Adding Support for Hardware Devices to Component Models for Embedded Systems

Luka Lednicki, Mario Žagar
Faculty of Electrical Engineering and Computing
University of Zagreb
Croatia
{luka.lednicki, mario.zagar}@fer.hr

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

Abstract—**Component-based development promises many improvements in developing software for embedded systems, e.g., greater reuse of once written software, less error-prone development process, greater analyzability of systems and shorter time needed for overall development. One of the aspects commonly left out of component models is communication of software components with hardware devices such as sensors and actuators. As one of the main characteristics of embedded systems is the interaction with their environment through hardware devices, the effects of this interaction should be fully included in component models for embedded systems. In this paper we present a framework that enables inclusion of hardware devices in different phases of the component-based development process, including system design, deployment, analysis and code synthesis. Our framework provides a way for software components to explicitly state their dependencies on hardware devices, promotes reuse of software components with such dependencies and provides a basis for including hardware devices in analysis of component based embedded systems. We evaluate the feasibility of our approach by applying it to the ProCom component model.**

Keywords – Component-based Development, Embedded Systems, Hardware devices, platform modeling

I. Introduction

Embedded systems are getting increasingly important in our daily lives, while at the same time getting more complex. Additionally, larger portions of functionality of embedded systems are being put into software, rather than hardware, which results in increased software complexity. Parallel with this trend there is a growing demand on software to be robust, reliable, flexible, adaptable, etc., while shorter time-to-market is desired. One of the approaches to tackle these issues is component-based software engineering (CBSE). CBSE promotes building systems from prefabricated software components, instead of coding from scratch, promising to lower time-to-market, manage complexity and produce software of higher quality. CBSE has proven to be successful in the domains of desktop- and Web applications and enterprise systems. However, embedded systems introduce some domain-specific issues (e.g., safety-criticality, real-time requirements, interaction with the environment), and to fully take advantage of the CBSE potential these must be addressed [1].

In this paper, we focus on enriching existing component models with support for proper handling of the interaction between a software system and its environment, the physical world that the system is embedded into. This interaction is done using *hardware devices*, such as sensors and actuators. The communication between software and hardware devices can be as simple as writing a value to a hardware pin or port, or as complex as invoking a service on a remote device. In all cases, this interaction with the environment implies that software components are dependent on the hardware or middleware used to communicate with the environment. As this affects reusability and analyzability of software components, failure to adequately express these dependencies can hinder the use of a component-based approach in the embedded system domain.

To address the problem of interaction between software components and hardware devices, we have investigated what is needed to properly integrate such devices into software component models for embedded systems, and devised a framework that allows us to describe hardware devices and hardware platforms that we can deploy software systems on, software components dependent on hardware devices. The framework also allows describing a mapping between hardware devices, hardware platforms and software components. Our approach has been developed in the context of ProCom component model [2], but is also applicable to other component models.

In Section II, we describe different ways in which hardware devices can impact the use of a component-based approach when developing software systems for the embedded domain. Section III provides an overview of how interaction of software components with hardware devices is managed in some of the existing component models. Our approach to inclusion of hardware devices in component models is presented in Section IV. Section V gives an example of how our approach can be used in developing software systems that interact with hardware devices, and Section VI concludes the paper.

II. Effects of Hardware Devices on Software Component Models

Dependencies of software components on hardware devices, as well as the communication between hardware and software impact all phases of a component-based development process. In this section we discuss these impacts, in order to be able to address them accordingly. We consider a component-based development process suitable for developing embedded systems, and comprising the following phases: design, deployment, analysis and synthesis. The phases are not strictly sequential and can be iterative.

In the *design* phase, a developer specifies models of (i) the software layer of the system being developed, as a composition of components, and (ii) the hardware layer, as a composition of the hardware devices the system will be deployed on. The former requires a means to manage interaction with hardware devices in the software layer. The latter requires a means to describe the actual instances of hardware devices and how they are connected to a particular instance of a hardware platform.

