A COMPONENT MODEL FOR EMBEDDED REAL-TIME SOFTWARE PRODUCT-LINES

Anders Wall and Christer Norström

Mälardalen Real-Time Research Centre, Department of Computer Engineering
Mälardalen University, Västerås, Sweden
{anders.wall, christer.norstrom}@mdh.se

Abstract: This paper proposes a component model suitable for use in the development of embedded real-time systems where resources, such as memory and CPUs, usually are very limited. A precise sematics for this model is defined. A precise semantics is necessary to enable architectural analysis of systems specified with such a model. A typical example of such analyses is scheduling where the temporal correctness is verified. The model is constructed with software product-lines in mind. Thus an essential part is how to specify and verify flexibility in the components. The model proposed in this paper is also independent from its infrastructure, i.e. operating system. This since the model makes no assumptions about its environment with regards to task models and component infrastructures such as name servers or object request brokers.

Keywords: architectures, components, embedded systems, systems design, real-time

1. INTRODUCTION

Product line architectures are an important research area in software engineering since time to market is becoming more and more essential to make the business successful. For example, delaying an introduction of a new mobile phone on the market may cause huge income losses and loss of market shares. The basic idea behind product line architectures is to create a generic architecture that can be tailored for different members of a product family (Dikel, *et al*., 1996; Bosch, 2000). The tailoring can be achieved by parameterization of generic components or by provide specific implementation for non-generic parts. Components that must be specifically implemented for a product will be referred to as *abstract components*. To use this concept for non real-time systems is a challenge, and an even bigger challenge is to employ this concept for embedded real-time systems that most often are implemented on hardware that is very limiting in terms of memory- and computational resources. Thus, the component infrastructure provided by industrial state-of-the-art, e.g. CORBA, Jini, DCOM, cannot be used. Only the component-based software engineering philosophies can be adopted in the small- embedded real-time systems community.

To be able to adopt the product-line architecture approach in real-time systems it requires that the component model support the specification of real-time attributes. Furthermore, the component model must also support specification of timing attributes both on concrete and abstract components. The reason for requiring support for specification of timing constraints on abstract components is to facilitate early analysis of the temporal behavior. Detecting design flaws early in the development and especially timing errors is important to avoid costly re-design in late phases of the project. A component model with a precise syntax and semantic does not only support analysis at different stages, it also enables the development of tools that can generate code automatically.

The first objective in this paper is to present semantics for a component model that facilitates a

component based development of embedded real-time systems, based on the product line architecture concept. Such a component model constitutes the core entity in the architectural description language that facilitates the management and generation of component based real-time system. However, no syntax is presented. A possible syntactical implementation is the widely spread, and well-known UML language. The graphical syntax used in this paper is for the clarification of concepts only. The second objective in this paper is a comparison of the expressiveness of our semantics and the semantics of existing components models such as port-based objects and IEC-1131.

Before describing the specific contribution of this paper, two existing component models for embedded systems will be briefly described, port-base objects (Stewart, *et al*., 1997), and IEC 1131 (IEC, 1995). Neither port-based objects nor IEC 1131 do have explicit support for early timing analysis. Further, none of the models are developed with the objective to support the product line architecture approach. However, both models provide good support for structural reuse without considering the real-time behavior.

The Port-based object approach (PBO) was developed at the Advanced Manipulators Laboratory at Carnegie Mellon University. The model is based upon the development of domain-specific components that maximizes usability, flexibility and predictable temporal behavior. Independent tasks are the bases for the PBO model. Independent tasks are not allowed to communicate with other components, and thus components are loosely coupled and are consequently, at least in theory, easy to reuse. Although a system consisting of only independent components does not exist, minimization of synchronization and communication among components is a desired design goal. The data flow is specified through in- and out-ports. Whenever a PBO needs data for its computation, it reads the most recent information from its in-ports without knowing about the producer of that data. When a PBO component wants to make information available for other components in a system, its data is stored on the out-ports. In order to make PBO components more flexible and reusable a parameterization interface is provided. Through a parameterization interface several different application specific behaviors can be implemented by one single component. Besides the data interface and the parameterization interface discussed above, each PBO has an I/O interface. In figure 1, a PBO is depicted.

