Run-Time Component Integration and Reuse in Cyber-Physical Systems

Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, Jiří Kunčar
Mälardalen Real-Time Research Centre
Västerås, Sweden
Email: rafia.inam@mdh.se

December 9, 2011

Abstract

We present the concept of runnable virtual nodes as a means to achieve predictable integration and reuse of software components in cyber-physical systems. A runnable virtual node is a coarse-grained real-time component that provides functional and temporal isolation with respect to its environment. Its interaction with the environment is bounded both by a functional and a temporal interface, and the validity of its internal temporal behavior is preserved when integrated with other components or when reused in a new environment. Our realization of runnable virtual nodes exploits the latest techniques for hierarchical scheduling to achieve temporal isolation, and the principles from component-based software-engineering to achieve functional isolation. In the paper we present a proof-of-concept case study, implemented in the ProCom component-technology executing on top of FreeRTOS based hierarchical scheduling framework.

1 Introduction

A contemporary Cyber-Physical System (CPS) is often required to monitor and control several disparate variables in its environment. From a development point of view, it often makes sense to develop the different control-functions as separate software-components [11]. Typically, these components are first developed and tested in isolation, and later integrated to form the final software for the system. Furthermore, many industrial systems are developed in an evolutionary fashion, reusing components from previous versions or from related products. It means that the reused components are re-integrated in new environments.

When multiple components are deployed on the same hardware node, the emerging timing behavior of each of the components is typically unpredictable. For a cyber-physical system with real-time constraints, this means that a component that is found correct during unit testing may fail, due to a change in temporal behavior, when integrated in a system. Even if a new component is still operating correctly in the system, the integration could cause a previously integrated (and correctly operating) component to fail. Similarly, the temporal behavior of a component is altered if the component is reused in a new system. Since this alteration is unpredictable as well, a previously correct component may fail when reused.

While using the temporal models of component behavior and requirements, some of the problems may be mitigated by using scheduling analysis [32, 34], however these techniques only allow very simple models; typically simple timing attributes such as period and deadline are used. In industry, components often exhibit a too complex behavior to be amenable for the scheduling analysis. And, even if a suitable analysis technique should exist, such analysis requires knowledge of the temporal behavior of all components in the system. Thus, a component cannot be deemed correct without knowing which components it will be integrated with. Further the reuse of a component is restricted since it is very difficult to know beforehand if the component will pass a schedulability test in a new system.

For complex real-time CPS, methodologies and techniques are required to provide not only functional isolation but also temporal isolation so that the run-time timing properties could be guaranteed. Further the real-time properties of the components should be maintained for their reuse in large-scale industrial CPS.

To remedy this situation we propose the concept of a *runnable virtual node*, which is an execution-platform concept that preserves temporal properties of the software executed in the virtual node [10]. It introduces an intermediate level between the functional entities and the physical nodes. Thereby it leads to a *two-level deployment process* instead of a single big-stepped deployment; i.e. deploying functional entities to the virtual nodes and then deploying virtual nodes to the physical nodes.

The virtual node is intended for coarse-grained components for single node deployment and with potential internal multitasking. The idea is to encapsulate the real-time properties into model-driven reusable components-based systems to achieve not only the predictable integrations and reusability of those components [10, 19] but also maintenance, testing, and extendibility. To achieve this, the timing properties of the components should be preserved so that real-time components integration and reuse can be made predictable.

Hierarchical Scheduling Framework (HSF) [14, 20] is known as a technique for providing temporal isolation between applications in the real-time community. Recently, HSF is proposed to develop complex CPS by enabling temporal isolation and predictable resource usage of CPS software [23]. In this paper, we integrate HSF within a component technology for embedded real-time systems; to realize our ideas of guaranteeing temporal properties of real-time components, their predictable integrations and reusability. We introduce the runnable virtual node, which includes the executable representation of the components (i.e. a set of tasks), a resource allocation, and a real-time scheduler to be executed within a server in the HSF. The server executes with a guaranteed temporal behavior, using its allocated CPU bandwidth, regardless of any other execution on the physical node. Thus, once a server has been configured for the virtual node, its real-time properties will be preserved when the virtual node is integrated with other virtual nodes on a physical node, or when a virtual node is reused in another context.

