2009 35th Euromicro Conference on Software Engineering and Advanced Applications

Using JavaBeans to Realize a Domain-Specific Component Model

Juraj Feljan, Jan Carlson
Mälardalen Real-Time Research Centre
Mälardalen University, Sweden
{*juraj.feljan, jan.carlson*}*@mdh.se*

Mario Žagar
Faculty of Electrical Engineering and Computing
University of Zagreb, Croatia
mario.zagar@fer.hr

***Abstract*—SaveCCM is a domain-specific component model developed specifically for safety-critical hard real-time embedded systems in the vehicular domain. This paper expands the scope of SaveCCM to make it usable also outside this narrow domain, as the general concepts behind SaveCCM are applicable for a wider range of embedded systems. We describe the extensions made to SaveCCM in order to adjust it to a broader scope, focusing on a new realization mechanism. In its original form, SaveCCM systems are realized by components being grouped and transformed into real-time tasks. We propose an alternative realization of SaveCCM — by transformation to JavaBeans, which makes the executable system more general and portable, and maintains the structure of the component-based design.**

***Keywords*-component-based software engineering; embedded systems; SaveCCM; JavaBeans**

I. Introduction

SaveCCM [1] is a component model developed at Mälardalen University, intended to provide support for designing, analyzing and implementing safety-critical hard real-time embedded systems in the vehicular domain. In the design phase, SaveCCM systems are built by connecting components, according to the *component-based software engineering* (CBSE) approach. In the realization phase these systems are realized by transforming components to real-time tasks, to meet the requirements in the targeted domain.

In this paper we describe how SaveCCM can be extended for a wider domain, for instance embedded systems with soft real-time requirements. With this broader scope in mind, we investigate an alternative realization of SaveCCM, more fitting the intended new domain than the original realization.

Component model is a paramount concept in CBSE, as it provides a means for component specification and component interoperability. Currently among the most widely adopted component models are *general-purpose component models*, such as JavaBeans [2] and .NET [3]. In the embedded systems domain, CBSE is utilized to a lesser degree. Most embedded systems have features (e.g. limited memory and processing power) which are not considered by general-purpose component models, thus emphasizing the necessity to develop *domain-specific component models*, such as Koala [4], PECOS [5] or SaveCCM [1].

Our work is related to that of Åkerholm et al. [6] and later Dannmann [7], where a realization of SaveCCM is defined by grouping components and transforming them into tasks of a real-time operating system. Their run-time architecture emphasizes resource efficiency, at the cost of not retaining the component structure in the final system. Contrasting that approach, our realization is component-based, and is applicable for Java compliant platforms.

The paper is organized as follows: In Section II we present key aspects of SaveCCM, and Section III presents how the scope of SaveCCM was extended. The new realization is described in Section IV and exemplified in Section V. Section VI concludes the paper.

II. SaveCCM preliminaries

The main architectural elements of SaveCCM are *components*, *switches* and *assemblies*. The interface of an architectural element is defined by a set of *input-* and *output ports*, and systems are built from architectural elements by connecting ports.

SaveCCM is based on the control flow (pipe-and-filter) paradigm, but data transfer and control flow are separated. The former is captured by typed *data ports* that act as one-place buffers with overwrite semantics, and the latter by *trigger ports*. There are also *combined ports* that have both triggering- and data functionality.

An example of the SaveCCM notation is given in Figure 1. Trigger ports are denoted by triangles, and data ports by small rectangles. Circles and semicircles mark input- and output ports, respectively.

Components represent basic units of encapsulated behavior. For *basic components* the functionality is typically defined by an entry function written in C. There are also *composite components*, for which the functionality is defined by an internal composition of subcomponents (and possibly delays and switches, described below).

A component is initially *idle* and remains in that state until all its trigger input ports are activated. At that point it is *triggered* and changes to the *active state*. This initiates the *read phase*, in which the data input port values are stored internally to ensure consistent computation. Next, computations are performed in the *execute phase*, followed by the *write phase*, in which data are written to the data output ports. Finally, the output trigger ports are activated, and the component returns to the idle state. This strict "read-execute-write" semantics ensures that once a component is triggered, the execution is functionally independent of any concurrent activity.

