Realizing a domain specific component model with JavaBeans

Juraj Feljan, Jan Carlson
Mälardalen Research and Technology Centre
Mälardalen University
PO Box 883, SE-721 23 Västerås, Sweden
{juraj.feljan, jan.carlson}@mdh.se

Mario Žagar
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3, HR-10000 Zagreb, Croatia
mario.zagar@fer.hr

ABSTRACT
SaveCCM is a domain specific component model developed specifically for safety-critical hard real-time embedded systems. The goal of this paper is to extend the scope of SaveCCM to make it usable also outside this narrow domain, as the general concepts behind SaveCCM are applicable as well for embedded systems that have soft or no real-time constraints. We describe the modifications made to SaveCCM in order to adjust it to the wider scope, focusing on defining a new realization mechanism. In its original form, a SaveCCM system is realized by component allocation to real-time tasks, which means that individual components are not observable in the run-time system. We propose realizing SaveCCM by a transformation to JavaBeans, making the advantages of component-based development present also at run-time. This way we also make the executable system more general and portable.

Categories and Subject Descriptors
D.2 Software Engineering: D.2.2 Design Tools and Techniques

General Terms
Design, Languages

Keywords
SaveCCM, JavaBeans, CBSE, component model, transformation between component models

1. INTRODUCTION
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[1]. 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 the most widely adopted component models are JavaBeans [18], .NET [10], Enterprise JavaBeans [16] and CORBA Component Model [11]. 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 [7]. 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 [12], PECOS [21], Rubus [4] or SaveCCM [1].

SaveCCM (SaveComp component model) is a domain specific component model targeting 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. However, in the realization phase these components are realized by transformation 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, and even desktop applications. With this broader scope in mind, we investigate an alternative realization of SaveCCM to preserve the design-time component structure in the run-time system, thus taking advantage of the CBSE benefits also at run-time.

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 and JavaBeans. Then, we present how we extended the scope of SaveCCM in Section 3. In Section 4 we discuss the new realization of SaveCCM. Section 5 shows a

[1] A combination of definitions by D'Souza and Willis [5] and Szyperski [19].

SERPS'08, November 4-5, 2008, Karlskrona, Sweden.

particular example of the realization. Section 6 presents related work and Section 7 concludes the paper.

2. BACKGROUND

In this section, we give brief overviews of the two component models SaveCCM and JavaBeans, focusing on aspects that are most relevant to our work. A complete overview of SaveCCM is presented in [2], and more information about JavaBeans can be found in [18].

2.1 SaveCCM

SaveCCM is a domain specific component model intended to provide support for designing 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 (pipes and filters) paradigm, but data transfer and control flow are separated. Thus, 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 these ports are equivalent to one trigger port and one data port.

Components represent basic units of encapsulated behavior. The functionality of a component is typically defined by an entry function, which is written in the C programming language. These are plain components. However, there are also composite components, in which the functionality is defined by an internal composition of subcomponents.

There are two additional types of components – a *clock component* and a *delay component* – which are in charge of manipulating trigger timing. A clock component is a trigger generator, and a delay component detains a trigger signal for a certain amount of time.

A component is initially *idle* and remains in that state until all its input trigger ports are activated. At that point it switches to *active* state, i.e., it has been *triggered*. This initiates the *read phase*, in which all data input port values are stored internally, to ensure consistent computation. Next is the *execute phase*, in which the computations are performed. After execution comes the *write phase*, in which data is written to the output ports of the component. Finally, the input triggers are reset and the output triggers are activated, before the component returns to the idle state. This mechanism is depicted in Figure 1. The strict "read-execute-write" semantics ensures that once a component is triggered, the execution is functionally independent of any concurrent activity.

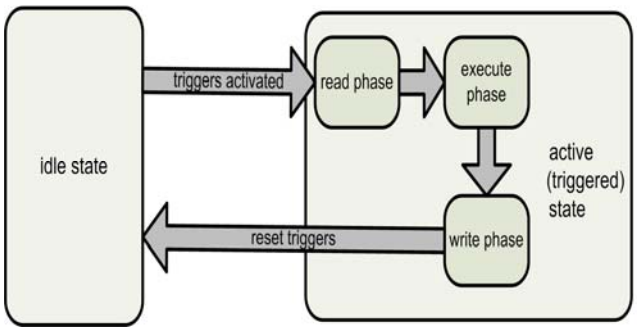

Figure 1: SaveCCM component semantics

Switches enable dynamic modification of the structure of connections between components by providing means for conditional transfer of data and/or triggering between components. *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.