Contributions

The work presented in this paper is within the context of ProCom component technology [31]. The main contributions of this paper are as follows:

- We *realize the concept of runnable virtual nodes* by embedding the HSF implementation within the ProCom component technology for an embedded platform running FreeRTOS operating system [16].

- We *introduce a two-level deployment process* instead of a single big-stepped deployment; i.e. deploying functional entities to the virtual nodes and then deploying virtual nodes to the physical nodes, thereby preserving the timing properties within the components in addition to their functional properties.
- We *provide a case study* as a proof of concept and run it on an AVR 32-bit board EVK1100 [15].
- We *test the runnable virtual node's real-time properties* for temporal isolations and reusability.

Outline: Section 2 presents the related work on component-based and model-driven component systems. Section 3 gives an overview about the ProCom Component Model and the HSF implementation. In Section 4, we describe the runnable virtual node and how it is used within the ProCom technology. Section 5 presents a case-study in which runnable virtual nodes have been used. Finally, Section 6 concludes the article.

2 Related Work

In this section, we describe some contemporary component-technologies available for embedded systems. Especially we focus on the deployment and integration of components and on the predictability in the time-domain of the resulting systems. The list is by far non-exhaustive, but rather focuses on state-of-the-art within industrial applications. In the academic domain several similar concepts exist; but to the best of our knowledge, no other technology employs a two-phase deployment process or employs HSF to support temporal isolation, ease of integration or component reuse.

2.1 AUTOSAR

AUTomotive Open System ARchitecture (AUTOSAR) [4] is an open standard for automotive electronics architectures. It is developed to deal with the increasing complexity and to fulfil a number of future vehicle requirements (such as safety, availability, driver assistance, infotainment etc.). The key features of AUTOSAR are modularity, configurability, standardized interfaces and a runtime environment.

Functional software is developed using component-based approach [1]. A component is developed over many layers of AUTOSAR, including: Application layer, Runtime Environment (RTE), Basic software and ECU (Electronic Control Unit, is a node in an automotive network) hardware. A Software Component (SW-C) at ECU level contains at least one or several runnable entities. A runnable is a small fragment of sequential code within a component [5]. In AUTOSAR, the deployment begins when the RTE generator maps all runnables to the OS tasks and builds inter- ECU and intra-ECU communications among them. After mapping, the RTE generator configures each ECU and constructs the OS tasks bodies.

A main disadvantage of AUTOSAR is that it lacks clear and well-defined timing properties that further affect the execution semantics too. On the other hand, ProCom puts special focus on such requirements right from the beginning of the component's development untill the component's deployment. The runnable virtual nodes of ProCom are reusable components preserving timing properties in them and are independent of the platform specifics. As compared to AUTOSAR, ProCom clearly distinguishes between the data and control flows among ProCom components. Deployment

in AUTOSAR is a single-stepped process and the executables are generated directly from the components, unlike ProCom where the deployment is performed in two steps at both modelling and synthesis levels. A tool suite to support the complete AUTOSAR methodology for hard real-time systems is still missing.

2.2 Rubus

The Rubus Component Model (RCM) [17] is developed in cooperation between Arcticus Systems [2] and Mälardalen University. It is used commercially in the automotive industry. In many aspects RCM is similar to the ProCom: for example rubus captures the functionality at two-levels of component hierarchy; at first level the components are passive while at the second level they are active; manages different ports for control and data flows. The component technology uses a graphical design tool and a scheduler for system design, and some plug-ins to perform analysis. Finally, the run-time infrastructure is generated using Execution Models (EM) for the desired platform.

In Rubus, the real-time requirements of the components are realized by the use of EMs, which are logical objects and are defined in the infrastructure at the run-time environment. The RCM is not restricted to the use of a specific run-time environment as long as the components preserve the semantics defined in them. However, the current task set can only be executed in the RubusRTOS [3].

