An Approach to Component-Based Software Engineering for Distributed Embedded Real-Time Systems

Uwe Rastofer, Frank Bellosa

Department of Computer Science IV, University of Erlangen-Nürnberg, Martensstraße 1, 91058 Erlangen, Germany

Uwe.Rastofer@informatik.uni-erlangen.de
Frank.Bellosa@informatik.uni-erlangen.de

ABSTRACT

The aim of Component-Based Software Engineering is to create applications from reusable, exchangeable and connectable components. But current component models lack support for important concepts of distributed embedded real-time systems. In this field a component's non-functional properties, e.g. execution time and resource usage, are as important as its functionality. In addition the non-functional properties are influenced by the platform on which the component is executed.

Therefore we propose a component model that separates the component's functionality from the platform-specific issues concurrency, synchronization, and distribution. We present a technique to describe the behavior of a component that is based on Use Case Maps (UCMs) and show how to deduce from these descriptions the behavior of an application that consists of connected components. In our model the UCMs also contain information on real-time requirements of the application.

Moreover we show how to adapt our components to an execution platform and how to create real-time applications with predictable properties from these components.

Keywords: Component-Based Software Engineering, Real-Time, Embedded Systems.

1. INTRODUCTION

Component-Based Software Engineering aims at decreasing development time and costs by creating applications from reusable, easily connectable and exchangeable building blocks - the components. The rules that these components have to follow are defined in a component model. Obeying the rules permits safe connection and substitution of components.

Current component models like Microsoft's COM [1] and Sun's Java Beans [2] are based on a uniform execution platform (i.e. the Win32 API and the Java Language and Virtual Machine) and some structuring conventions for components. These enable the composition of components by matching functional interfaces only. Non-functional issues, e.g. how long it takes to run a component and how much memory it consumes, are not addressed by these component models. It is assumed that a component can use nearly unlimited virtual memory, that there are no other timing constraints apart from the user's patience, and that one component's resource usage will not have any influence on other components.

Both assumptions, the negligible effects of non-functional properties and the uniform execution platform, do no longer hold in the field of distributed embedded real-time systems. In a real-time application functionally correct results have to be computed in time. A late result is as useless as a wrong result. Moreover it must be possible to predict beforehand if an application will always produce results in time.

Resources that can be used by an embedded application are usually scarce. Available processor time and memory are very limited due to hardware costs. Thus a component can easily influence others simply by consuming too many resources. To make matters worse, there is a variety of different hardware architectures and operating systems on the embedded market which creates a large number of diverse execution platforms.

Many embedded applications are also distributed across multiple nodes in a network. These nodes are connected via a field bus with real-time characteristics, i.e. bounded communication times. If a component that resides on a different node has to be contacted, the remote communication takes much longer than in the local case and therefore the timing behavior of the whole application is changed.

In the next section we introduce a concept for platform-independent components. We describe how these components are structured in section 3. In section 4 we show the creation of applications from components, whereas the final mapping of such applications to an execution platform is explained in section 5.

2. PLATFORM-INDEPENDENT COMPONENTS

To produce a correctly working embedded real-time application all non-functional properties of the execution environment have to be taken into account. Therefore the functional code of these applications is tangled with synchronization primitives, scheduling decisions, and communication calls. But the base functionality of these components is still independent of a certain execution environment.

To overcome platform-specific components we propose a component model that separates the component's functionality from platform-specific issues like concurrency, synchronization, and distribution. Applications are built from components in a platform-independent model and are later mapped to execution environments that introduce platform-specific features.

3. COMPONENT STRUCTURE

In the platform-independent model components have a number of input and output ports. A component can communicate with others by sending an uni-directional event on one of its output ports. An event is received by a component on one of its input ports. The events are typed and they may contain data. Since each port only supports events of a single type, the type hierarchy of events also creates a hierarchy of compatible input and output ports (this is described in more detail in section 4). Each port also has a name to uniquely identify ports that have the same type.

The reception of an event induces an activity in the receiving component. Otherwise components are passive and have no activity of their own. The induced activity executes an event handler that is associated with this input port. The event handler is not allowed to block during its execution. When the event handler returns the activity that was injected into the component also terminates, i.e. event handlers have run-to-completion semantics. During its execution the event handler can create new events. These events are transmitted on the component's output ports when the event handler returns. It is the responsibility of the output port to distribute an event to all connected input ports of other components. If the component has to send intermediate events the event handler has to be split into multiple small event handlers.