In the *deployment* phase, a mapping between the software- and hardware layers is defined. In other words, the software components are allocated to the underlying hardware that will execute them. In this phase we must be able to explicitly identify the dependencies of software components on the hardware devices, in order to ensure that the hardware targeted for deployment satisfies these dependencies.

Embedded systems have particularities such as limited resources and real-time requirements, which increase the relevance of extra-functional properties compared to, for example, desktop- and Web applications. In order to guarantee constraints on extra-functional properties, extensive analysis has to be performed. During the *analysis* phase, effects of the hardware devices on the behavior of the software components must be taken into consideration.

During the *synthesis* phase executable code is generated based on the models specified in the design- and deployment phases. During the synthesis we must ensure that the code generated for software components reflects the specifics of the platform, with respect to communication with hardware devices.

As *reuse* is one of key concepts of CBSE, additionally we consider the effects hardware has on the ability to reuse components developed in different contexts. For successful reuse, we must ensure that components dependent on hardware can be deployed on different platforms.

With regards to the aforementioned concerns, the objectives of our work are to:

- provide means to describe hardware elements in a way that they can be integrated into component models for embedded systems;
- enable specification how software components depend on hardware devices, and description of communication between the two;
- allow inclusion of both functional and extra-functional properties of hardware devices and physical platform in analysis of component-based software systems;
- enable analysis of systems in early stages of development, before they are fully implemented; and
- promote reuse of both software components and hardware device descriptions.

III. Background and Related Work

We have identified four different levels of support for hardware dependencies in a component-based context.

A. *Outside of the Component Model*

Many component models, especially those developed for research purposes, do not provide any method for including hardware devices in system design. All communication with the environment is performed at input and output at the top level of the system. In this approach, functionality must be modeled separately from hardware interaction. Therefore, functionality specifically developed to fit particular hardware is difficult to represent. Furthermore, propagating all hardware interaction to the top level can be particularly cumbersome in complex systems, where many nesting levels exist.

SaveCCM [3] is an example of such a component model. In SaveCCM software components are not allowed to directly communicate with hardware devices. Instead, communication with them takes place outside of the component model.

B. *Code Level*

Many component models do not provide ways to explicitly state dependencies on hardware devices. However, they allow to communicate with them in the code of software components through direct method calls to the underlying platform. This approach can severely limit reuse of software components, as components with such hard-coded communication with hardware cannot be used on multiple hardware platforms or when the configuration of the hardware platform is changed.

An example of such a component model is Rubus [4]. Rubus was created by Articus Systems for developing dependable real-time systems. Reuse is not the main focus of Rubus, rather it is to provide a higher abstraction layer and better basis for analysis. Thus platform and device dependent information are part of basic software components.

C. *Using Specialized Entities*

Some component models introduce new entities, separate from software components, which are used to interact with hardware devices. With a way to explicitly describe dependencies and communication with hardware devices, and a clear separation of hardware and software components we can easily reuse parts of systems or include hardware devices in analysis of systems. A drawback of this approach is that it hinders the possibility of hierarchical component composition. As components cannot specify their interaction with hardware devices through their interface, we cannot reuse composite components that contain hardware entities.

A component model that uses this approach is COMDES-II [5]. COMDES-II provides a two-layered component model. The upper layer a system is defined by active software components named *actors*. The lower layer is used to define the behavior of actors using *function block instances*. Actors interact with hardware devices using entities called *input* and *output signal drives*. Drives can be used to communicate over a network (*communication drivers*) or to sense or actuate physical signals (*physical drivers*).

AUTOSAR [6], also provides similar level of support. AUTOSAR is a component-based architecture created by a partnership of a number of automotive manufacturers and suppliers. Dependencies on hardware devices are encapsulated in *sensor* and *actuator* software components. These components provide a special interface for managing their interaction with hardware devices. They are dependent on specific sensor or actuator hardware devices. However, AUTOSAR does not provide means for hierarchical composition of components. As it does not provide support to state hardware dependencies for all component types we still argue that sensor and actuator components act as specialized entities.