According to the SaveCCM graphical notation, components are represented by rectangles with the Component stereotype. Other architectural elements (switch, assembly, clock, delay) are presented by rectangles with matching stereotypes. Trigger ports are denoted by triangles and data ports by small rectangles. Output ports are recognized by semicircles, while circles mark input ports. The notation is depicted in Figure 3 in Section 5, where we give an example of a SaveCCM system.

SaveCCM is mainly targeted at design-time, and makes no explicit assumptions about realization in its specification. Having the safety-critical hard real-time embedded systems domain in mind, the envisioned approach is realization by allocating components to tasks [3]. This enables high runtime efficiency and detailed timing analysis using standard real-time analysis techniques.

SaveCCM systems are developed using a custom development environment called SaveIDE [15], which is implemented in the form of a plugin for the Eclipse IDE [6].

2.2 JavaBeans

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 [18] which describes how classes and interfaces from the package should be used to implement the *Java bean*[2] concept. A Java bean is a Java class that complies with conditions stated in the specification.

Each Java bean has to be able to run in two different environments. First, a bean needs to be capable of running inside a builder tool, as builder tools are used for configuring beans. This is referred to as the *design environment* or *design-time*. In addition, a bean must be able to be used during *run-time* within a generated application.

Java beans are defined as reusable software components that can be manipulated visually in a builder tool. However, their use is not dependent on tools. Many beans have a visual aspect both at design- and run-time (*visual beans*), but this is not required. *Non-visual beans* are invisible at run-time, but are visible during design-time.

[2] The term "JavaBeans" stands for the technology, while the term "Java bean" or simply "bean" signifies a particular software component that conforms to the JavaBeans component model.

Individual Java beans vary in functionality, but have the following typical common features:

- properties,
- events,
- methods,
- customization,
- introspection, and
- persistence.

A *bean property* is a named attribute of a bean that can affect its behavior or appearance. Examples of bean properties include color, label, font etc. Properties can have arbitrary types, including both primitive types and class or interfaces types. Properties are accessed via method calls on the owning bean.

Beans use the Java Event Model for communication. *Events* provide a convenient mechanism for allowing beans to be plugged together in a builder tool. For a bean to be the source of an event, it must implement methods that add and remove listeners for a particular type of event. For a bean to receive an event, it must implement an event listener interface.

The *methods* of a bean are normal Java methods which can be called from other objects. A bean's methods represent its interface, through which the bean can be accessed and manipulated.

When a user is composing an application in a builder tool, he needs to be able to customize the beans he is using. *Customization* is the process of modifying the appearance and behavior of a bean within a builder tool, so that the bean meets the user's specific needs. Customization is done at design-time.

Introspection is the automatic process of analyzing a bean to reveal its properties, events and methods. Introspection is used by builder tools to provide easy and straightforward visual manipulation of beans.

Persistence refers to the characteristic of data to outlive the execution of the program that created it. The mechanism that makes persistence possible is called *serialization*. Object serialization means converting an object into a data stream and writing it to storage. A serialized object can then be reconstructed by *deserialization*. All beans are required to support serialization.

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 is restricted to follow the pipes-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, many aspects of SaveCCM have a potential to be useful in a somewhat broader scope,

Inspired by model-driven development (MDD), a methodology in which software is developed not by writing code, but by constructing high level models that can be transformed into code by automated transformation engines [14], we separate platform specific aspects of SaveCCM from those that are platform independent. This separation can also be viewed as separating domain specific from domain independent features, as most of the platform specific characteristics are conditioned by the specific domain.

In SaveCCM, platform specific aspects are found in:

- the behavior implementation of plain 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.