Its main difference from the ProCom technology is at the deployment and at the execution levels. In Rubus, the deployment is a single-stepped process for the desired platform. On the other hand, in Procom, the deployment is a two-stepped process. In Rubus, the required hardware components are directly modeled in the Rubus components, unlike ProCom where the components are developed independently from the hardware details and the hardware specifics are taken care at the last step of the deployment process. Another difference is the platform dependence of the Rubus technology on the underlying operating system [17]. The platform on which the component has to be executed is modeled within the Rubus components and the final executables are generated only for that particular platform.

2.3 AADL

Architecture Analysis and Design Language (AADL) was developed as a SAE Standard AS-5506 [29] to design and analyze software and hardware architectures of distributed real-time embedded systems. Modeling of software and hardware parts is supported by software components and execution platform components respectively [22]. Properties and new functional aspects can be attached to the elements (e.g., components, connections) using the properties defined in the SAE standard, and communication among components is performed using component interfaces i.e., ports. Ocarina [18] is a tool suite that facilitates the design of AADL models and their mapping on a hardware platform, assessment of these models, automatic code generation, and deployment.

Deployment in AADL is supported by a middleware API called PolyORB (PolyORB for code generation in Ada while PolyORB-HI for code generation in C). Runnable entities are presented by processes. A process contains many tasks and it is a self-contained runnable entity that executes on a hardware platform.

Unlike ProCom, the deployment in AADL is done in a single step to directly execute the generated code on the physical platform. Another type of runnable entity for

AADL employs a hierarchical scheduling concept in a partition. A partition is a combination of several processes and a scheduler called Virtual Processor [13]. But this kind of runnbale entity is also deployed in a single step. It provides temporal partitioning like runnable virtual node of ProCom, but it does not consider the reusability aspect of the runnable components.

2.4 Deployment and Configuration specification

The Deployment and Configuration specification (D&C) [26] is standardized by the Object management Group. Its main purpose is to facilitate the deployment of component-based applications onto target platform. It uses a Platform Independent Model (PIM) for the model components with three level, and a Platform Specific Model (PSM) for the CORBA Component Model (CCM).

Recently, an extension has been proposed to support the development of applications with real-time properties and provide a deployment plan, called RT-D&C [27]. The metadata about the temporal behaviour of components is added to the specification at the PIM level to facilitate the real-time analysis of the components. However, a RT-planner, who configures the real-time application after using the real-time analysis tools, is required to assign the timing properties to the application which is different from the ProCom deployment. As compared to ProCom where the timing properties are preserved within the runnable virtual nodes, RT-D&C only preserves them till the components development at PIM level.

3 Background

This section presents the background technologies our work uses. We provide an overview of the ProCom component technology, followed by and introduction of the Hierarchical Scheduling Framework, and its implementation in FreeRTOS.

3.1 ProCom Component Model

Component-Based Software Engineering (CBSE) and Model-Based Engineering (MBE) are two emerging approaches to develop embedded control systems like software used in trains, airplanes, cars, industrial robots, etc. The ProCom component technology combines both CBSE and MBE techniques for the development of the system parts, hence also exploits the advantages of both. It takes advantages of encapsulation, reusability, and reduced testing from CBSE. From MBE, it makes use of automated code generation and performing analysis at an earlier stage of development. In addition, ProCom achieves additional benefits of combining both approaches (like flexible reuse, support for mixed maturity, reuse and efficiency tradeoff) [10].

The ProCom component model can be described in two distinct realms: the modeling and the runnable realms as shown in Figure 1. In Modeling realm, the models are made using CBSE and MBE while in runnable realm, the synthesis of runnable entities is done from the model entities. Both realms are explained as follows:

3.1.1 The Modeling Realm

Modeling in ProCom is done by four discrete but related formalisms as shown in Figure 1. The first two formalisms relate to the system functionality modeling while the later

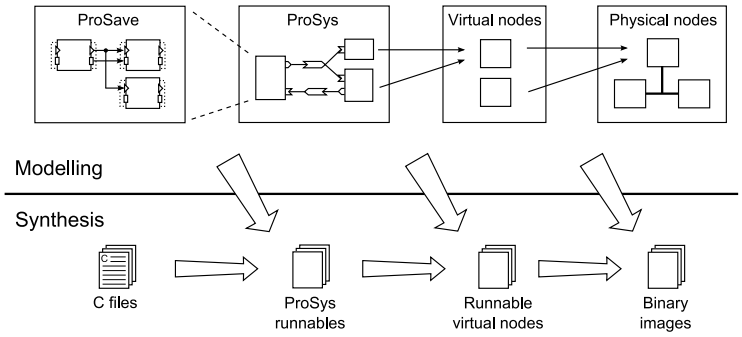

Figure 1: An overview of the deployment modelling formalisms and synthesis artefacts.