The component model must also fix to certain extend how event handlers are implemented. They can be thought of as methods or functions written in a portable programming language like ANSI-C or C++ which is available on the target platforms. The methods get the received event as a parameter and return a set of events to be transmitted. Event handlers can also be generated from executable specifications, e.g. ROOM State Machines [3].

Figure 1 shows the structure of a component in our model.

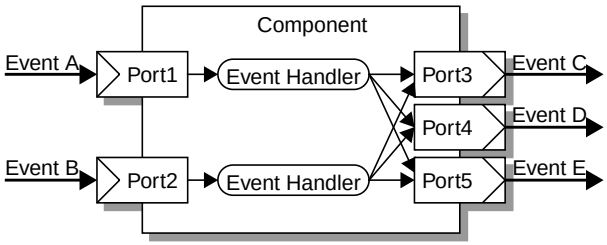

Fig. 1. Component Structure

Up to now components have been black boxes that receive and transmit events. This interface specification is not sufficient to enable safe composition of components and verification of non-functional properties. An approach that provides additional information on the behavior of components is needed [4].

In our model we use the Use Case Map (UCM) notation [5] for the specification of the component's behavior. The component developer associates a UCM with each component. Figure 2 shows the UCM for a simple buffering component. The component receives Data events on its

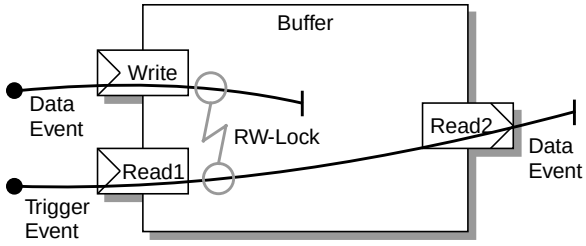

Fig. 2. Component Behavior for a Simple Buffer

Write port and stores the latest data item internally. The stored item can be read by sending a Trigger event through port Read1. This will create a Data event on port Read2. This is shown using two paths (~). Both paths start (●) outside the component. The Write path terminates (I) in the component whereas the Read path terminates outside. UCMs can also express AND/OR forks and joins of paths. For a more detailed discussion of UCM constructs see [5].

In principle, a component can receive any number of events at the same time and therefore many event handlers may execute concurrently. When event handlers access shared data these accesses have to be synchronized. The grey connection between the two paths in Figure 2 depicts such a case. As shown in [6] synchronization can be separated from the algorithmic code by attaching special objects that encapsulate the synchronization strategy. Each synchronization object can either implement one basic synchronization strategy

or can be built from a set of basic synchronization objects. In our model synchronization within the component is not possible because event handlers are not allowed to block. Instead synchronization takes place when a component receives an event at an input port. So each synchronized component has to specify a synchronization strategy and a mapping of its ports to that strategy. In Figure 2 the synchronization strategy is a Reader Writer Lock.

4. COMPONENT CONNECTION AND SYSTEM CREATION

The task of the system designer is to build an application from the aforementioned components. The system designer does this by creating instances of components and connecting output ports of component instances to input ports of other component instances. When an output port will be connected to an input port, the output port must transmit events of the same type or a sub-type of the events that are received by the input port. There is no restriction on how many input ports may be connected to an output port and vice versa. The resulting system is a directed graph with the nodes being components and the edges being connections. Figure 3 shows a simple control application, where a component reads data from a sensor, another component processes the data and a third component influences the system via an actuator.

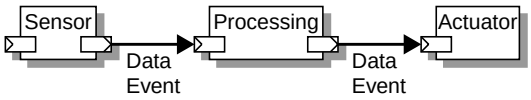

Fig. 3. A Simple Control Application

The connection of an output port to an input port also connects the respective paths that were specified in the UCMs of both components. Note that connecting two input ports to the same output port creates an AND fork of the path, while connecting two output ports to the same input port creates an OR join. Since paths always originate outside a component (components are passive) the new paths start outside a component as well. For the simple control application from Figure 3 all possible paths are shown in Figure 4. The system designer has to decide which paths that arise from component connection

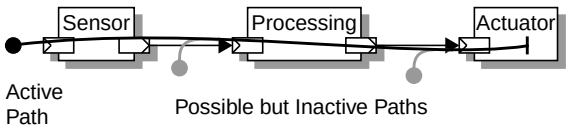

Fig. 4. Paths in the Simple Control Application

are active, i.e. on which paths activities are injected into the application. By selecting active paths the system designer implicitly creates a UCM for the behavior of the whole application.

