Using JavaBeans to Realize a Domain Specific Component Model

Juraj Feljan, Jan Carlson
Mälardalen Research and Technology 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 a transformation to JavaBeans, which maintains the structure of the component-based design, and also makes the executable system more general and portable.

1 Introduction

SaveCCM [1] (SaveComp Component Model) is a domain specific component model targeting vehicular systems (i.e. safety-critical hard real-time embedded systems), developed at Mälardalen University. In the design phase, SaveCCM systems are built by connecting components, according to the CBSE approach. In the realization phase these systems are realized by transforming components to real-time tasks, to meet the requirements on efficiency and reliability in the targeted domain.

In this paper we describe how SaveCCM can be extended for a wider domain, for instance embedded systems with soft- or no real-time requirements. With this broader scope in mind, we investigate an alternative realization of SaveCCM, more fitting the intended new domain than the realization by allocating components to tasks.

Component-based software engineering (CBSE) is a discipline that promotes development of software systems from preexisting *software components*. A component is a reusable part of software that has a clearly specified interface and can be combined with other components to build larger units (a combination of definitions by D'Souza and Willis [6] and Szyperski [23]). The usage of components facilitates comprehension of complex systems and simplifies maintenance by allowing individual components to be updated with newer versions without modifying the rest of the system. Components can be developed separately from the system they are used in, which shortens time-to-market and enables reusability of the same component across different systems.

In order for components of a particular system to communicate, they must conform to a *component model*. A component model is a means for providing component interoperability, i.e. it defines standards which component developers and users must follow. Currently, among the most widely adopted component models are JavaBeans [21], .NET [12], Enterprise JavaBeans [17] and CORBA Component Model [13]. These are *general purpose component models*, used mainly in application- and enterprise domains, where CBSE has proven quite successful. On the other hand, we have the embedded systems domain, where CBSE is utilized to a lesser degree. General purpose component models usually focus on enabling design phase simplicity, relying on powerful hardware to handle the model overhead. However, most embedded systems have very limited memory and processing power at their disposal, and they are often subject to real-time constraints or even have a safety-critical role. These features are not considered in general purpose component models, thus emphasizing the necessity to develop *domain specific component models*, such as Koala [25], PECOS [11], Rubus [4] or SaveCCM [1].

The remainder of this paper is organized as follows. In Section 2 we describe the background of our work by presenting key aspects of SaveCCM. Section 3 presents how the scope of SaveCCM is extended. In Section 4 we discuss the central part of our work — a new realization of SaveCCM. Section 5 shows a particular example of the new realization, Section 6 presents related work and Section 7 concludes the paper.

2 SaveCCM preliminaries

In this section we give an overview of SaveCCM, focusing on aspects that are most important to understand our work. A complete specification of SaveCCM can be found in the SaveCCM Language Reference Manual [2].

SaveCCM is a domain specific component model intended to provide support for designing, analyzing and implementing embedded control applications for vehicular systems, mainly considering the safety-critical subsystems responsible for controlling vehicle dynamics (such as power-train, steering, braking, etc.).

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*. SaveCCM systems are built from architectural elements by connecting ports.

SaveCCM is based on the control flow (pipe-and-filter) paradigm. Data transfer and control flow are separated, and therefore SaveCCM distinguishes between *trigger ports* that capture control flow, and *data ports* that capture data transfer. Data ports are typed and have overwrite semantics, and only data ports of matching types can be connected. There are also *combined ports* that have both triggering- and data functionality, but semantically a combined port is equivalent to one trigger port and one data port.

For an example of the graphical SaveCCM notation, see Figure 5 in Section 5. 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. The functionality of a component is typically defined by an entry function, which is written in C. These are *basic components*. However, there are also *composite components*, for which the functionality is defined by an internal composition of subcomponents (and possibly delays and switches, described below). Subcomponents can be basic components or again composite components (thus creating a hierarchy of arbitrary depth).