IEC 1131 is standard for programmable control systems and a set of associated tools e.g., debuggers, test tools, programming languages. The part of IEC 1131 related to our work is concerned with the programming language and is referred to as IEC 1131-3. IEC 1131-3 structures an application hierarchical and provides mechanisms for executing an application and for communication. The model is shown in figure 2.

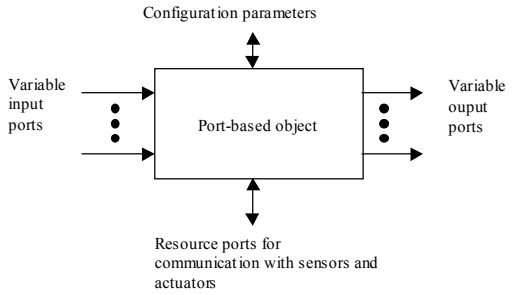

Fig. 1 Port-based objects

A *configuration* in IEC 1131 encapsulates all software for an application. In a distributed system several configurations allocated on different nodes may communicate with each other. A configuration consists of one or several *resources* that provide the computational mechanisms. A program is written in any of the languages proposed in the standard, i.e. *Instruction Lists*, *assembly languages*, *structured text* that is a high level language similar to Pascal, *ladder diagrams*, or *function block diagrams* (FBD). Ladder diagrams and FBD are graphical programming languages and where FBD is the most relevant for component-based development of embedded systems. Like in PBO, data-flow is specified in IEC 1131 function blocks by connecting in-ports and out-ports. Out-ports contains the result from a computation based on inputs and the current state of the function block. Furthermore, real-time tasks can be associated with a function block. Tasks can either be periodic or event-driven. Communication between function blocks within in the same program is straightforward whereas communication between function blocks in different programs is supported by special mechanisms.

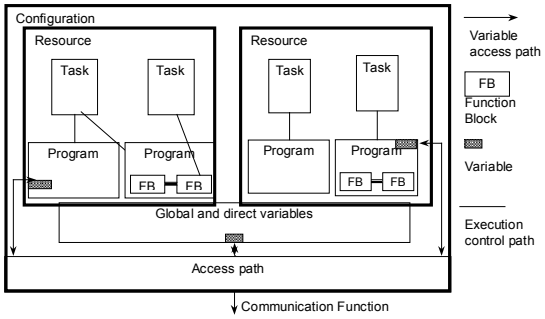

Fig. 2 IEC 1131-3 function blocks

Both PBO and FBD can be considered as special cases of the component model presented in this paper. By parameterization and task assignment our component model can express the very same

properties. Thus, it is more general and expressive. We also introduce the concept of abstract components as a way to specify variability. Furthermore, more complex temporal requirements can be specified in our model.

A component is considered as a description of an encapsulation of services, defined in terms of its interfaces and its services. Encapsulated services can be implemented in any ordinary programming language, whereas the component is implemented in a specific *component description language*. Components can be hierarchically composed. Consequently, a component may encapsulate other components, sub-components. We will refer to such a construction as an *aggregation*, which is the very same terminology used in DCOM. The encapsulation of an aggregated component is somewhat broken as its interfaces may become visible in the component's interface.

A component in our framework can resides in one out of three different states, *abstract component*, *concrete component* and *component instance*. The different states are depicted in figure 3. Concrete components and abstract components are both descriptions of encapsulations. However, an abstract component has an interface but no implementation of the behavior. The reason for having abstract components is to enable specification of components whose behavior must be tailored when reused across different applications. However, their interfaces are fixed.

A system is then generated according to the component and their interconnections. When generating a component instance, the component is dissolved into ordinary tasks and entry functions that can execute in any real-time operating system that supports the task model of the component. Thus, no special component infrastructure is needed in contrast to, e.g. CORBA. Tasks are defined in the control interface, whereas the entry functions correspond to the services.

A *task* is the architectural construction that defines the temporal constraints under which components executes. Concrete components together with tasks and parameterization defines the behavior, both functionalwise and temporalwise, for a concrete component.

The remainder of this paper is outlined as follows. In Section 2, the formal semantics of our component model is described. Section 3 discusses how systems are built based on instances of our components. An example of how this is done is also provided. Finally conclusions and future work is discussed in Section 5.

