can be inserted by use of code weaving.

From a scalability point of view it is assumed that an EP can be built both for a small embedded system having a single processor with constrained memory as well as for a larger cooperating embedded system running on a distributed processor and memory system with support for IPC over heterogeneous communication networks. A challenge in such a distributed context is the handling of large data structures that for performance or size reasons may need partitioning and distribution of the data over more than one processor and memory node. Partial results arriving from several nodes must then be possible to join and merge into common data structures and storage locations allocated for handling of the total result.

B. *Semantic Assumptions*

The actions of an actor are supposed to be executed only once for each triggering of the actor. The time needed to perform an action causes a corresponding execution latency delay. A feedback loop from output to input of one action or a chained or mutual feedback loop involving a set of connected actions is executed once per activation in order to implement expected behavior, resulting in one de facto synchronizing delay within the feedback loop. However, as stated previously synchronizing delays must be inserted and thus declared explicitly in all feedback loops and on all actor output ports.

To enable deterministic communication between actors, where each one is associated with different triggering conditions, the following assumptions are made:

1) Consecutive data elements from a producing actor triggered with a faster rate can be aggregated and buffered before input to a consuming actor. Slower rate consumers are in this case assumed to have ability to select among elements in each data packet or between the buffer stages.

2) A fast producer can produce data to be used by several consumers, working at different rate, in this case consumers use the input data sampled at their activation.

3) Data from a slower producer represent the latest and best information from this producer and may be used as valid in several consecutive periods by a faster consumer.

4) To keep track of the number of data values delivered by a producer (e.g., to allow accumulation or averaging of such values) the producer can mark the number of values or provide a counter as a parallel output port for this purpose.

5) To reduce the probability of meta-stable conditions:

a) Harmonic relations between periods are preferred;

b) All external aperiodic events are double buffered.

Additional synchronizing delays can be inserted by a designer or a design tool as to simplify the partitioning and allocation of an actor on multiple processors and also to enable or simplify the communication between different

distributed tasks implementing the actor. For example in cases when the overall WCET of an actor is too long for one single processor to handle within its deadline. Of course such partitioning has an influence on the temporal semantics and can only be made when such additional synchronizing delay, resulting in a pipelined computation, can be accepted.

V. Conclusions

Our main contribution in this paper is to make a division between two types of design concerns supported by *1)* a restricted programming language PL used to describe data structures and data mappings and *2)* a system language SL used to describe the composition and interconnection of actions, components and actors via ports, delays and feedback loops. Information given in PL and SL descriptions can be analyzed and used to setup a system's actors' execution and communication intervals and triggering events that then are implemented by OS and middleware services.

The proposed approach simplifies the design, reuse and reconfiguration of software for cooperating embedded systems by providing a clear separation in concerns and by pushing delays and feedback loops, causing state full behavior, to the actor and system levels. Further, the OS supported time and event triggered execution of actors that can be partitioned and distributed over processors is a model that partly resemble some other software component and architecture description related proposals as mentioned in the introductory parts of this paper and can to some extent also contribute with a complementary and partly alternative, semantic model for industrial automation languages such as IEC 61131 and IEC 61499. The embracing would require the responsibilities between connected ECCs and OS services to be investigated and clarified further, and potentially influencing a future updated version of the IEC 61 499 standard.

Acknowledgment

The work is done in the research profile CERES as part of the EPC project founded by Swedish Knowledge Foundation.

References

[1] M. D. McIlroy, "Mass Produced Software Components", NATO Software Engineering Conference, October 1968", Ed. P. Naur and B. Randell, NATO, Brussels, 1969, pp. 138-155.

[2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", Communications of the ACM, vol. 15, no. 12, December 1972, pp. 1053-1058.

[3] F. DeRemer and H. Kron, "Programming in the Large versus Programming in the Small", Proc. International Conference on Reliable Software, Los Angeles, California, 1975, pp. 114 – 121.

[4] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE", Proceeding of the IEEE, vol. 79, no. 9, September 1991.

[5] C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM, 21(8), August 1978, pp. 666-677.

[6] R. Milner "A Calculus of Communicating Systems", Springer Verlag, LNCS 92, 1980.

[7] E. A. Lee, "Embedded software", Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[8] T. A. Henzinger, B Horowitz, and C. M. Kirsch, "Giotto: a time-triggered language for embedded programming", Proc. of the IEEE, vol. 91, no. 1, January 2003, pp. 84-99.

[9] B. Selic, "Tutorial: real-time object-oriented modeling (ROOM)", Proceedings IEEE Symposium on Real-Time Technology and Applications, June 1996.

[10] J. H., Christensen, "Basic Concepts of IEC 61499", Conference "Verteile Automatisierung", Distributed Automation, Magdeburg, Germany, 2000, pp. 55-62.

[11] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications", 2nd IEEE International Conference on Industrial Informatics, INDIN '04, 2004.

[12] V. Dubinin and V. Vyatkin, "Towards a Formal Semantic Model of IEC 61499 Function Blocks", Proc. 4th IEEE Int. Conf. on Industrial Informatics, INDIN, 2006.

[13] V. Vyatkin, "Execution Semantic of Function Blocks based on the Model of Net Condition/Event Systems", IEEE International Conference on Industrial Informatics, INDIN, 2006.

[14] Vyatkin et al, "Alternatives for Execution Semantics of IEC 61499", Proc. 5th IEEE Int. Conf. on Industrial Informatics, INDIN, 2007.

[15] V. Vyatkin and V. Dubinin, "Sequential Axiomatic Model for Execution of Basic Function Blocks in IEC61499", Proc. 5th IEEE Int. Conf. on Industrial Informatics, INDIN, 2007.

[16] P. Caspi, N. Scaife, C. Sofronis and S. Tripakis, "Semantics Preserving Multitask Implementation of Synchronous Programs", ACM Transactions on Embedded Computing Systems, Vol. 7, No. 2, Article 15, February 2008.

[17] Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs", Communications of the ACM, Vol. 21, pp. 613-641, 1978.

[18] Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Proc. COMPSAC'97, Washington, DC, Aug. 1997, pp. 6-13.

[19] D. Garlan, R. T. Monroe and D. Wile, "Acme: architectural description of component-based systems", Foundations of component-based systems, Cambridge University Press, USA, 2000, pp. 47-67.

[20] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications", Proc, 11th Int. Conf. on Compiler Construction, 2002, pp. 179 – 196.

[21] C. Sünder C., A. Zoitl, J. H. Christensen, H. Steininger and J. Fritsche, "Considering IEC 61131-3 and IEC 61499 in the context of Component Frameworks", Proc. IEEE 6th Int. Conf. on Industrial Informatics (INDIN 2008), Daejeon, Korea, July 2008.

[22] C. Sünder, I. Gosetti, V. Vyatkin, and B. Favre-Bulle, "Comprehensive Formal Description of IEC 61499 Control Devices", 6th IEEE International Conference on Industrial Informatics, INDIN, 2008.

[23] R. W. Brennan, M. Fletcher, and D. H. Norrie, "An Agent-Based Approach to Reconfiguration of Real-Time Distributed Control Systems", IEEE Transactions On Robotics and Automation, vol. 18, no. 4, August. 2002.

[24] R. W. Brennan, "Toward Real-Time Distributed Intelligent Control: A Survey of Research Themes and Applications", IEEE Transactions on Systems, Man, and Cybernetics, vol. 37, no. 5, September 2007.