The behavior of plain 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 to allow Java to be used as the implementation language as well. Java is platform independent and ubiquitous, as it is widely accepted and used in a wide range from embedded systems to desktop computers.

Regarding realization, we propose JavaBeans as the target technology. The motivation for using JavaBeans comes from three directions. First, it is a platform independent technology and follows the “write-once, run-everywhere” philosophy, thus blending in well with extending SaveCCM’s scope. Second, it is compatible with the proposed component behavior implementation in Java. The third reason comes from the drawback of the current realization. Although realization by transformation to tasks is suitable for hard real-time embedded systems, it fails to keep the design-time component structure of a system at run-time. This way the CBSE approach is lost during the synthesis and CBSE benefits, such as the possibility to dynamically replace or update components, cannot be exploited at run-time. We address this by proposing a new realization of SaveCCM by transformation to a different component model, namely JavaBeans.

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 the proposed realization of SaveCCM by transformation to JavaBeans. In order to achieve the transformation, we define a mapping from SaveCCM to JavaBeans, or in other words, an object-oriented representation of SaveCCM elements in terms of JavaBeans. We name this mapping SaveJava and describe it in the following subsection. We also describe the component execution mechanism and the tool for automatic transformation.

4.1 The SaveJava classes

The terms “class” and “bean” are used equivalently throughout this subsection. What makes classes beans involves making them implement some special interfaces, and naming their methods in a certain way.

