An IDE for Component-Based Design of Embedded Real-Time Software

Jimmie Wiklander*, Johan Eriksson* and Per Lindgren*
*EISLAB, Luleå University of Technology, 971 87 Luleå
Email: {Jimmie.Wiklander/Johan.Eriksson/Per.Lindgren}@ltu.se

Abstract—This paper describes work in progress on a tool for component-based design of embedded real-time software. The tool supports graphical modeling of software systems using concurrent reactive objects and components, as well as generation of C code from the model. The resulting application code can then be combined with a lightweight kernel for execution on bare metal.

I. INTRODUCTION

Tool support is instrumental for efficient and correct design of embedded software. In this paper we present ongoing work on an Integrated Development Environment (IDE) for Component-Based Design (CBD) of embedded real-time software, supporting the design methodology presented in our earlier work [1]. The aim is to construct system models based on concurrent reactive objects and components in a graphical environment and to generate efficient C code from the model for execution on bare metal. More specifically, the system structure is defined graphically and the actual method code is written manually in C. The tool ensures that this code does not violate the properties of the model, such as state encapsulation in objects.

Distinctive to our approach is that timing requirements are captured as part of the model and are preserved throughout the design process; they can then be utilized by our kernel for scheduling at run-time. This stands in contrast to other modeling approaches (such as RT-UML [2] and MARTE [3]), which are typically based on successive transformations between different models, an approach that increases complexity of the design process. Due to space limitations, this work does not include any comparison to other modeling approaches.

The rest of this paper is organized as follows. In the next section (Section II) we describe the underlying modeling framework. In Section III we focus on the functionality and intended use of the tool. In Section IV we present the internal format of the tool. Finally, in Section V we discuss imminent future work.

II. MODELING FRAMEWORK

Our software design methodology relies on a unified, consistent modeling of both hardware and software. The modeling framework is based on the notions of reactive objects and time-constrained reactions. A more detailed presentation of the framework can be found in [1]. In this section we highlight the key aspects of time-constrained reactions, reactive objects, their abstraction to hierarchical concurrent reactive components, and discuss modeling interaction between components/objects as well as a system’s interaction with its environment.

A. Time-Constrained Reactions

Time-constrained reactions [4] lie at the heart of our model. Interaction between the system and its environment, as well as between components of the system is modeled as discrete events occurring at specific times. Following a reactive approach, functionality is specified in terms of time-constrained reactions to these events.

Timing requirements on system operation can be specified by defining the earliest and the latest reaction time (baseline and deadline) relative to the time of the event triggering the reaction. The time window between the reaction baseline and its deadline is called a permissible execution window for this reaction (Fig. 1).

B. Concurrent Reactive Objects

The fundamental modeling construct of the framework is a concurrent reactive object (CRO) [5]. Each object has a state and one or several methods, and it is reactive in the sense that it reacts to an incoming event by executing one of its methods. Thus a CRO is either idle (maintaining its state) or executes a method. Methods execute run-to-end, that is, once a method has started execution, no other method of the same object may preempt it. However, any two methods of different objects may execute concurrently. A method’s code can perform computations on local variables, read/mutate the object’s state, and invoke a method of the same or another object by sending a message to it.

Methods can be of two kinds (which corresponds to two kinds of messages sent between objects): asynchronous, which are executed concurrently with the caller and can be delayed by a certain amount of time, and synchronous, with the caller...
blocks until the invoked method completes execution, optionally returning a value (such methods cannot be delayed). Thus the permissible execution window of an asynchronous message can be either inherited or explicitly specified in the code relatively to the caller’s baseline; in either case, it is viewed in the model as a separate reaction. A synchronous message, on the other hand, always inherits the caller’s time constraints and is viewed as a part of the original time-constrained reaction. Both asynchronous and synchronous messages can carry data (the values of arguments of the invoked methods).

Each CRO has a provided interface (the methods, or input ports, of the object that can be invoked by other objects) and a required interface (the methods, or output ports, in the object’s environment that it may invoke). In the model, an output port in the required interface of an object can be linked to an input port in the provided interface of another object, creating a communication path between two different CRCs. Multiple output ports can be connected to a single input port.

C. Concurrent Reactive Components

Concurrent reactive objects create a flat, non-hierarchical structure of a system. To support efficient development of complex software system, we introduce the notion of concurrent reactive components (CRC), which contain no own state or methods but instead encapsulate a number of objects and other components, creating a simple hierarchical structure. Like objects, components have provided and required interfaces, but the ports in the interfaces are connected not to methods, but to ports in the interfaces of inner objects or components. Thus communication across component boundary is in fact communication between two objects belonging to two different components, which allows for an efficient implementation where a component hierarchy has no overhead at run-time.

It is worth noting that while a CRO can only execute one method at a time with concurrency existing between different objects, a CRC can be concurrent in itself as it may encapsulate multiple objects.

D. Classes and Instances

Every instance of a CRO/CRC belongs to a class, which can be defined at the top level in a module or locally within another class; in the latter case, it can only be used inside that class. A CRO definition defines the object’s state variables, methods (these are written manually in C and the tool verifies that the C code complies with the model’s requirements as outlined below), and the provided and required interfaces of the object. In the C code of a method we may:

1) define local variables,
2) read/update the object’s state,
3) perform calculations on the object’s state, method argument, and local variables,
4) invoke an asynchronous method of this object or of another object using the name of an output port (see Fig. 1):  

\[ \text{ASYNC(PortName, BaselineOffset, RelativeDeadline, Argument)} \]

5) invoke a synchronous method of another object using the name of an output port:

\[ \text{SYNC(PortName, Argument)} \]

6) return a value (only if the method is synchronous).

A CRC definition defines the provided and required interfaces and specifies instances of CROs and CRCs encapsulated in the component.