two represent the deployment modeling of the system. Functionality of the system is modeled by the ProSave and ProSys components at different levels of granularity. The basic functionality (data and control) of a simple component is captured in ProSave component level, which is passive in nature. At the second formalism level, many ProSave components are mapped to make a complete subsystem called ProSys that is active in nature. Both ProSys and ProSave allow composite components. For details on ProSave and ProSys, including the motivation for separating the two, see [9, 31].

The deployment modeling is used to capture the deployment related design decisions and then mapping the system to run on the physical platform. Many ProSys components can be mapped together on a virtual node (many-to-one mapping) together with a resource budget required by those components. After that many virtual nodes could be mapped on a physical node i.e. an ECU. The relationship is again many-to-one. This part represents all the physical nodes, their intercommunication through the network and the type of the network etc. Details about the deployment modeling are provided in [10].

3.1.2 The Runnable (or Executable) Realm

is the synthesis of runnables/executables from the ProCom model entities. The primitive ProSave components are represented as a simple C language source code in runnable form. From this C code, the ProSys runnables are generated which contain the collection of operating system tasks. Virtual node runnables implement the local scheduler and contain the tasks in a server. Hence a runnable virtual node actually encapsulates the set of tasks, resource allocations, and a real-time scheduler within a server in a two-level hierarchical scheduling framework. Final binary image is generated by connecting different virtual nodes together with a global scheduler and using the middleware to provide intra-communications among the virtual node executables.

3.1.3 Deployment and Synthesis Activities

Rather than deploying a whole system in one big step, the deployment of the ProCom components on the physical platform is done in the following two steps:

- First the ProSys subsystems are deployed on an intermediate node called virtual node. The allocation of ProSys subsystems to the virtual nodes is many-to-one

relationship. The additional information that is added at this step is the resource budgets (CPU time).

- The virtual nodes are then deployed on the physical nodes. The relationship is again many-to-one, which means that more than one virtual node can be deployed to one physical node.

This two-steps deployment process allows not only the detailed analysis in isolation from the other components to be deployed on the same physical node, but once checked for correctness, it also preserves its temporal properties for further reuse of this virtual node as an independent component. Section 4 describes this further.

The PRIDE tool supports the automatic synthesis of the components at different levels [8]. At the ProSave level, the XML descriptions of the components is the input and the C files are generated containing the basic functionality. At the second level, ProSys components are assigned to the tasks to generate ProSys runnables. Since the tasks at this level are independent of the execution platform, therefore, the only attribute assigned at this stage is the period for each task; which they get from the clock frequency that is triggering the specific component. A clock defines the periodic triggering of components with a specified frequency. Components are allocated to a task when (i) the components are triggered by the same event, (ii) when the components have precedence relation among them to be preserved. The synthesis of the runnable virtual nodes and final executables is given in Section 4.2.

3.2 Hierarchical Scheduling Framework

A two-level Hierarchical Scheduling Framework (HSF) [14] is used to provide the temporal isolation among a set of subsystems. In hierarchical scheduling, the CPU time is partitioned among many subsystems (or servers), that are scheduled by a global (system-level) scheduler. Each server contains its own internal set of tasks that are scheduled by a local (subsystem-level) scheduler. Hence a two-level HSF can be viewed as a tree with one parent node (global scheduler) and many leaf nodes (local schedulers) as illustrated in Figure 2. The parent node is a global scheduler that schedules subsystems. Each subsystem has its own local scheduler, that schedules the tasks within the subsystem.

The HSF gives the potential to develop and analyze subsystems in isolation from each other [25]. As each subsystem has its own local scheduler, after satisfying the temporal constraints, the temporal properties are saved within each subsystem. Later, a global scheduler is used to schedule all the subsystems together without violating the temporal constraints that are already analyzed and stored in the subsystems. Accordingly we can say that the HSF provides partitioning of the CPU between different servers. Thus, server-functionality can be isolated from each other for, e.g., fault containment, compositional verification, validation and certification, and unit testing.