There are two additional types of components — *clock* and *delay* — which are in charge of manipulating trigger

978-0-7695-3784-9/09 $26.00 © 2009 IEEE
DOI 10.1109/SEAA.2009.22

IEEE computer society

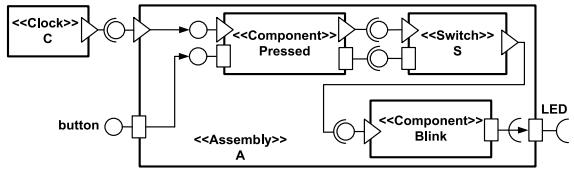

Figure 1. Example of a SaveCCM system

signals. A clock is a trigger generator, while a delay detains a trigger signal for a certain amount of time.

Switches enable dynamic modification of the connections between components by providing means for conditional transfer of data and/or triggering. A switch consists of a number of *conditional connections*, each representing a mapping between an input- and an output port of the switch (either data or trigger), guarded by a logical expression. If this expression evaluates to true, the connection holds, otherwise it is broken. Data input ports of a switch which are part of an expression are called *setports*.

Assemblies are encapsulated subsystems. As an assembly can break the "read-execute-write" semantics, it should only be viewed as a mechanism for naming a collection of components and hiding the internal structure, rather than a mechanism for component composition.

III. Broadening the scope of SaveCCM

SaveCCM is mainly intended for safety-critical hard real-time embedded systems, which has impact on a number of its characteristics. For instance, the communication between components follows the pipe-and-filter style, and a component can not freely access its ports at any time during its execution. However, although developed with this very specific domain in mind, SaveCCM has potential to be useful in a somewhat broader scope, e.g. in embedded systems with soft real-time requirements and more moderate resource constraints.

Separating domain specific aspects of SaveCCM from those that are domain independent, the domain specific aspects are found in: (i) the implementation of basic components, (ii) the realization, and (iii) particular analysis techniques.

The implementation of basic components: The behavior of basic components is currently implemented using C, which is the expected solution in the original SaveCCM domain. To cover more application types we propose Java instead of C. Although C is still the dominant language for embedded systems, employment of Java in this domain is increasing thanks to growing processor and memory resources of embedded devices, and the availability of editions of Java tailored for resource constrained devices (e.g. Java Micro Edition [8]). Examples of embedded devices running these special Java editions include TINI [9] and Sun SPOT [10].

The realization: SaveCCM makes no explicit assumptions in its specification about the realization. Having the safety-critical hard real-time embedded systems domain in mind, the envisioned approach is realization by allocating components to tasks [6], [7]. This enables high run-time efficiency and detailed timing analysis using standard real-time analysis techniques. For the wider domain we propose JavaBeans as the realization technology. The motivation for this comes from JavaBeans being a highly platform independent technology compatible with the proposed component behavior implementation in Java.

Analysis: The original SaveCCM approach relies heavily on different analysis techniques to determine or estimate properties of the system beforehand, in order to ensure predictability. Some of these techniques require detailed information about the underlying platform to be accurate, and would thus be categorized as platform specific. Investigating these methods further, however, is outside the scope of this paper.

IV. A JavaBeans realization of SaveCCM

As the basis of the new realization, we have developed an object-oriented representation of the SaveCCM elements and mechanisms in terms of JavaBeans, named *SaveJava*. Based on SaveJava, we are building a tool to automatically perform the transformation from a SaveCCM system definition to its JavaBeans realization.

SaveJava consists of two categories of classes: the *generic classes* and *specific classes*. The former make up the core of SaveJava, as they represent features common to all SaveCCM systems, and are unmodified across different realized systems. This category includes the *executor class* which is in charge of component scheduling and execution. Specific classes represent aspects of a particular SaveCCM system, such as individual components or data ports of a given type. This category encompasses a *system class* responsible for setting up the run-time architecture of a system realization. Its main method instantiates objects from the generic classes and specific classes, according to the structure of the SaveCCM system. Figure 2 shows a UML diagram of the generic classes (attributes and methods are omitted for the sake of readability).

The main SaveCCM constructs (basic- and composite components, clocks, delays, switches) are represented by Java beans. Thus, the SaveCCM component notion is maintained in our realization. Each port is realized by an individual object, and components hold references to their ports.

Since SaveCCM assemblies, switches and combined ports are semantically redundant, they are not given direct SaveJava counterparts in the form of beans or classes. Instead, they are first replaced by simpler SaveCCM constructs. Assembly borders are removed by directly connecting the internals of an assembly with elements on the outside. Switches are broken down into individual conditional connections, and a combined port becomes one data- and one trigger port.