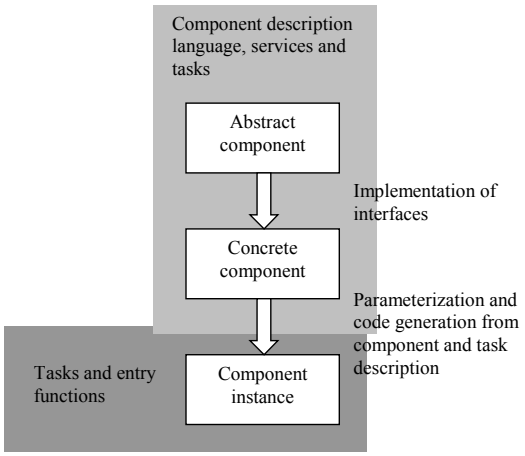

Fig. 3 Component states

2. THE COMPONENT MODEL

In this section we will describe our component model suitable for embedded real-time systems. The component model is based on the port-based object concept. Ports constitute the data interface for components as they define what data the component expects and the data it produces. However, port-based objects exhibit an overwrite semantics while our ports also can have buffered semantics. Besides having data interfaces, components in our framework have two additional interfaces, *control interface*, and *parameterization interface*. The execution of, and synchronization among services in a component is controlled through its control interface. The parameterization interface defines the points of variation of a component's behavior.

Moreover, components can be hierarchically composed, thus a component may encapsulate one or several other components. Furthermore, components can be either *concrete* or *abstract*. A concrete component, in its smallest constituent, encapsulates a service or other concrete components. An abstract component on the other hand, exists as a design entity only. The abstract component indicates that when the component is reused, the service it encapsulates must be rewritten, i.e. tailored for its new environment.

Definition 1. A component X is a tuple $<I_D(X),I_C(X),I_P(X),F(X),C(X), s_X>$, where $I_D(X)$ *is* the data interface, $I_C(X)$ is the control interface, $I_P(X)$ is the a parameterization interface, $F(X) = \{f_1,\ldots,_n f\}$ is the *set of services* encapsulated by component X, $C(X)=\{c_1,\ldots,c_m\}$ *is the* set of aggregated component encapsulated by component X, and s_X is the state. □

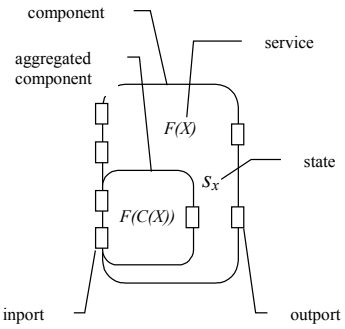

Fig. 4 The component's structure

A component's state can be internal variables whose values are kept intact between subsequent executions. The state may change value due to being manipulated by the services encapsulated by the component. Moreover, a components state is a composition of its own state, i.e. internal variables, and all states of the components it encapsulates, i.e. aggregated components.

Definition 2. A state is a persistent property that only can be changed by the services in a component. The state S_x of component X, is the recursive composition of all aggregated components states. $S_x = s_x \times S(C(X))$, where S_x is the composed state of component X, s_x is the state contribution from X and $S(C(X))$ is the set containing the states for all aggregated components.

The main difference between services and components is that components have a state.

2.1 Data interface

The data interface defines the input to, and the output from a component. We refer to input and output as ports and one data interface can consist of several such ports. Ports can exhibit two different semantics, overwriting semantic and buffering semantics. When overwriting semantics is specified, data consumers with a frequency lower than the producers might miss some data provided by the producer. On the contrary, if buffered semantics is specified data can be consumed in the pace of the consumer as long as the buffer is sufficiently large. Syntactically, the data interface specifies all ports in, and out from a component, each ports semantics, and the mapping from each port to the services in the component, or aggregated component, which require them.

Definition 3. A data interface for component X, is a set of in ports $I_{ID}(X)$ and a set of out ports, $I_{OD}(X)$. Each $f \in F(X) \cup F(C(X))$ is a function $in_n \times \ldots \times in_m \times s_X \rightarrow out_i \times \ldots \times out_j \times s_X'$, where $in_n, \ldots, in_m \in I_{ID}(X), out_i, \ldots, out \in I_{OD}(X)$, s_X is the state and s_X' is the updated state.