Using HSF a subsystem (runnable virtual node in our case) can be developed and analyzed in isolation, with its own local scheduler at first step of deployment and its temporal properties are preserved. Then at the second step of deployment, multiple virtual nodes (subsystems) are integrated onto a physical node using an arbitrary global scheduler without violating the temporal properties of the individual subsystems analyzed in isolation.

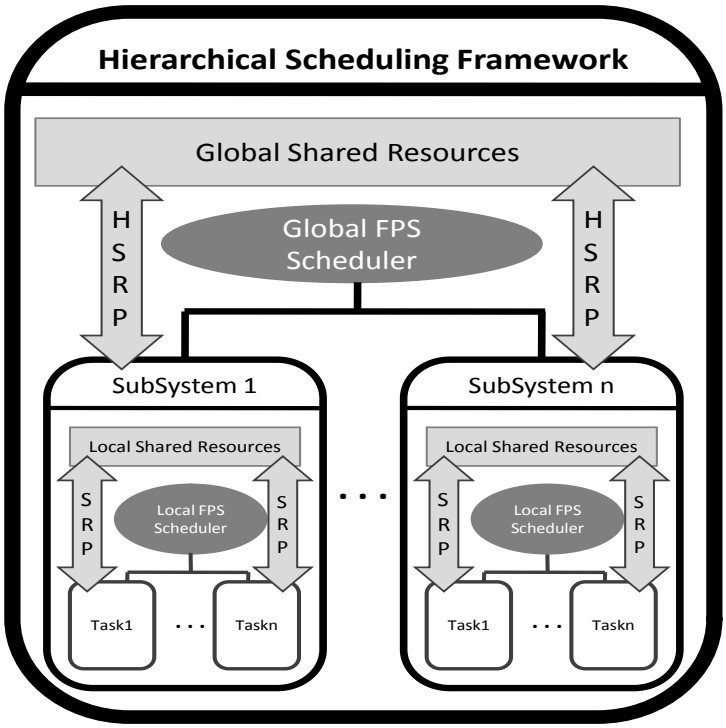

Figure 2: Two-level Hierarchical Scheduling Framework

3.3 FreeRTOS and its HSF Implementation

In this work we use FreeRTOS as the operating system to execute both levels of the HSF. FreeRTOS is a portable open source real-time kernel with properties like small and scalable, support for 23 different hardware architectures, and ease to extend and maintain.

The official release of FreeRTOS only supports a single level fixed-priority scheduling. However, a recent work has been presented that implements a two-level HSF for FreeRTOS [20] with associated primitives for hard real-time sharing of resources both within and between servers [21]. The HSF implementation supports two kinds of servers, idling periodic [33] and deferrable servers [35]. The implementation uses fixed priority preemptive scheduling (FPPS) for both global and local-level scheduling. For local resource-sharing (within a server) the Stack Resource Policy (SRP) [6] is used, and for global resource-sharing (between servers) the Hierarchical Stack Resource Policy (HSRP) protocol [12] is used with three different overrun mechanisms to deal with the server budget expiration within the critical section [7]. The HSF supports reservations by associating a tuple $< Q, P >$ to each server where P is the server period and Q ($0 < Q \leq P$) is the allocated portion of P.

The FreeRTOS has been adopted as one of the supported operating systems of ProCom. Given Q, P, and information on resource holding times, the schedulability of a server and/or a whole system can be calculated with the methods presented in [21].

4 Runnable Virtual Node

The concept of runnable virtual node is used to achieve not only temporal isolation and predictable temporal properties of real-time components but also to get better reusability of components with real-time properties. Further it reduces efforts related to system-level testing, validation and certification. This concept is based on a two-level deployment process. It means that the whole system is generated in two steps rather than a single big synthesis step. At the first level of deployment, the functional properties (functionality of components) are combined and preserved with their extra-functional properties (timing requirement) in the runnable virtual nodes. In this way it encapsulates the behavior with respect to timing and resource usage, and becomes a reusable component in addition to the design-time components. Followed by the second level of deployment where these runnable virtual nodes are implemented on the physical platform along with a global scheduler.