Data ports are realized using Java Generics, allowing a single class hierarchy to represent data ports of different types. As an example, a data input port holding a value of string type would be represented by the StringDataInPort specific class which extends the DataInPort<String> generic class.

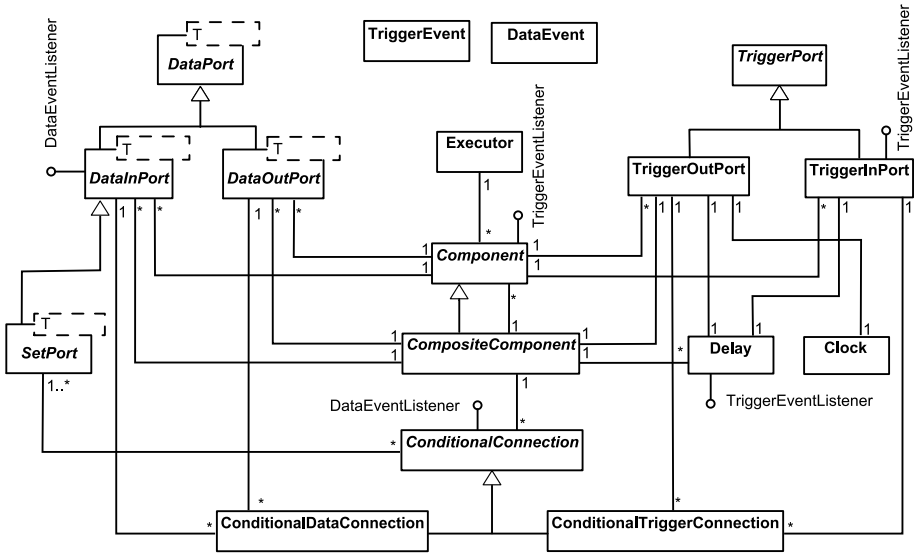

Figure 2. The SaveJava generic classes

Each component defined in a SaveCCM system is realized by a specific class which extends either the Component or CompositeComponent abstract generic class. Clock components and delay components are realized by the Clock and Delay beans, respectively.

SaveCCM connections are realized using the Java event model, which is the standard way to achieve communication between Java beans. Connecting one port to another is done by registering the destination port as a listener of the source port. An event type is realized by an event class and an event listener interface. In SaveJava there are two event types, one for data port connections and one for trigger port connections. Data connections use the DataEvent class and the corresponding DataEventListener interface. Trigger connections use the TriggerEvent class and the TriggerEventListener interface.

A conditional connection between two data ports or two trigger ports is realized by the ConditionalDataConnection and ConditionalTriggerConnection classes, respectively. Each conditional connection listens to all its setports, and when a change is detected the condition is retested. According to the condition state, the connection is left unchanged, set or broken.

The proposed component execution mechanism is a variant of the one used by Lednicki et al. [11]. Every realized SaveCCM system has one *executor*, an object instantiated from the executor class. This object holds a queue of triggered components and executes them one by one, in the same order as they got triggered.

A component is registered as a listener of all its trigger input ports. When one of them is activated, the component inspects the state of its other input triggers. If they are all active, the component is triggered, meaning that it saves the state of data input ports internally (i.e. performs the read phase) and adds itself to the executor's queue for execution. The component then waits for its turn to be executed. When it comes, the execute phase is performed, followed by the write phase. The component then returns to idle state and resets its triggers. Each of these phases (read, execute, write, reset triggers) is realized by calling a corresponding method.

The scenario is somewhat different for composite components. When a composite component becomes triggered it does not add itself to the executor's queue. Rather, it performs the read phase and then forwards data and triggering to its subcomponents according to the internal connections. To discover the end of execution, the composite component keeps track of the number of currently active subcomponents. When a subcomponent gets triggered or finishes executing, this number is increased or decreased accordingly. When it reaches zero, the composite component has finished executing, and the write phase is performed.

V. Realization example

In this section we present the realization of a simple SaveCCM system with a button as its input, and a LED for output. When the button is pressed, the LED blinks. Otherwise, the LED keeps its previous state.