D. Explicitly Encapsulated in Software Components

Component models can also encapsulate communication with hardware devices in software components, but expose it through the component's interfaces. Compared to approaches that use specialized entities for interaction with hardware devices, this approach enables us to organize components dependent on hardware devices in multiple levels of hierarchy

Our approach also falls into this category since it provides an explicit way to define how software components are connected to hardware devices. For this we do not use specialized entities, but instead extend the definition of standard software components. This lets us reuse all parts of component model framework and tools while including hardware devices in software component and system definition.

IV. Overview of our Approach

Led by the objectives described in Section II, we have devised a framework that allows us to include hardware devices in component models, and applied it to the ProCom component model.

The ability to reuse components or complete systems is one of the main goals of CBSE. Having components that are dependent on a particular instance of hardware device, or how this device is connected to the platform, can severely limit possibility of their reuse. For this reason we have separated our framework in three layers: *software layer*, *hardware layer* and *mapping layer*. With this separation we are able to independently describe software system and hardware platform, making them suitable for reuse in different scenarios. We can then connect these two layers through the mapping layer when developing a complete system. An overview of how these three layers are connected is given in Figure 1.

In our approach we have a clear distinction between *types* and *instances* for both hardware and software entities. Types are entity definitions that are context-independent. They can be easily reused in different settings or stored to repositories for future use. Once we want to use an entity in a concrete system, we are in fact creating an instance of that entity type. Instances are not copies of the entity, but a representative of the general entity in a specific context. For example, when we are describing a hardware device, we are actually describing a device type. Once we want to use the device in a system we need to create a new instance of that device type. Instances can also refine properties of an entity depending on the usage context.

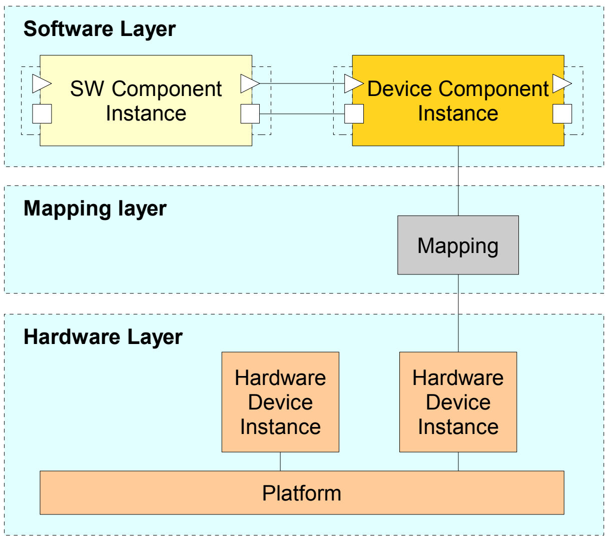

Figure 1: Overview of three layers of our approach relate to each other.

As we handle hardware devices using extended software components, and not specialized entities, we are able to reuse many solutions that already exist in ProCom component model. For the purpose of defining attributes for hardware components we leverage Attribute Framework [7], which allows us to define extra-functional properties for architectural elements of the component model. Also, integration with ProCom allows us to use ProCom Analysis Framework with different types of analysis, such as parametric worst-case execution time analysis [8], model checking of behavioral models [9] and fault-propagation.

A detailed metamodel that describes our approach is given in Figure 2. Next, each of the three layers will be described in more detail.

A. Software Component Layer

To enable interaction of component-based applications with hardware devices we have introduced a new type of component named *device component.* This entity is derived from ordinary software components. Its purpose is to encapsulate dependencies of component-based software system on hardware devices and enable communication with these devices.

When looking at a device component as a black-box, it has the same interface and semantics as all software components. The difference between normal software components and device components is in their internals: device components do not provide the ability for the developer to explicitly specify their realization. This is because they inherit their realization from hardware devices (described in Section IV.B.2)) once the two are mapped together.