A runnable virtual node includes the executable representation of the components assigned to the tasks, a resource allocation (period and budget of server), and a real-time scheduler, to be executed within a server in the hierarchical scheduling framework.

4.1 Applying Virtual Node Concept to ProCom Component Model

In ProCom, a virtual node is an integrated model concept. It means that the virtual nodes exist both on the modeling level and on the synthesis level as shown in Figure 1. In the synthesis realm they are called runnable virtual nodes.

At the modeling level, each virtual node contains a set of integrated ProSys components plus the execution resources (a period and budget) required for these ProSys components. The priorities of virtual nodes cannot be assigned at the modelling level. The priorities of a component are relative to other components in the system. Since virtual nodes are developed independently and are meant to be reused in different systems, therefore, the priorities are assigned to virtual nodes later during the synthesis process at the execution level.

At the execution level, the runnable virtual node contains a set of executable tasks, resources required to run those tasks and a real-time local scheduler to schedule these tasks. Note that the runnable virtual node is generated as a result of first deployment step; it is platform independent and not executable as a stand-alone entity.

4.2 The Synthesis of the Final Executables

The final executables are generated by assigning priorities to the servers and tasks in the runnable virtual nodes, completing the task bodies with the user code, synthesizing communication among those nodes (if needed) and linking them together with the operating system. These executables then can be downloaded and executed on a physical node. As the real-time properties of runnable virtual nodes are preserved within the servers, therefore when integrated with other runnable virtual nodes on a physical node, the real-time properties of the whole integrated system will be guaranteed by the schedulability analysis of the whole system.

The communication among runnable virtual nodes is provided by a System server, which is automatically generated for inter-node communication (if needed), at this step. The main functionality of the server is to send and receive messages among the nodes. It contains two tasks to achieve this purpose: a sender task and a message-port updater task. Additionally there can be a hardware-driver tasks in it, if needed. The system

server has the highest priority of all the servers in the system, with a very small execution time and its only functionality is to copy the messages from the sender port of one component to the receiver port of another component.

An idle server is also generated within the two-level hierarchical scheduling to test the temporal isolation among the runnable virtual nodes. When there is no other server in the system to execute, then the idle server will run. It has the lowest priority of all the other servers, i.e. 0. It contains only an idle task to execute [20]. This is useful for maintaining and testing the temporal separation among servers and also useful in testing system behavior. This information is useful in detecting over-reservations of server budgets and can be used as feedback to resource management.

5 Case Study: Cruise controller and an adaptive cruise controller

The PROGRESS Integrated Development Environment (PRIDE) tool [28] supports development of systems using ProCom component models and it has been used for developing the examples of cruise controller (CC) and an adaptive cruise controller (ACC). The purpose of these fictitious examples is to evaluate and demonstrate the execution-time properties and reusability of the run-time components with real-time properties. First, the CC system is realized and exercised to test the temporal isolations among run-time components. Its basic functionality is to keep the vehicle at a constant speed. Then the ACC system extends this functionality by keeping a distance from the vehicle in front by autonomously adapting its speed to the speed of the preceding vehicle and by providing emergency brakes to avoid collisions. To evaluate the reusability of real-time components, the ACC system is realized by the reuse of some runnable virtual nodes from the CC system.

In the remainder of this section we describe the development of both applications using the ProCom component model. The ProSave, ProSys, and virtual node components are modelled for both examples. The presentation is followed by the synthesis of executable binaries using a hierarchical scheduling technique and the evaluation of final executables on the target platform.

5.1 System design

The CC system is designed from two ProSys components, Cruise Controller and Vehicle Controller, which are modelled and deployed on two different virtual nodes. For the ACC system, the Cruise Controller component is replaced with the Adaptive Cruise Controller component as shown in Figure 3. These virtual nodes communicate with each other through input and output message ports. The detailed design of these ProSys components is in turn shown in Figures 4, 5, and 6.