A SaveCCM definition of the system is given in Figure 1. The system consists of a clock and an assembly encompassing two basic components and a switch. All data ports in the system are of boolean type. The data ports of the assembly represent the state of the button and the state of the LED. The Pressed component is used to make the button signal more robust — the button has to remain pressed/released for a number of consecutive clock cycles to be interpreted as pressed/released. According to the interpreted button state received at its setport, the switch establishes or breaks the trigger connection between Pressed and Blink. If this connection is established, Blink outputs true for a number of clock cycles, then false for as many cycles, which makes the LED blink. If the connection is broken, the previous value is preserved

```
public class SystemExample {
  public static void main(String[] args) {

    Executor executor = new Executor();

    // components
    Clock c = new Clock(100, 0,
        new TriggerOutPort("cTrig"));
    Component pressed = new Pressed(executor, null);
    Component blink = new Blink(executor, null);

    // conditional connections
    ConditionalConnection condConn =
        new ConditionalTriggerConnection(
        blink.getTriggerInPort("blinkTrigIn"),
        pressed.getTriggerOutPort("pressedTrigOut"),
        new BooleanSetPort("setPort", true));

    // connections
    c.getTriggerOutPort("cTrig").
        addTriggerEventListener(
        pressed.getTriggerInPort("pressedTrigIn"));
    pressed.getDataOutPort("pressedDataOut").
        addDataEventListener(
        condConn.getSetPort("setPort"));

    // input/output init and connection (omitted)

    // start the system
    executor.start();
    c.start();

  }
}
```

Figure 3. The system class

at the output port of the assembly, making the LED retain its current state.

The system realization consists of, in addition to the generic classes, six specific classes: a system class, two classes for the basic components (Pressed, Blink), two classes for the data ports (BooleanDataInPort, BooleanDataOutPort) and a class for the setport (BooleanSetPort). In the system class, shown in Figure 3, objects are instantiated and connected, following the structure of the input system. Port objects are created in the constructor method of the element they belong to. It is notable that the assembly borders are removed, by connecting the trigger output port of the clock directly with the trigger input port of pressed. After creating and connecting the system's objects, the external devices are initialized and connected to the system (this code is omitted from Figure 3). Finally, the threads of the executor and the clock are started.

VI. Conclusions

The original aim of SaveCCM is to bring CBSE benefits, such as reusability and alleviated maintenance, to the development of vehicular control systems with resource constraints and hard real-time demands. In this work, we have extended aspects of SaveCCM to make it suitable for a broader domain of embedded systems with soft real-time demands and less severe resource constraints. In particular, we have defined a new realization of SaveCCM, based on JavaBeans instead of real-time tasks.

Having in mind the addressed issues, a systematic evolution of SaveCCM has been achieved. Any platform independent analysis that can be performed on SaveCCM is valid also for our realization. As an additional benefit, with our realization the component structure from design-time is retained also in the final realized system.

Future work includes improving the executor mechanism to allow more flexible scheduling than the current non-interleaving first-in-first-out ordering, for instance by identifying beans that can be executed in parallel. We also want to investigate if a similar approach can provide a JavaBeans realization for ProCom [12], a successor of SaveCCM that is currently being developed.

Acknowledgment

This work was supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and the Unity Through Knowledge Fund via the DICES project.

References

[1] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, "The SAVE approach to component-based development of vehicular systems," *Journal of Systems and Software*, vol. 80, no. 5, May 2007.

[2] Sun Microsystems, "JavaBeans technology," http://java.sun.com/javase/technologies/desktop/javabeans/.

[3] Microsoft, ".NET Framework," http://www.microsoft.com/Net/.

[4] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, 2000.

[5] M. Winter, C. Zeidler, and C. Stich, "The PECOS software process," in *Workshop on Component-based Software Development Processes*, 2002.

[6] M. Åkerholm, A. Möller, H. Hansson, and M. Nolin, "Towards a dependable component technology for embedded system applications," in *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, January 2005.

[7] K. Dannmann, "Synthesizing real-time components to run-time tasks," Master's thesis, University of Oldenburg, 2009.

[8] Sun Microsystems, "Java Micro Edition," http://java.sun.com/javame/index.jsp.

[9] Maxim, "TINI," http://www.maxim-ic.com/products/microcontrollers/tini/.

[10] Sun Microsystems, "Sun SPOT," http://www.sunspotworld.com/.

[11] L. Lednicki, J. Carlson, and M. Žagar, "Uniform treatment of hardware- and software components," in *Eight Conference on Software Engineering Research and Practice in Sweden*, November 2008.

[12] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković, "A component model for control-intensive distributed embedded systems," in *11th International Symposium on Component Based Software Engineering*. Springer Berlin, October 2008.