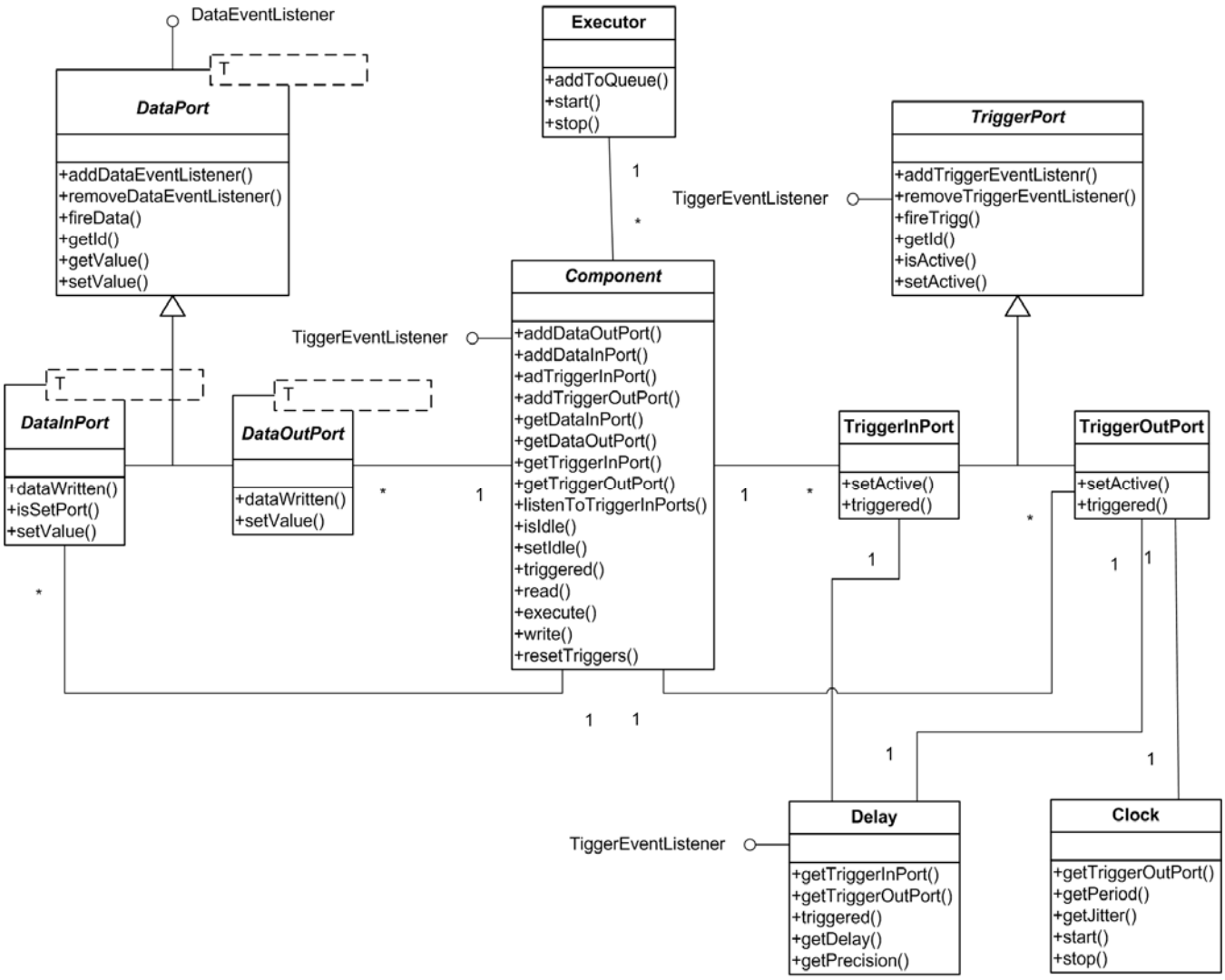

Figure 2: The SaveJava generic classes

SaveJava consists of three categories of classes:

- generic classes,
- specific classes, and
- a system class.

The generic classes make up the core of SaveJava, as they represent features common to all SaveCCM systems and are unmodified across different systems. A UML diagram of the generic classes is shown in Figure 2. The specific classes are generated during the transformation and represent aspects of the particular SaveCCM system, such as individual components and data ports of a given type. The system class is used for setting up the run-time architecture of the system realization. Its `main` method instantiates objects from generic and specific classes, according to the structure of the system.

Components are realized with a simple hierarchy. The hierarchy root is the `Component` abstract class, which represents mechanisms common to all SaveCCM components. For each component type defined in an input system, one additional specific component class is generated during the transformation, extending the generic one.

Each port is realized by an individual object, and components hold references to their ports. The alternative could have been to represent ports indirectly by methods in the component classes. However, we find the proposed solution more straightforward.

Two separate hierarchies are used to represent ports – one for data ports and one for trigger ports. Data ports are realized using Java Generics, allowing a single hierarchy between ports of different types. In addition to the ones existing prior to the transformation, additional data port classes are generated during the transformation. For instance, a data input port holding a value of string type would be represented by the `StringDataInPort` specific class which would extend the generic class `DataInPort<String>`.

The third type of SaveCCM port, combined port, becomes one data port and one trigger port in SaveJava. This is done in order not to complicate the mapping with a third type of port and is

possible because semantically, a SaveCCM combined port is equivalent to a data and a trigger port.

SaveCCM connections have no class representation in SaveJava, instead they are realized using the Java Event Model, as this is the standard way to achieve communication between beans. Connecting one port to another one is done by registering the destination port as the listener of the source port. An event type is realized by an event class and an event listener interface. In SaveJava there are two types of events, 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.

Clock components and delay components are realized by the `Clock` and `Delay` classes, respectively. Although clocks and delays in SaveCCM are special types of components, in SaveJava their classes are in no relation to the component hierarchy. However, this has no effect on the realized systems.

4.2 The component execution mechanism

The proposed component execution mechanism is a variant of the one used by Lednicki [8]. Every SaveCCM system transformed to JavaBeans will have one *executor*, an object which holds a queue of triggered components and executes them one by one, in the same order as they got triggered.

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 input data 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 by resetting its triggers. Each of these phases (read, write, execute, reset triggers) is realized by calling the corresponding component method.

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. A delay component runs in the same thread as the clock component it is connected to.