The Cruise Controller component contains three elements as shown in Figure 4: an HMI Input to set the mode to on or off, and detecting the speed or the manual braking signal respectively, a Control Unit to compare the current speed with the desired speed and to send the signals to throttle or brake output port accordingly, and an HMI Output to communicate the status to the driver via the display. The Vehicle Controller component contains seven elements as shown in Figure 5: two Calc Max Value components: to choose the maximum (of throttle and input message port) speeds and maximum (of brake pedal and input message port) brakes, and to provide these values to Engine Con-

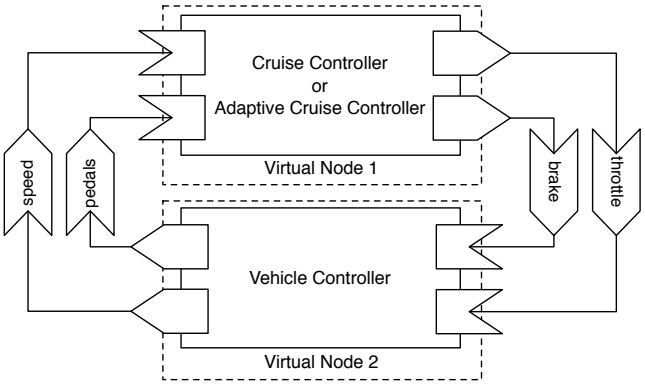

Figure 3: Deploying ProSys components on virtual nodes

troller and Brake Controller components respectively. The Speedometer writes current speed to the output port periodically.

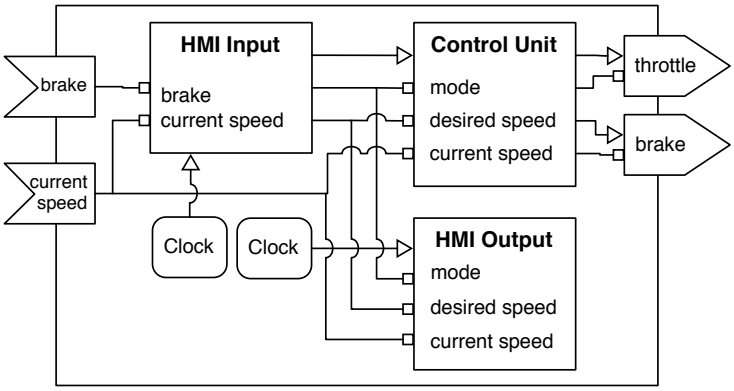

Figure 4: The Cruise Controller (CC) component

To extend the functionality to the ACC system, the CC virtual node is replaced by an ACC virtual node containing the same interface as that of CC component. The ACC system reuses the Vehicle Controller component from the CC application. The ACC component contains the following elements in addition to the Cruise Controller's elements: a Distance Sensor component to evaluate the distance to a vehicle/obstacle in front of the vehicle, a SpeedLimiter component to compute the vehicle's desired speed relative to the vehicle/object ahead as shown in Figure 6.

The ProSys components are then mapped to the virtual node components. Each virtual node is assigned a period and an execution budget to be executed in a local server within a two-level hierarchical scheduling framework. For the CC system, the Cruise Controller component is mapped to the `Virtual Node1` and the Vehicle Controller component is mapped to the `Virtual Node2`. In ACC system, the Adaptive Cruise Controller component is mapped to the `Virtual Node1` while `Virtual Node2` is reused from the CC system. The periods and budgets for these virtual nodes are assigned at the modelling level as shown in Figure 7.