When the system designer builds an application it is possible to create loops in the component graph. These structural loops may or may not cause behavioral loops. This depends on whether the structural loop also connects the paths in a cyclic way. On the one hand, behavioral loops can be useful, e.g. for computations where the results are incrementally refined. On the other hand, cyclic paths can be infinitely long which prevents the verification of the application's real-time properties. Therefore we allow cycles in active paths but require each loop to have a bounded number of iterations. This can be achieved by adding a specific component to the loop. This component's UCM contains an OR fork with a branch out of the loop. The termination branch is taken after a fixed number of iterations or after a time-out depending on the component's behavior specification.

After selecting active paths the real-time requirements of the application can be annotated on these paths. The real-time annotations that a system designer specifies are usually end-to-end characteristics of critical paths within the application. These critical paths are often called transactions [7].

In the simple control application a whole cycle of reading the sensor, processing the data and controlling the actuator has to be completed every 10 milliseconds. Therefore the period (T) is 10 milliseconds and the deadline (D) is equal to the period. The value for the starting time (T0) is zero, i.e. this activity should be started immediately when the application starts. The real-time annotations are also shown in Figure 5.

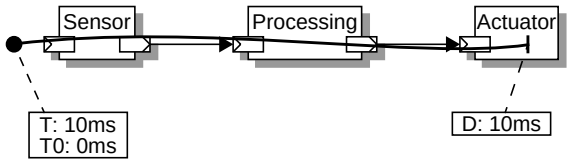

Fig. 5. UCM for the Simple Control Application

In order to improve the accuracy of the calculations it may be necessary to read the sensor and do the processing at a higher rate. If the actuator has to be controlled at the original rate the buffering component from Section 3 can be plugged in. As shown in Figure 6 there are now two active paths. The path that reads the sensor and does the processing has a shorter period (and deadline) than the actuator control path.

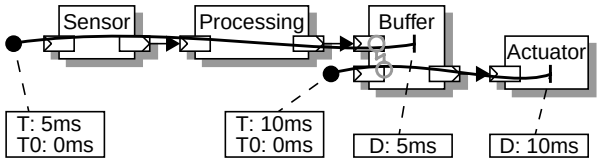

Fig. 6. UCM for another Control Application

Having only a single level of diagrams where the connections of all components are displayed rapidly leads to crowded pictures that are difficult to comprehend. A hierarchy of containers in which a graph of connected components acts as a single component on the next higher level can solve this problem. It is possible to automatically create a UCM for each container from the components inside it.

Although the hierarchy helps the system designer it is not required or used for the further analysis of the application. Here only the components on the lowest level and their direct connections are considered.

5. MAPPING TO AN EXECUTION ENVIRONMENT

When the system designer has created a complete application it still consists of platform-independent components. Before the application can be executed all components have to be mapped to the target execution environment and the specified real-time properties have to be verified. The whole mapping process consists of multiple steps. If one step fails, e.g. no feasible schedule can be found, the decisions of previous steps have to be reconsidered.

First of all the graph of connected components is partitioned into sets of related components. The partitioning process is guided by co-location and dis-location relations between components. Components that heavily communicate with each other should reside on the same node, whereas other components should be located on different nodes due to fault tolerance. Each set of components is then mapped to a single node of the distributed execution environment. More than one set may be mapped to the same node. Because some of the components, e.g. sensors and actuators, only work on the nodes that have the physical devices attached additional location constraints arise. Although co-location decisions can be deduced automatically from the UCM of the application, location constraints and dis-location relations have to be specified by the user.

When the location of each component is known, all paths that span multiple nodes are automatically cut up so than only local paths exist afterwards. The local paths are connected by special communication paths which model the sending of messages on the field bus. Figure 7 shows the local and the communication paths of the simple control application after the Sensor and Processing components were assigned to host A and the Actuator component was assigned to host B.