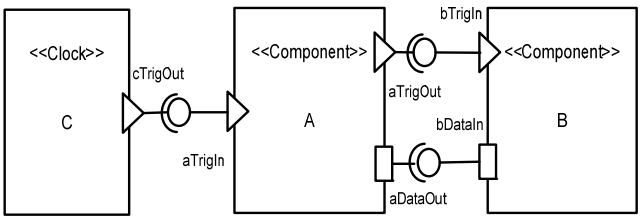

Figure 3: Example of a SaveCCM system

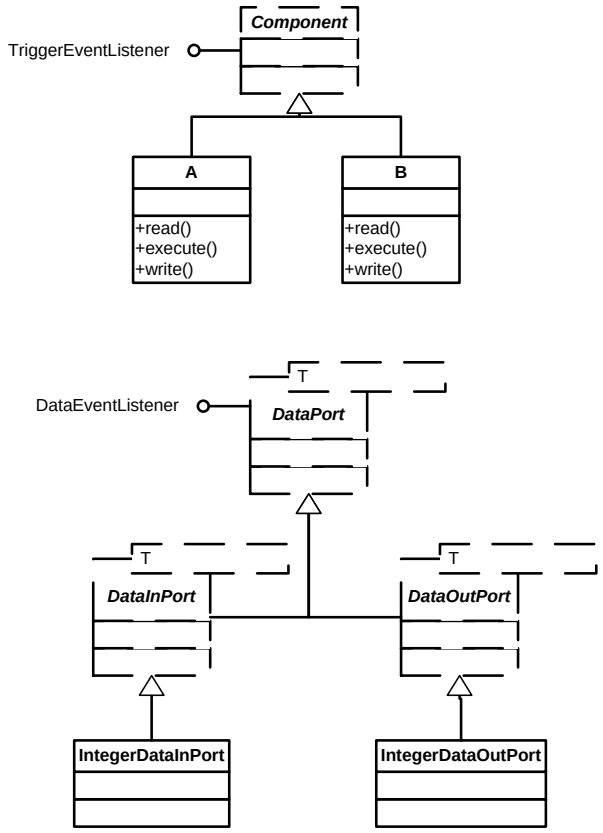

Figure 4: Realization classes

4.3 The transformation tool

Based on the SaveJava mapping, we have developed a tool that automatically performs the transformation from SaveCCM to JavaBeans. The tool takes as input a description of a particular SaveCCM system (represented by the .save file produced by the SaveIDE), and generates realization code (generic classes, specific classes and a system class) as output.

The tool is implemented in Java. For parsing the input file, we use Java Architecture for XML Binding [17], 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. However, this process can be tedious, as it requires full understanding of SaveJava.

5. REALIZATION EXAMPLE

In this section we present an example of the transformation from SaveCCM to JavaBeans. We use the simple SaveCCM system in Figure 3 as input. It consists of one clock component and two plain components. The clock C triggers the component A, which triggers component B and sends data of integer type to it.

The transformation results in four specific classes, two for the plain components A and B, and two for data ports. The clock and trigger ports are represented by generic classes.

The generated specific classes are shown in Figure 4. The methods shown in the figure are the ones being overridden in the

```
public class ExampleSystem {

  public static void main(String[] args) {

    // instantating system elements
    Executor executor = new Executor();
    Component a = new A(executor);
    Component b = new B(executor);
    Clock c = new Clock(
      100, 5, new TriggerOutPort("cTrigOut"));

    // connecting system elements
    c.getTriggerOutPort("cTrigOut").
      addTriggerEventListener(
        a.getTriggerInPort("aTrigIn"));
    a.getDataOutPort("aDataOut").
      addDataEventListener(
        b.getDataInPort("bDataIn"));
    a.getTriggerOutPort("aTrigOut").
      addTriggerEventListener(
        b.getTriggerInPort("bTrigIn"));

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

  }
}
```

Figure 5: The system class

child classes. The generic classes that are part of the hierarchy are shown with dashed lines.

As we mentioned, apart from the generic and specific classes, also the system class is generated (shown in Figure 5). In it, objects are instantiated, following the structure of the input system and connected accordingly. Objects representing ports are created in the constructor of the component to which they belong. When all objects are created, the threads of the clock and the executor are started.

6. RELATED WORK

Our work is most closely related to the work by Åkerholm et al. [3]. They define the aforementioned realization of SaveCCM by allocating components to operating system tasks. Their run-time architecture is applicable for any real-time operating system, but a particular mapping from components to tasks would have to be developed for each targeted task model. Contrasting this, our realization is applicable for any Java compliant platform in soft or no real-time domains. Their and our contributions result in SaveCCM now having two complementary realizations.