Device components are only used to express the existence of dependencies on hardware devices, but not the specifics of a device, i.e., how it is connected to the platform or the code for actual communication with hardware. A device component has exactly one hardware dependency. In case of composite components, its device dependencies must mach the combined dependencied of its subcomponents. This way the software layer stays hardware- and platform-independent. Any system or composite component that contains device components can still easily be reused in a new system or on different platforms.

B. Hardware Layer

The hardware layer allows us to describe *physical nodes* (i.e., processing unit such as microcontrollers or ECUs that runnable code can be deployed to), *hardware devices* such as sensors and actuators and *platforms* which consist of instances of physical nodes and hardware devices and to which we can deploy software systems.

We have designed the hardware layer based on research of what is needed to promote the ability of reuse of software components. However, we also wanted to provide the ability to reuse structures defined in hardware platform. For this purpose we have divided hardware into three separate parts which can be developed independently to each other: *physical node specification*, *hardware device specification* and *platform instantiation*.

1) Physical Node Specification

In our model, physical nodes describe different processing units such as microcontrollers or ECUs. They are reusable as they only describe a type of unit and do not contain any information about how they are used or configured in a particular system.

Physical nodes define a list of inputs and outputs they provide. Inputs and outputs are defined by their type, e.g., one-bit digital I/O, serial communication port, analogue input, etc. Also, for each input or output we define actual program code that will be used for its initialization and data transfer.

Physical nodes can also be characterized by extra-functional properties such as their processing power, available memory, behavioral models, execution times for input or output functions and other similar attributes.

2) Hardware Device Specification

Hardware devices are peripherals such as sensors and actuators that are connected to physical nodes in order to interact with the environment. Each hardware device represents a specific, real-world sensor or actuator.

Each hardware device references a device component for which the device can be used as realization. It should be noted that one device component can be referenced by many different hardware devices. For example, a temperature sensor device component can be referenced by two different implementations of (i.e., hardware devices) temperature sensor. However, a device component (in the software layer) is not dependent on any of these implementations.

Similarly to a list of inputs and outputs provided by physical nodes, hardware devices define a list of inputs and outputs that they require for communicating with them.

A part of hardware device specification is the code for communication with the device. This code is merged with software component code during the synthesis phase of the development process, leaving software components free of hardware-specific code. In that way software components can be reused on different hardware configurations. However, this code leaves out actual function calls needed for communication, which is defined in the physical node specification. This allows us to reuse the same code regardless of which input or output of a physical node the device is connected to, or use it on different physical nodes.

Similar to physical nodes, we can also define attributes that describe extra-functional properties of hardware devices.

3) Platform Instantiation

We have defined platform as a collection of physical node instances on which we can deploy software systems. Except creation of physical node instances, platform instantiation also encompasses creation of hardware device instances and connections of these instances to instances of physical nodes. It should be noted that we do not use type-instance paradigm