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

Composite components have additional ports on the border between the internals and externals: one internal trigger output port, and for each external data port there is a corresponding internal data port of the same type. These internal ports are used to forward the input of the composite component to its subcomponents, and to let the subcomponents produce the output of the composite. The internal trigger port initiates internal computation by triggering subcomponents connected to it. A composite component has finished executing when no subcomponent is triggered or active.

There are two additional types of components — *clock* and *delay* — which are in charge of manipulating trigger 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 such a logical expression are called *setports*. Switches are not triggered, they respond immediately to arrival of data- or trigger signals at their input ports, and relay them according to the current mappings.

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.

The SaveCCM semantics is formally defined by a two step transformation — first from the full SaveCCM language to a similar but simpler language called SaveCCM Core, and then into timed automata with tasks. These transformations are detailed in the reference maunal [2].

SaveCCM systems are developed using a custom development environment called SaveIDE [15], which is implemented in the form of a plugin for the Eclipse platform [7].

3 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. Moreover, the task-based realization permits a very lightweight runtime framework, addressing the resource constraints faced in many vehicular systems.

However, although developed with this very specific domain in mind, many aspects of SaveCCM have a 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 mainly found in:

- implementation of basic components,
- component realization, and
- particular analysis techniques.

After identifying these areas, we can set about modifying them in order to expand the domain in which SaveCCM can be used.

3.1 Implementation of basic components

The behavior of basic components is currently implemented using C, which is the standard and expected solution in the original SaveCCM domain. To cover more application types we propose Java to be used as the implementation language. Java is platform independent and ubiquitous, as it is widely accepted and used in a wide range from embedded systems to desktop computers.

Although C is still the dominant language for embedded systems, employment of Java in this domain is increasing. First, thanks to growing processor and memory resources of embedded devices. Second, thanks to the availability of special editions of Java (albeit limited compared to Java Standard Edition), tailored for resource constrained devices. These are, for instance, Java Micro Edition [19], targeted for mobile devices, PDAs, TV set-top boxes and printers, or Java Standard Edition for Embedded [20] — an optimized, "headless" (no support for mouse- and keyboard input, nor for display graphics) version of the standard desktop Java. Examples of embedded devices running limited Java are TINI [10] and Sun SPOT [22].

3.2 System 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 [3, 5]. This enables high run-time efficiency and detailed timing analysis using standard real-time analysis techniques. As this approach targets real-time operating systems, to be in line with the desired wider domain we propose JavaBeans as the realization technology. The motivation for using JavaBeans comes from two directions. First, it is a platform independent technology and follows the "write-once, run-everywhere" philosophy, thus blending in well with extending the scope of SaveCCM. Second, it is compatible with the proposed component behavior implementation in Java.

The JavaBeans technology is a portable, platform-independent software component model for the Java SE platform. The technology consists of a Java package (`java.beans`) and the JavaBeans specification [21] which describes how classes and interfaces from the package should be used to implement the *Java bean* concept. A Java bean is a Java class that complies to conditions stated in the specification.

3.3 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, while others can be performed on a higher, platform independent, level of abstraction. Investigating these methods further, however, is not within the scope of this paper.

4 Realization of SaveCCM by transformation to JavaBeans

In this section we present our realization of SaveCCM by transformation to JavaBeans. In order to achieve the transformation, we defined a mapping from SaveCCM to JavaBeans, or in other words, an object-oriented representation of the SaveCCM elements and mechanisms in terms of JavaBeans. We named this representation SaveJava.

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 (see Section 4.1). The latter 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 1 shows a UML diagram of the generic classes (attributes and methods are omitted for the sake of readability).

The generic classes provide a framework for realizing SaveCCM systems with JavaBeans. To get the full realization of a particular system, specific classes need to be created for that system. The envisioned approach is to do this automatically from a SaveCCM system definition (see Section 4.2).