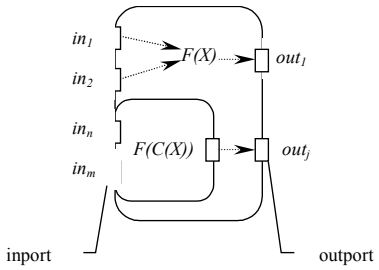

Fig. 5 A components data interface

Out ports may be associated with several services or aggregated components encapsulated by the component. However, only one encapsulated service or component can act as a data producer to an out port in the component instance. Thus, the set of services and components connected to an out-port are mutually excluding each other. Component instances is further elaborated on in Section 3.

2.2 Control interface

A components control interface specifies the restrictions under which its computational units, i.e. aggregated components and services, execute. The control interface defines the execution of computational units in terms of their temporal behavior and in term of relations to other computational units.

In order to control the execution of components, services and aggregated components must be assigned to a task. Task*s* define the temporal attributes that control the execution of components. Depending on the scheduling strategy, the actual attributes may vary. For instance, if the task is event-driven, no period time is specified. As components are "independent" from tasks, any scheduling strategy can be applied on a component. Thus, tasks that control the execution of components can be of any type, i.e. periodic, sporadic or aperiodic. A task can be associated with one or several aggregated components and services. Each service or aggregated component executes under restrictions imposed by its task. If no task is assigned to the individual service or aggregated component, they execute according to the task associated with the component instance defined on system level (See Section 3). If the execution of several elements in a component is controlled by one single task, the execution order among them submits to the specified precedence relations. Else if no executional relations are specified among the elements in a component, their execution order is non-deterministic.

Definition 4. *Task(X)* is a set of pairs $\langle \tau, x \rangle$ where τ is a *task* and $x \in F(X) \cup F(C(X))$.

However, executing constituents of a component in random order might not be sufficient. In order to specify the exact execution order, *precedence relations* and *mutex relations* among services and aggregated components is used. A precedence relation is a transitive, binary relation among services or aggregated components. If element *A* precedes element *B*, then *B* may start its execution earliest at the end of *A*'s execution.

Definition 5. *Precedence(X)* is a set, possibly empty, of pairs $\langle x_i, x_j \rangle$ where x_i precedes x_j and $x_i, x_j \in F(X) \cup F(C(X))$

Mutex is a binary, symmetric relation among component constituents such that if x_1 *mutex* x_2, then neither x_1 nor x_2 is permitted to execute while the corresponding party, or a transitively related party is executing.

Definition 6. *Mutual(X)* is a set, possibly empty, of pairs $\langle x_i, x_j \rangle$ where x_i mutually exclude x_j and $x_i, x_j \in F(X) \cup F(C(X))$

Now all parts of a components control interface is defined. Consequently, the control interface itself can be defined in Definition 7 as a tuple consisting of task assignments, precedence relations and mutex relations.

Definition 7. A control interface for component *X*, is a tuple $I_C(X)$, where $I_C(X) = \langle Task(X), Precedence(X), Mutual(X) \rangle$.

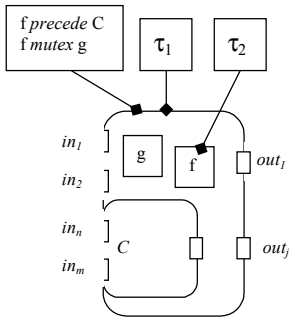

Fig. 6 The control interface

As definition 5 and definition 6 suggest, synchronization among execution entities is specified within the scope of components, i.e. among services and aggregated objects. Section 3 discusses component instances and systems that are composed of components. On the system level it is possible to specify synchronization among components. Consequently, synchronization may not take place between a component and an aggregated component or encapsulated service.

So far, specification of the temporal behavior of components with tasks has been discussed. However, tasks do not specify the execution time of components. The execution time for a component is dependent on the services it encapsulates. Thus the execution time is specified per service. As services not always are completely implemented, e.g. abstract components or components at design-time, the execution time usually specifies the budget that must be adhered to by the service.