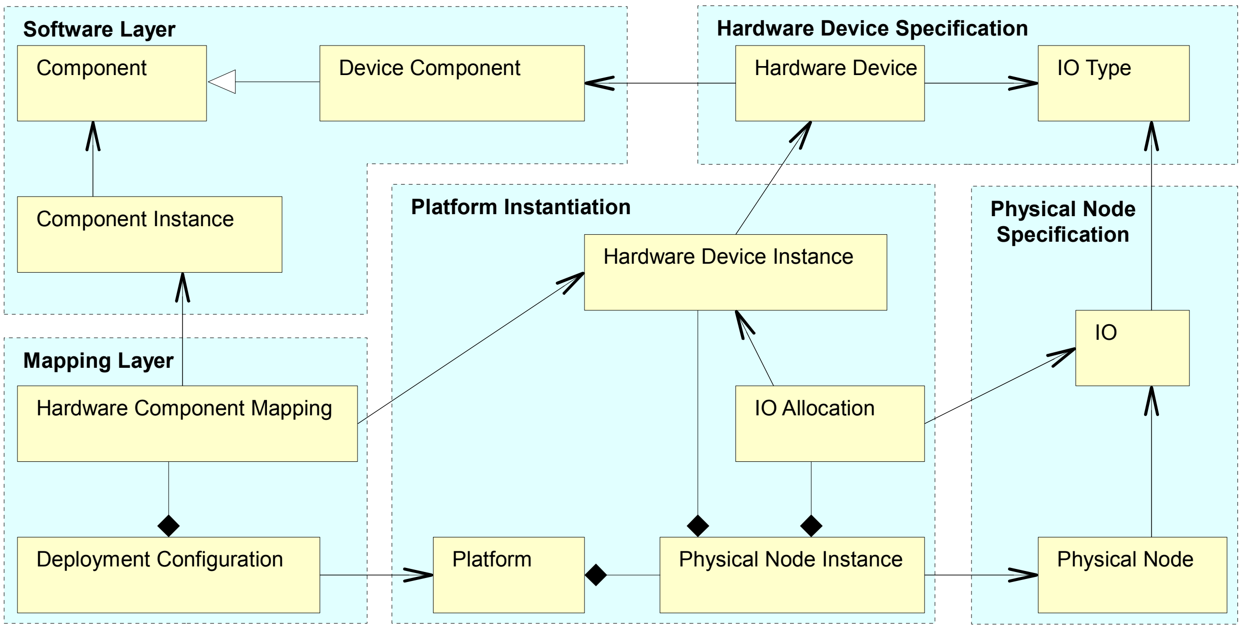

Figure 2: Metamodel that contains all entities we use to add support for hardware devices in software component models.

for platform. We assume that platforms will be collections of reusable physical nodes, and will be specific for every system, there will be no need for their reuse.

Connections between hardware devices instances and physical node instances are implicit: device instances are contained by physical node instances. Allocation of hardware device instances to inputs or outputs of physical node instances is done through *IO Allocation*. Once the allocation of inputs and outputs is defined, we can also validate a platform by checking if requirements of all hardware device instances are fulfilled by inputs and outputs of physical node instances they are connected to.

C. Mapping Layer

As already stated, we have defined software and hardware layers to be as distinct as possible in order to promote reuse of structures defined in them. In order to create systems consisting of both, we had to introduce the mapping layer. The mapping layer allows us to define connections between device component instances in software layer and hardware device instances in hardware layer. By this we put our reusable units in the context of a system and are able to provide platform-specific code for platform independent, reusable software components.

Mapping between the two can be created only if type of hardware device instance references type of device component instance. By having this constraint we can easily assure that a system is deployed (i.e., component instances are allocated to physical node instances) in a valid way.

Besides the platform-specific code, the mapping also allows us to propagate platform- or device-specific values for extra-functional properties.

Our approach supports mapping of component instances to hardware device instances even in early stages of system development. By having reusable descriptions, models and extra-functional properties defined for hardware devices and physical nodes we are able to test and analyze behavior of a system before it is fully implemented. This allows us to detect potential problems and avoid changes in late stages of system development.

Another benefit of separate mapping model is that it allows a more flexible process, where software and hardware can be addressed separately in any order, and interleaved. Also, it enables us to provide partial mappings in early stages of development.

V. Example

To illustrate use of our approach, we will demonstrate it on an example. The example will model a simple temperature control system using ProCom component model.

A. The ProCom component model

ProCom is a component model for distributed embedded systems in the vehicular and automation domains. These systems often have a safety-critical role and have to perform in real-time. Therefore, ProCom explicitly addresses extra-functional properties such as timing (e.g., worst case execution time) and resource usage (e.g., static memory, CPU). ProCom follows a model-based methodology centered around a rich notion of reusable architectural design-time components. A ProCom component can consist of source code, models of timing and resource usage, analysis results and documentation.

The external view of a component consists of ports and attributes. Through the ports the functionality provided by a component can be accessed, while the attributes represent additional information about a component, such as extra-functional properties.