Next we describe how particular SaveCCM constructs are mapped to Java. The main SaveCCM constructs (basic-

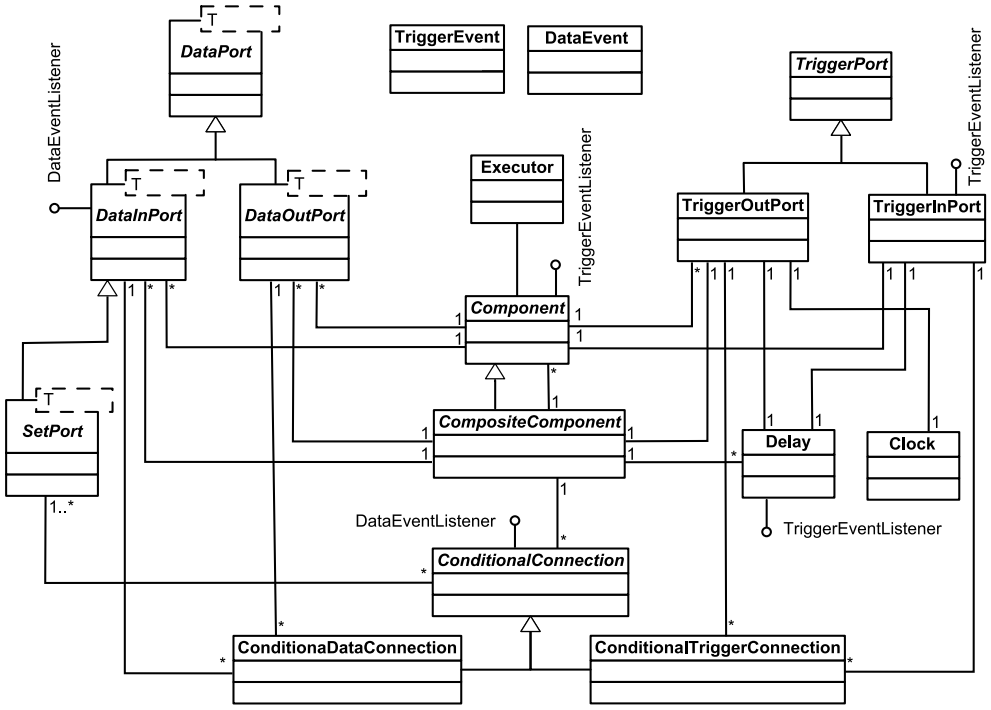

Figure 1. The SaveJava generic classes

and composite components, clocks, delays and switches) are represented by Java beans. Thus, the component notion from SaveCCM is maintained in our JavaBeans 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 constructs, following the translation from full SaveCCM to SaveCCM Core. Assembly borders are removed by directly connecting the internals of an assembly with entities outside of the assembly. Switches are broken down into individual conditional connections, and a SaveCCM combined port becomes one data port and one trigger port in SaveJava.

Two separate hierarchies are used to represent ports — one for data ports and one for trigger ports. The generic port classes are shown in Figure 2 (attributes and method parameter/return types are omitted for the sake of readability). Data ports are realized using Java Generics, allowing a single class 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.

SaveCCM components are represented with generic classes depicted in Figure 3. These classes implement

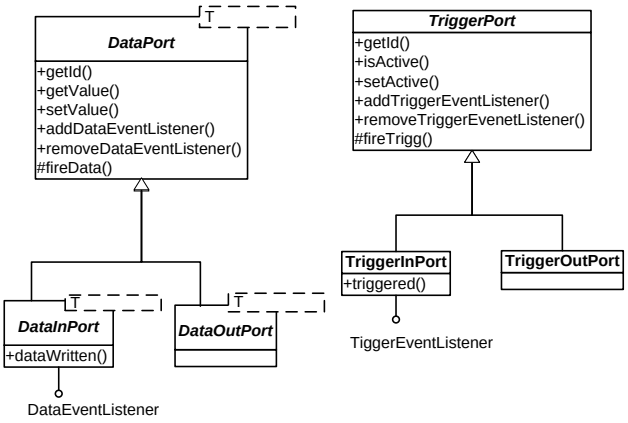

Figure 2. The generic classes for ports

mechanisms common to all basic- and composite SaveCCM components. Each component defined in a SaveCCM system is realized by a specific class which extends either the `Component` or `CompositeComponent` abstract generic class. The three abstract methods from `Component` — `read`, `execute` and `write` have to be implemented in the specific class. For composite components the `execute` method is left blank, as the behavior is defined by the composite component's internal composition, not by code. In addition to external ports inherited