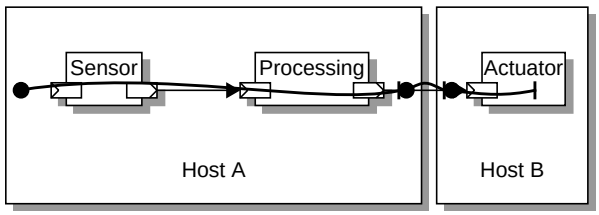

Fig. 7. UCM after Splitting Paths

After this step there is a set of local paths on each node. These paths are automatically mapped to tasks of the local execution platform. For a linear path there is a one-to-one relationship between a path and a task. The same applies to OR forks because only one branch of the OR fork will be taken at a time. Unlike OR forks, AND forks create new independent activities. Therefore an AND fork with *n* branches is the origin of *n-1* new tasks. The remaining branch will be taken by the task that entered the AND fork. An OR join unites multiple paths but the underlying tasks still execute independent of each other. Therefore tasks are only destroyed in AND joins where all but one of the entering tasks stop to execute. After mapping paths to tasks the application consists of a number of linear tasks that can be used for scheduling analysis.

The next step in the process of finding a schedule for a set of local tasks, is to get worst-case execution times (WCETs) for the event handlers that are executed by the tasks. This can either be done by measuring the component's performance on this platform and storing the values in a database for further use, or the WCET can be estimated using information about the code that was generated by the compiler. Compilers have to do control flow analysis anyway, so combining this information with the execution times of low-level instructions can lead to useful estimates although the influence of caches can distort execution times considerably.

With the information about tasks, WCETs of event handlers, and synchronization strategies of event handlers a schedule can be computed using well-known real-time scheduling theory like Rate-Monotonic Analysis (RMA) or Earliest Deadline First (EDF) [8] [9], depending on the scheduling strategy used by the node's operating system. When a local schedule for all nodes has been found the global schedule including the schedule for the communication between nodes is checked. If either a local schedule cannot be found because the load of the tasks on this node is too high or global constraints are violated, the whole process has to be started again with a different mapping of components to nodes.

When a valid schedule was found, all components have to be adapted to the platform and to the application. Special code has to be generated for each port. The code for input ports includes synchronization primitives that implement the specified synchronization strategy and calls to the communication system for events that are received from a remote component. For the output ports the generated code contains a list of all connected input ports, code to create new tasks for AND forks, and calls to the communication system for events that are sent to remote components. Additional code is necessary to initialize the application and set up the task schedule. The generated code together with the code of the event handlers represent the localized versions of the components. Together with the initialization code these components form an executable, distributed embedded real-time application.

6. CONCLUSION

In distributed embedded real-time systems the execution platform has a huge influence on non-functional properties of components. Therefore we propose a component model that separates the component's functionality from the platform-specific issues concurrency, synchronization, and distribution. The resulting components can be reused more easily because they are adaptable to the constraints of diverse execution environments. Moreover it is possible to create applications with predictable real-time properties from these components.

7. REFERENCES

[1] G. Eddon, H. Eddon: Inside Distributed COM. Microsoft Press, 1998.

[2] G. Hamilton: JavaBeans API Specification. Version 1.01, Sun Microsystems, Mountain View, 1997.

[3] B. Selic, G. Gullekson, P.T. Ward: Real-Time Object-Oriented Modeling. Wiley, New York, 1994.

[4] M. Büchi, W. Weck: A Plea for Grey-Box Components. Technical Report No 122, Turku Centre for Computer Science, Turku, Finnland, 1997.

[5] R. Buhr, R. Casselman: Use Case Maps for Object-Oriented Systems. Prentice Hall, UpperSaddle River, 1996.

[6] S. Reitzner: "Virtual Synchronization: Uncoupling Synchronization Annotations from Synchronization Code". In: Proceedings of the ACM SAC'98, Symposium on Applied Computing, New York, 1998.

[7] P. Cornwell: Reusable Component Engineering for Hard Real-Time Systems. Ph.D. dissertation, University of York, UK, 1998.

[8] A. Burns, A. Wellings: Real-Time Systems and Their Programming Languages. Addison-Wesley, Reading, 1990.

[9] J.A. Stankovic: Deadline Scheduling for Real-Time Systems – EDF and Related Algorithms. Kluwer, Boston, 1998.