E. Kernel support and interaction with the environment

A system model (i.e. the system’s structure in terms of CROs and communication paths between them, and C code written for each method) can be used to generate C code for the application. Execution on a hardware platform also requires infrastructure supporting the model in the form of a lightweight kernel supporting scheduling of method execution and message passing between objects.

Embedded systems interact with their environment and this must be reflected in the model. This is modeled using a special construct, an environment interface component that defines an interface to the hardware. An instance of such interface component enables reading from hardware registers and writing to them and is also used to connect specific hardware interrupts to asynchronous methods of particular objects. The model does not make any assumptions regarding the behavior of the environment.

Sometimes it is necessary to include legacy software that is not based on the CRO model (typically, external software libraries). Such libraries can be treated as part of the system’s environment, and we can use an environment interface component as a wrapper to legacy code.

III. IDE FOR EMBEDDED REAL-TIME SOFTWARE

In this section we focus on the functionality and intended use of the tool, that is, constructing system models in a graphical environment. An example system (a process controller) is also presented.

A model of a particular system along with configuration settings is stored in a project file in XML format. Each project window initially contains two tabs: a configuration tab and a system tab.

A. System Tab

The system tab supports creating a graphical model of a system using modeling constructs described in the previous section. Definitions of objects and components can be created and edited in the tool, and instances of objects and components can be included in a component definition. The CRC/CRO definitions for the process controller example are shown in Figure 2. The content of the system and app CRC definitions from the example are shown in Figures 3 and 4, respectively. Connecting one interface port to another enables message passing (sending a synchronous or an asynchronous message) between instances at run-time. The content of a CRO definition (state and methods) can be edited in a built-in
text editor by opening a separate CRO tab for the object in question.

B. Configuration Tab

Certain settings have to be specified to generate code for a particular hardware platform. This is done in the configuration tab. In addition, it is necessary to specify the “root” CRC (the CRC that contains the whole system and that should be instantiated at system startup). The root component cannot have any ports in its interfaces and all interface ports of the inner objects/components must be connected for code generation to succeed.

C. Control system example

This example implements a simple control application, where the root component (called system) defines the environment interface envinst and the CRC instance appinst, encapsulating the CROs implemented in software. Thus the interfaces of the app component describe interaction with the environment, in this case there are two outputs (control_out, get_feedback) and three inputs, or entry points: the startup routine (start) and two interrupt handlers (inc, dec). The application component instantiates three objects (adapter, controller, and logger).

The adapter object definition provides three methods:

- a startup routine for setting up the local state, and initializing the controller and logger by calling their initialization methods, and
- inc and dec methods for increasing/decreasing the local state variable (these also invoke setpoint).

The controller object definition is specified in a similar way:

- init initializes the local state and invokes a private (not visible in the interface) method process,
- setpoint updates the internal state variables, and
- process, a private method that samples the AD, performs control calculations and sets a new output value periodically (the definition of a private method is that it is not part of the object’s interface, so it cannot be invoked from outside the object; private methods of an object can be edited by opening a CRO tab).

The logger is similar to the controller in that it has an initialization method and a method that periodically samples the AD for logging purposes.

IV. Internal Data Format

A model is stored in an XML file capturing the hierarchical structure of the system and method implementations. At the top level, it contains one or several modules:

```xml
<module name=mymodule>
[CONTENT]
</module>
```

In this paper we have restricted us to using only one module, but combining multiple modules in one project is useful for creation of commodity libraries for component re-use.

Each module contains one or several class definitions for CROs and CRCs. A class definition specifies a required
Fig. 4. The definition of the app CRC (provided/required interfaces of the app CRC in white and encapsulated CRO instances in yellow) with arrows indicating message paths.

interface (the list of output ports `arg`) and a provided interface (the list of input ports `res`). Assignments to the provided interface ports of the class are defined in `con`. Here is an example of a class definition (CONTENT stands for inner modeling constructs included in the class definition):

```xml
<class name="controller"
    arg="control_out, get_feedback"
    res="init, setpoint">
    [CONTENT]
    <result con="init=init, setpoint=setpoint"/>
</class>
```

A class definition of a CRO may contain:

1) One state structure with C code declaring state variables of the object, e.g.:

```xml
<state>
    <![CDATA[ int i; ]]>
</state>
```

2) One or several methods with C code implementation, written according to guidelines discussed in in Section II, e.g.:

```xml
<method
    name="init"
    arg="int arg"
    ret="void">
    <![CDATA[ASYNC(process, 100, NO_ARG);]]>
</method>
```

And a class definition of a CRC may contain:

1) Zero or more class definitions (as described above) only visible in the scope of the component and any sub-component(s) (referred to as local class definitions).

2) One or more instances of objects or components, created within an instance of this class, e.g.:

```xml
<inst
    name="contrinst"
    class="controller"
    classarg="get_feedback->get_val,
    control_out->control_out" />
```

Here `name` is the instance name, `class` selects the class definition to instantiate, and `classarg` are the argument(s) for instantiation.

V. FUTURE WORK

There are a number of obvious extensions that will be pursued in the immediate future. In particular, we will also add support for combining a number of ports into a single port, which is important to avoid clutter in graphical representation of larger embedded systems. We will also work on introducing a graphical representation of timing constraints for method execution. Finally, we will add support for component repositories (the tool currently supports creating multiple instances from a class definition but there is no support for re-use of definitions between projects).

ACKNOWLEDGMENT

This work was supported in part by the Knowledge Foundation in Sweden under a research grant for the SAVE-IT project, the EU Interreg IV A North Programme (grant no. 304-15591-08), the ESIS project (European Regional Development Fund, grant no. 41732), and the EU AESOP project.

REFERENCES