In order to be able to design both the complete system and the low level control functionality, ProCom has been divided into two layers. The upper layer, called ProSys, models a system as a collection of complex, active, concurrent, and typically distributed subsystems that communicate via asynchronous message passing. The lower layer, ProSave, on the other hand models smaller parts of control functionality. ProSave components communicate through trigger (control flow) and data ports (data flow).

B. Temperature Control System

Our example temperature control system consists of two temperature sensors that monitor temperature in a water tank and a heater that will engage if the temperature drops below a defined temperature. A graphical representation of all software and hardware layers of the system, and the mapping between the two layers, is given in Figure 3.

Our software layer consists of a clock (an element that creates periodical triggering signals), two instances of TemperatureSensor device component (TS1 and TS2), one instance of ControlUnit software component (CU1) and one instance of HeaterActuator device component (HA1). The component instances are connected in such a way that the clock triggers both TS1 and TS2. When both of them have finished their execution they forward temperature values to CU1 and generate signals that trigger its execution. Depending on given temperature values, CU1 performs calculations and provides signals to HA1 to be turned on or off.

It should be noted that TS1, TS2 and HA1 just serve just for describing interaction of software components with hardware devices, but are not device-specific. In that way whole software layer is reusable on different hardware platform configurations.

In the hardware layer we need to include specifications of physical nodes and hardware devices, and instantiate our platform. For the purpose of this example we will not fully specify the hardware but will only use parts that satisfy the needs of our system. Physical node specification will consist only of one physical node which we will call MicroCrtl. MicroCtrl will provide three IOs: two analog and one digital. For temperature sensors we use hardware devices that require analog input. We also specify heater hardware device which requires digital output. To instantiate our platform, we will create an instance of MicroCtrl with name Micro1. Micro1 will have two instances of the analog temperature sensor device (AT1 and AT2) and one instance of the heater device (H1). We will allocate the instances of temperature sensor to the analog inputs and the instance of heater device to the digital output of Micro1.

To complete our system, we need to define mappings between device components in software layer and hardware