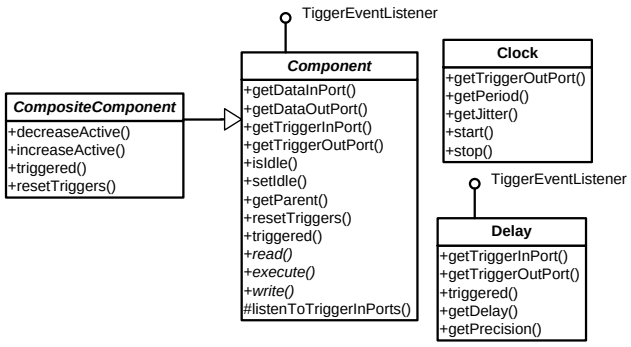

Figure 3. The generic classes for components

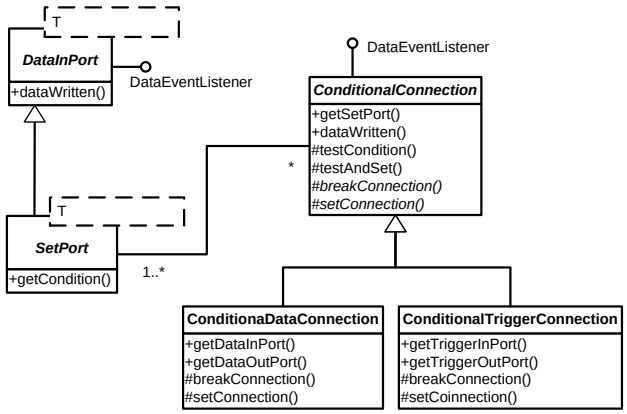

Figure 4. The generic classes for conditional connections

from `Component`, `CompositeComponent` has internal data ports and one internal trigger output port. It also holds references to the internal entities (components, delays and conditional connections), as shown in Figure 1.

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.

Conditional connections are realized by the generic classes shown in Figure 4. 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.

For a condition to be evaluated as true, the latest value received at the setport should match the stored value. If a conditional connection has more than one setport guarding it, all of them should be evaluated as true in order for the connection to be established. Different condition rules can be achieved by extending the `ConditionalDataConnection` or the `ConditionalTriggerConnection` class, and overriding the `testCondition` method.

4.1 The component execution mechanism

The proposed component execution mechanism is a variant of the one used by Lednicki [8]. 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 always registers as a listener of all its trigger input ports. When one input trigger of a component is activated, the component inspects the state of its other input triggers. If they are all active, the component is triggered, meaning that it adds itself to the executor's queue for execution and saves the state of data input ports internally, i.e. it performs the read phase. Since the executor's queue is a FIFO structure, the component waits for its turn to be executed. When this time 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, write, execute, reset triggers) is realized by calling a corresponding method.

This 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 by calling the `read` method and then activates its internal trigger output port, which then relays triggering signals to subcomponents. The `execute` method is never called, as the behavior is defined by the executing subcomponents. After a composite component finishes executing, it calls its `write` and `resetTriggers` methods.

For a basic component it is trivial to observe when it finishes executing — upon returning from the `execute` method. To discover the end of execution for a composite component we use a counting mechanism. Each composite

component holds 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` method is called.

All components are executed in the same thread, managed by the executor. Alternatively, each component could have been given its own thread, but this would introduce the need for elaborate thread synchronization to ensure that the specifics of the SaveCCM semantics are satisfied. Clocks, on the other hand, have their own threads, as this allows them to correctly generate triggering at the specified rate.

4.2 The transformation tool

Based on SaveJava, we are developing a tool that automatically performs the transformation from a SaveCCM system definition to its JavaBeans realization. The tool takes as input a description of a particular SaveCCM system (represented by the `.save` file, in XML format, produced by SaveIDE), and generates realization code (a set of generic and specific classes) as output. The generic classes exist prior to the transformation and are merely copied to the target directory, while specific classes are generated during the transformation, according to the SaveCCM system specified in the input file.