Definition 8. *F(X)* is a set of services encapsulated by component *X* . Each element in *F(X)* is a pair $\langle f_i, t_i \rangle$ where f_i is a service and t_i is the service's execution time or its time budget.

2.3 Parameterization interface

The parameterization interface defines the points where the behavior of an implemented component can be varied between uses. Such a point is referred to as a *variation point*. In Section 2.2 was the control interface that provides variability through the task independence discussed, i.e. the constraints under which a component executes can be varied. However, through a components parameterization interface, *behavior* and *structure* of a component can be varied. The constituents in a component instance, i.e. services and aggregated components that are present in the actual component instance, define a component's structure. The behavior of a component is defined by the parameterization of each aggregated component and service that is part of the components structure. As in-ports, discussed in Section 2.1, determines the dynamic behavior of a service in terms of the calculated result, the behavioral parameterization statically specifies the behavior.

As an example, consider a navigation component for an autonomous vehicle. Depending on the type of sensor, e.g. infrared sensor, bump sensor, radio, the algorithm for calculating and presenting sensor values to the rest of the system will vary quit radically.

Definition 9. A parameterization interface for component *X* , $I_p(X)$ is a set of tuples $\langle c_i, P_i \rangle$, where c_i is a service or aggregated component defining the component structure and P_i is the set of parameters specifying its behavior.

3. SYSTEMS AND COMPONENT INSTANCES

As discussed in Section 1, components are distinguished from instances of components. An instance of a component is a function of a concrete component and its parameters. Consequently, component instances consist of fully implemented entities. Moreover, more than one service or

aggregated component can produce data on the same out-port in a component. In component instances such conflicts are resolved through the structural parameterization. Basically, every producer of data to the same out-port mutually excludes each other.

Definition 10. An instance of component *X*, *Instance*(*X),* is a concrete component with structural and behavioral parameterization and task assignments

A *system* consists of a set of component instances and their interconnections. A system describes the software architecture that implements functional- and quality requirements of an application. Component interconnections define control-flow and data-flow through an application. Data-flow is specified by connecting in-ports and out-ports in a consistent manner. By consistent is meant that all present in-ports are provided a data producer, i.e. an out-port. The control-flow is considered consistent if all computational entities, i.e. services and aggregated components, in a component are assigned a task. Moreover, if components in a system require synchronization among each other, i.e. precedence and mutually exclusion, this also is specified at system level. As a consequence, all elements of a component will obey the restrictions imposed by the specified synchronization. However, as discussed in Section 2.2, synchronization may also be specified among computational elements within a component. The component level synchronization will by obeyed by computational elements within a component when the component is allowed to execute according to the system level synchronization. Hence, synchronization is hierarchically specified.

4. A COMPARISON OF THE MODELS

In this section is the expressiveness of the proposed component model compared to port-based objects and IEC1131 in order to show that it is capable of specifying the same properties and in some cases, show that the semantics is more expressive. The comparison is made based on the constructions for which a semantic was specified in this paper, i.e. hierarchical composition, flexibility, temporal constraints, synchronization. It will be shown that the proposed model can express the same properties as both IEC 1131 and the port-based object model. However, our model is more expressive when it comes to specification of temporal attributes and synchronization. The notion of abstract components is also unique. When it comes to communication, both IEC 1131 and port-based objects have some explicit constructions specified. In our model communication among components can be implicitly specified through, for instance, shared memory protected by a semaphore, i.e. mutual exclusion.

4.1 Hierarchical composition

Hierarchical composition of component is essential for building reusable components of convenient size. A hierarchical approach, i.e. the possibility to specify aggregated component, support this by combining several smaller components with a unified and single interface to the rest of the system. As described in Section 2, our model comprises the concept of aggregated components. In the IEC 1131 standard, a function block can be composed by several other function blocks. Thus, IEC 1131 also can express hierarchical composition. Port-based objects on the other hand, have no such concept. Typically such components get to small to be practically useful in an industrial software reuse oriented organization.