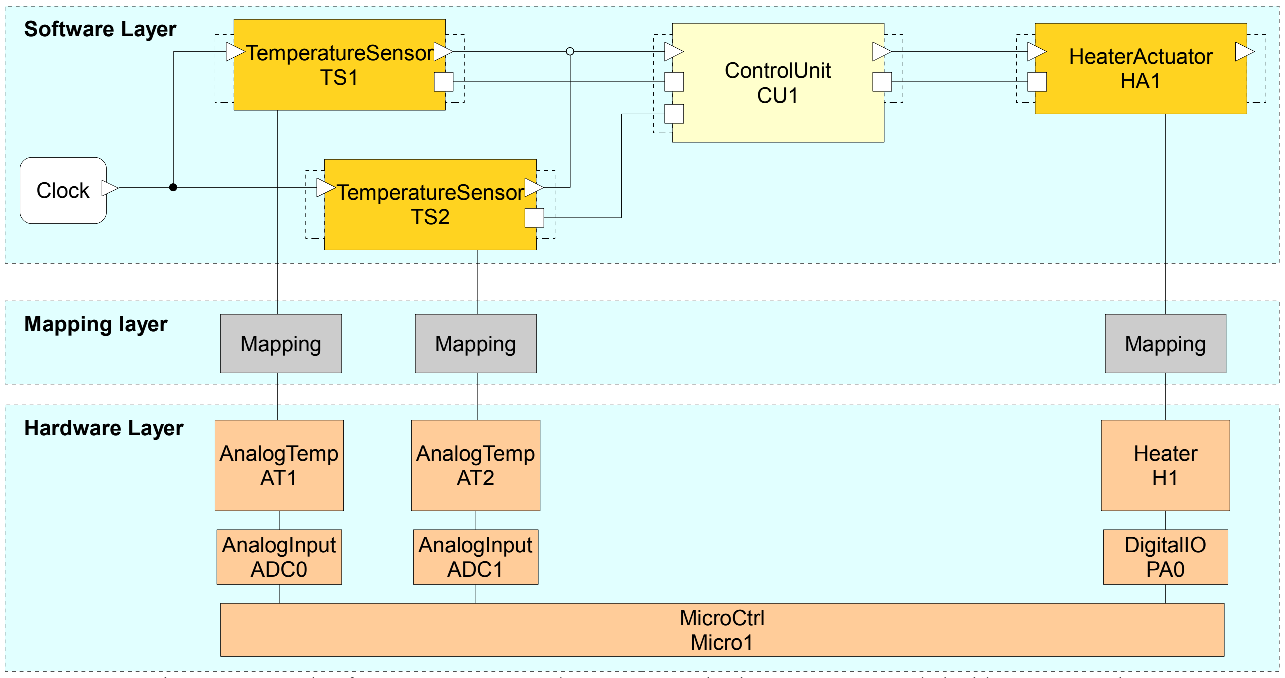

Figure 3: Example of a temperature control system created using ProCom extended with our approach.

devices in hardware layer. For this, we will define mappings between AT1 and TS1, AT2 and TS2, and HA1 and H1.

VI. Conclusion

In this paper, we have presented our approach for managing hardware devices such as sensors and actuators in component models for embedded systems. Our framework consists of three layers: software layer, hardware layer and mapping layer. These three layers enable separation of device dependencies in software and models of the actual hardware and allows us to reuse software components and hardware models. The hardware layer enables us to specify all aspects of hardware devices and platforms needed for their integration into component models. In the software layer we enable explicit definition of dependencies of software components on hardware devices. The mapping layer enables us to connect instances of software components to hardware device instances and in that way to design complete systems including software and hardware. The mapping also allows propagation of extra-functional properties of hardware devices to component model. In early stages of system development we can also define just partial mappings. Our approach promotes reuse of software components, hardware device specifications and platform node specification by creating clear distinction between types and instances of these entities, and by removing platform- and device-specific code out of software components.

Acknowledgment

This work was supported by the Unity Through Knowledge Fund via the DICES project, the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and the Swedish Research Council project CONTESSE (2010-4276).

References

[1] I. Crnković and M. Larsson, Building Reliable Component-Based Software Systems, Artech House Publishers, 2002.

[2] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic. A component model for control-intensive distributed embedded systems. In 11th International Symposium on Component Based Software Engineering. Springer Berlin, October 2008., pp. 310-317

[3] 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, May 2007. pp. 655-667

[4] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K- Lennart Lundbäck, The Rubus Component Model for Resource Constrained Real-Time Systems, 3rd IEEE International Symposium on Industrial Embedded Systems, 2008, pp. 177-183

[5] K. Xu, S. Krzysztof, and A. Christo, COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems, RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007, pp. 199-208

[6] H. Heinecke, W. Damm, B. Josko., A. Metzner, H. Kopetz, A. Sangiovanni-Vincentelli., and M. Di Natale, Software Components for Reliable Automotive Systems, Design, Automation and Test in Europe, 2008, pp. 549-554

[7] S. Sentilles, P. Stepan, J. Carlson, and I. Crnković, Integration of Extra-Functional Properties in Component Models, 12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582, Springer Berlin, East Stroudsburg University, Pennsylvania, USE, June, 2009, pp. 173-190

[8] T. Leveque, E. Borde, A. Marref, and J. Carlson, Hierarchical Composition of Parametric WCET in a Component Based Approach, In 14th IEEE Int. Symposium on Object/Component/Service-oriented Real-time Distributed Computing, 2011, pp.261-268

[9] D. Ivanov, M. Orlic, C. Seceleanu, and A. Vulgarakis, REMES tool-chain – A set of integrated tools for behavioral modeling and analysis of embedded systems. In 25th IEEE/ACM International Conference on Automated Software Engineering, 2010, pp. 361-362