The tool is implemented in Java. For parsing the input file we use Java Architecture for XML Binding [18], a technology which maps between XML elements and Java objects, thus providing an easy and intuitive way for XML parsing.

It is possible to define a partial system in SaveCCM, transform it to JavaBeans and then continue developing the system in terms of JavaBeans. Also, using the generic classes as a framework, it is possible to code a system realization by manually writing specific classes. However, the intended approach is to model a full SaveCCM system in SaveIDE, and then automatically generate the realization using our tool.

At the current stage the tool supports only a subset of SaveCCM — automated transformation can be done for SaveCCM systems that contain basic components, clocks and delays. Switches, assemblies and composite components are not yet supported.

5 Realization example

In this section we present an example of a SaveCCM system realization. The system to be realized has 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 5. 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 a toggling value (`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 at the output port of the assembly, making the LED retain its current state.

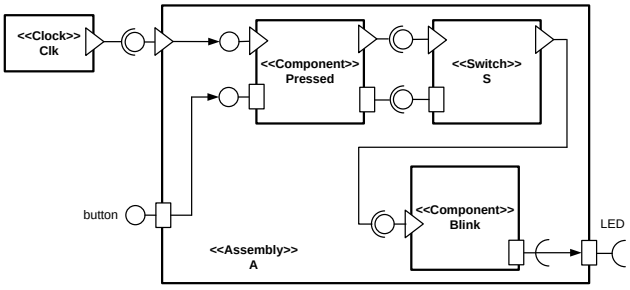

Figure 5. Example of a SaveCCM system

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 6, 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`. This way, delegations are replaced by connections. After creating and connecting the system's objects, the external devices are initialized and connected to the system. However, this code is not part of the realization and is omitted from Figure 6. Finally, the threads of the executor and the clock are started.

6 Related work

Our work is most closely related to the work done by Åkerholm et al. [3] and later by Dannmann [5]. They define the aforementioned realization of SaveCCM by grouping components and transforming them into tasks of a real-time operating system. As the first step of this transformation,

```
public class Test {

  public static void main(String[] args) {

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

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

    // connections
    clk.getTriggerOutPort("clkTrig").
        addTriggerEventListener(
        pressed.getTriggerInPort("pressedTrigIn"));
    pressed.getDataOutPort("pressedDataOut").
        addDataEventListener(
        connection.getSetPort("setPort"));

    // input/output initialization
    ...
    // input/output connections
    ...

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

  }

}
```

Figure 6. The system class

the control flow of the SaveCCM system is analyzed. Rules for how components may be allocated to tasks are applied, resulting in the creation of task trees. In the final phase, the task trees are processed to generate the final code of each task. This includes generation of glue code specifying the order in which the components in a task are executed, and for taking care of inter- and intra task communication. The run-time architecture developed by Åkerholm et al. and Dannmann emphasizes resource efficiency, at the cost of not retaining the component structure in the final system. Contrasting their approach, our realization is component-based, and is applicable for any Java compliant platform in soft real-time domain. Combining their and our contributions results in SaveCCM now having two complementary realizations.

Petričić [14] also addresses a transformation of SaveCCM, by defining a transformation between SaveCCM and UML. According to the taxonomy proposed by Visser [26], her transformation can be classified as a migration, since SaveCCM and UML are on the same level of abstraction. The transformation we defined is a synthesis, as it lowers the level of abstraction.

Marvie [9] experiments with transformations from an abstract model to a technological one, from the perspective of model-driven development. He defines an experimental model of a message filtering system and defines transformations to several technologies, among them JavaBeans. Similarly to our work, he realizes an abstract model of a system using the JavaBeans technology. However, he defines the realization of an example toy model, while we do the same for a component model.