Petričić [13] also addresses the transformation of SaveCCM, by defining a transformation between SaveCCM and UML. According to the taxonomy proposed by Visser [20], 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 meta-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.

7. CONCLUSIONS AND FUTURE WORK

We have modified aspects of SaveCCM making it suitable for an expanded domain, for instance embedded systems with soft or no real-time constraints, and desktop applications. In particular, we have defined a realization of SaveCCM using the JavaBeans technology. This new realization follows the achieved domain expansion and allows for CBSE benefits to be exploited both at design-time and run-time. Thus, having in mind the addressed issues, a systematic evolution of SaveCCM has been achieved.

The current version of SaveJava does not cover all SaveCCM elements, as composite components and switches are missing. Including the remaining SaveCCM elements in SaveJava requires some amount of work, but will not contribute much to the general concept.

The executor mechanism runs components sequentially, in a non-interleaving fashion. In the future, we would like 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.

One important feature of JavaBeans is not exploited to its full potential. JavaBeans are notorious for their visual aspect, but the beans developed here are invisible. As part of future work it is worth exploring the possibility of giving these beans a visual representation, thus making them even more configurable and pluggable in a JavaBeans compliant tool. This would greatly improve usability of the new SaveCCM, eliminating the need to fully understand SaveJava if one wishes to modify the generated code.

In the paper we have discussed the possibility of using a general purpose technology to realize (implement) a system modeled in a component model used for a particular domain, namely using JavaBeans for realizing SaveCCM. Our solution makes the realized systems more general and portable, and usable outside of the original narrow SaveCCM domain.

8. ACKNOWLEDGEMENT

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

9. REFERENCES

[1] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, M. Tivoli, The SAVE approach to component-based development of vehicular systems, Journal of Systems and Software, 80:655-667, 2007

[2] M. Åkerholm, J. Carlson, J. Håkansson, H. Hansson, M. Nolin, T. Nolte, P. Pettersson, The SaveCCM Language Reference Manual, MRTC report, Mälardalen University, 2007

[3] M. Åkerholm, A. Möller, H. Hansson, M. Nolin, Towards a dependable component technology for embedded system applications, Tenth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, p. 320-328, 2005

[4] Arcticus Systems, Rubus component model, http://www.arcticus-systems.com

[5] D. D'Souza, A.C. Willis, Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998

[6] Eclipse, http://www.eclipse.org

[7] H. Hansson, M. Åkerholm, I. Crnkovic, M. Torngren, SaveCCM - A Component Model for Safety-Critical Real-Time Systems, Proceedings of the 30th EUROMICRO Conference, 627-635, 2004

[8] L. Lednicki, Component-based development for software and hardware components, master thesis, Mälardalen University, 2008

[9] R. Marvie, MDA, Model Transformations and Platforms: Advocating Technological Jumps, LIFL research report, 2004

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

[11] OMG, CORBA Component model, http://www.omg.org/technology/documents/formal/components.htm

[12] R. van Ommering, F. Van der Linden, K. Kramer, J. Magee, The Koala Component Model for Consumer Electronics Software, IEEE Computer, 33(3):78-85, 2000

[13] A. Petričić, UML profile for SaveComp Component Model, master thesis, Mälardalen University, 2008

[14] H.N. Pham, Q.H. Mahmoud, A. Ferworn, A. Sadeghian, Applying Model-Driven Development to Pervasive System Engineering, Software Engineering for Pervasive Computing Applications, Systems, and Environments, p. 7, 2007

[15] S. Sentilles, J. Håkansson, P. Pettersson, I. Crnkovic, Save-IDE – An Integrated development environment for building predictable component-based embedded systems, 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008

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

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

[18] Sun Microystems, Java SE Desktop Technologies - JavaBeans, http://java.sun.com/javase/technologies/desktop/javabeans

[19] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 2002

[20] E. Visser, A Survey of Strategies in Program Transformation Systems, Electronic Notes in Theoretical Computer Science, Elsevier, 2001

[21] M. Winter, C. Zeidler, C. Stich, The PECOS Software Process, Workshop on Components-based Software Development Processes, 2002.