4.2 Specification of variation points

Beside the possibility of having components of suitable size, requirements on their behavior and characteristics may vary between uses in different products in a product-line. This variation is accomplished through the parameterization interface and the concept of abstract components in our model. Through the parameterization interface, the behavior can be varied without violating the encapsulating of the component, whereas abstract components specify the need for a possible specialized implementations in a reuse situation. The port-based object model also has a parameterization interface. But there is no equivalent to our abstract components. Thus, in cases where only a common interface can be specified in the product-line architecture, port-based objects will fail to do so. In IEC 1131, there is no means for specifying flexibility explicitly, although one may solve this by using ordinary input-data to a function block as a constant that specify some variable property.

4.3 Specification of temporal constraints

The temporal constrains on a real-time system is of vital importance since correctness of such systems is defined to be both functional- and temporal correctness. Furthermore, as many parameters as possible is desirable when tuning the temporal behavior since this will minimize the semantic gap between the high-level temporal requirements and the task model provided by a real-time operating system, i.e. the infrastructure. As the infrastructure may vary between subsequent reuses, and thus the task model, components are required to be fairly independent from the actual temporal attributes, e.g. period time, deadline, offset. In our model, components are completely independent from the task models; there is only a relation between tasks and component or services in order to specify the

temporal constraints under which it must execute. IEC 1131 have a similar approach. However, the temporal attributes are restricted to a very small number and they are quite simple. Typically they specify a period time and priorities. The port-based object model is equally weak on the ability to express temporal constraints. But here are the temporal attributes, i.e. the period time in case of periodic execution, specified in the actual components. Thus, it is hard to use this model in an infrastructure that differs from the one intended for the component.

Generally, both IEC 1131 and port-based objects have quit tight coupling to a specific infrastructure, whereas our proposed model makes very few assumptions about the environment in which it will execute.

4.4 Specification of synchronization

Synchronization is an essential part of implementing the temporal requirements of a real-time system. In our model mutual exclusion between components and services can be specified. Moreover, precedence relations specify and control the order in which components are executed. In IEC 1131, there is a semaphore concept that can implement a mutual exclusion relation, but there is no equivalence to the precedence concept. In the port-based object model, the concept of synchronization among components is not defined. This is a major shortcoming of the models when they are used in large and complex systems.

5. CONCLUSIONS

In this paper is a component model suitable for use in the development of embedded real-time systems proposed. The model is particular suitable for systems where resources such as computational power and memory are very limited. A precise sematics for this model is defined. A precise semantics is necessary to enable architectural analysis of systems specified with such a model. A typical example of such analyses is scheduling where the temporal correctness is verified. The model is constructed with software product-lines in mind. Thus an essential part is how to specify flexibility in the components. The proposed model has been compared with two existing models, IEC 1131 and the port-based object model. The comparison shows that our model is as expressive as both of them are, but it extends the possibilities of specifying temporal properties as well as specifying synchronization. Furthermore, the model proposed in this is also independent from its infrastructure, i.e. operating system. This since the model makes no assumptions about its environment with regards to task models and component infrastructures such as name servers or object request brokers.

As future work we will implement our model and integrate it into a framework for designing software product-line architectures. The actual syntax has not yet been decided. However, we will investigate the possibility to use the industrial de facto standard UML (Stevens and Pooley, 1999). Being a language for specification of embedded real-time systems, we must look into the problem of specifying temporal- and resource constraints, e.g. memory consumption, CPU consumption. Preferably, we would like to assign budgets to components that all parts of a component that is part of its execution in a particular product instance must adhere to.

6. REFERENCES

Bosch J. (1999). Product-Line Architectures in Industry: A Case Study, In: Proceedings of the international conference on Software engineering, pp. 544 – 554,

Dikel, D., Kane D., Ornburn S., Loftus W., and Wilson J. (1996). Applying Software Product-Line Architecture. *IEEE Computer*, **30**, Nr 8. pp 49-55,

International Electrotechnical Commision, (1995). Application and Implementation of IEC 1131-3

Stevens P., and Pooley R., (1999). *Using UML – Software Engineering with Objects and Components*, Addison-Wesley

Stewart D. B., Volpe R. A., Khosla P. K., (1997). Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, *IEEE Transactions on Software Engineering*, **23**, Nr. 12