7 Conclusions and future work

The original aim of SaveCCM is to bring CBSE benefits, such as reusability and alleviated maintenance, to the development of vehicular systems with resource constraints and hard real-time demands. In this work, we have extended aspects of SaveCCM, making it more platform- and domain independent. In particular, as the main part of our work, we have defined a new realization of SaveCCM. By adding to the realization by transformation to tasks our realization by transformation to JavaBeans, we have expanded the scope of SaveCCM to embedded systems with soft real-time demands and less severe resource constraints. Having in mind the addressed issues, a systematic evolution of SaveCCM has been achieved. Any platform independent analysis that can be performed on SaveCCM, such as the analysis performed with the UPPAAL Port model checker [24] integrated with SaveIDE, 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.

The current version of the realization tool does not support all SaveCCM elements, as composite components, assemblies and switches are missing. Including them in the tool is part of future work. Also, as the tool is currently standalone, we plan to implement it in the form of an Eclipse plugin and integrate it more tightly with SaveIDE.

The executor mechanism runs components sequentially, in a non-interleaving fashion. We plan to investigate different approaches to component execution and find ways to improve scheduling, for instance by identifying beans to be executed in parallel. Closely tied to scheduling is the issue of analysis of the new realization, with respect to timing, resource consumption, etc.

A new component model, called ProCom [16], is currently being developed at Mälardalen University. This component model is similar to SaveCCM in several aspects, and we would like to investigate if the SaveJava approach can be extended and modified to provide a JavaBeans realization for ProCom.

Acknowledgement

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*, 80(5):655–667, May 2007.

[2] M. Åkerholm, J. Carlson, J. Håkansson, H. Hansson, M. Nolin, T. Nolte, and P. Pettersson. The SaveCCM Language Reference Manual. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-207/2007-1-SE, Mälardalen University, January 2007.

[3] 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 ependable Systems (WORDS 2005)*. IEEE, January 2005.

[4] Arcticus Systems. Rubus Component Model. `http://www.arcticus-systems.com/`.

[5] K. Dannmann. Synthesizing Real-Time Components to Run-Time Tasks. Master's thesis, Carl von Ossietzky University of Oldenburg, Germany, February 2009.

[6] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Boston, MA, USA, 1999.

[7] Eclipse. `http://www.eclipse.org/`.

[8] L. Lednicki. Component-based development for software and hardware components. Master's thesis, Mälardalen University, Sweden, June 2008.

[9] R. Marvie. MDA, Model Transformations and Platforms: Advocating Technological Jumps. Technical report, INRIA, September 2004.

[10] Maxim. TINI. `http://www.maxim-ic.com/products/microcontrollers/tini/`.

[11] C. S. Micael Winter, Christian Zeidler. The PECOS Software Process. In *Workshop on Component-based Software Development Processes*, 2002.

[12] Microsoft. .NET Framework. `http://www.microsoft.com/Net/`.

[13] OMG. CORBA Component Model. `http://www.omg.org/technology/documents/formal/components.htm`.

[14] A. Petričić. UML profile for SaveComp Component Model. Master's thesis, Mälardalen University, Sweden, June 2008.

[15] S. Sentilles, J. Håkansson, P. Pettersson, and I. Crnković. Save-IDE An Integrated development environment for building predictable component-based embedded systems. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008.

[16] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A component model for control-intensive distributed embedded systems. In M. R. Chaudron and C. Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer Berlin, October 2008.

[17] Sun Microsystems. Enterprise JavaBeans technology. `http://java.sun.com/products/ejb/`.

[18] Sun Microsystems. Java Architecture for XML Binding. `http://java.sun.com/developer/technicalArticles/WebServices/jaxb/`.

[19] Sun Microsystems. Java Micro Edition. `http://java.sun.com/javame/index.jsp`.

[20] Sun Microsystems. Java SE for Embedded. `http://java.sun.com/javase/embedded/`.

[21] Sun Microsystems. JavaBeans technology. `http://java.sun.com/javase/technologies/desktop/javabeans/`.

[22] Sun Microsystems. Sun SPOT. `http://www.sunspotworld.com/`.

[23] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, New York, NY, USA, 2002.

[24] Uppsala University, Aalborg University. UPPAAL Port. `http://www.uppaal.org/port/`.

[25] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[26] E. Visser. A survey of strategies in program transformation systems. In *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.