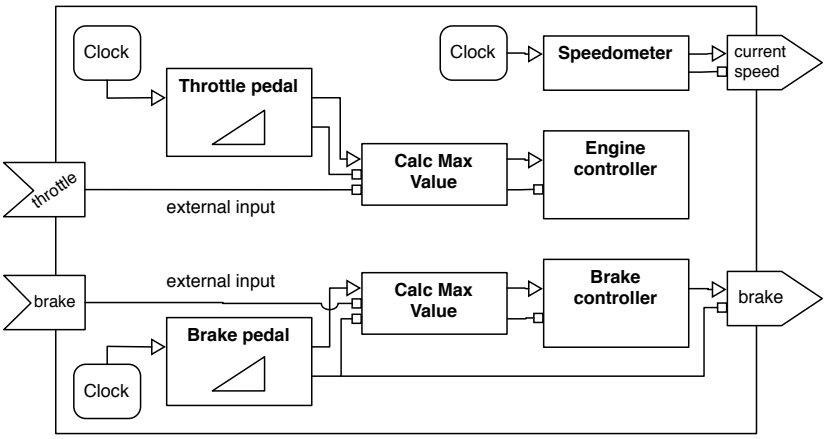

Figure 5: The Vehicle Controller (VC) component

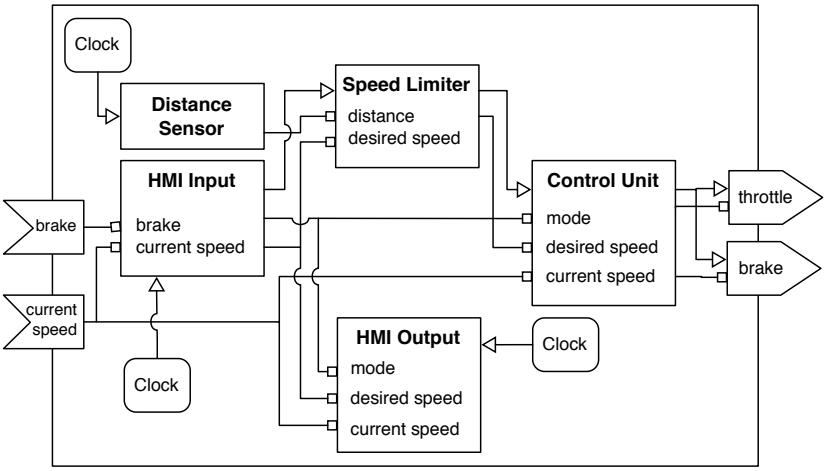

Figure 6: The Adaptive Cruise Controller (ACC) component

5.2 Synthesis

As described in Section 4.2, the PRIDE tool automatically synthesizes the code from the ProCom models at different stages. It takes the models as input, and generates all low-level platform independent code.

In the first step of the final synthesis/deployment process for the case study, two runnable virtual nodes are produced for both CC and ACC systems: one runnable virtual node for `Virtual Node1` and one for `Virtual Node2`. These generated nodes contain tasks definitions in them.

One task is synthesized for each clock. For CC example, two tasks are generated for the CC component: `CCT1` task including HMI Input and Control Unit; and `CCT2` task including HMI Output component. Three tasks are generated for the VC component: `VCT1` task including Throttle pedal, Calc Max Value, and Engine Controller; `VCT2` task including Brake pedal, Calc Max Value, and Brake Controller; and `VCT3` task including the Speedometer.

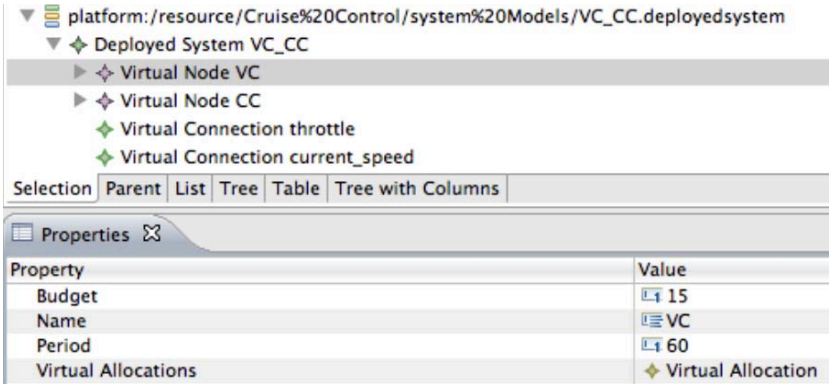

Figure 7: The timing properties of the virtual node

For ACC example, three tasks are generated for the ACC component: `ACCT1` task including Distance Sensor, `ACCT2` task including HMI Input, Speed Limiter, and Control Unit; and `ACCT3` task including HMI Output component.

Generating Final Binaries: In the second step of the final synthesis/deployment part, the priorities are assigned to the runnable virtual nodes (also called servers now) and to the tasks in them. Four servers are generated for both examples.

Server	CC	ACC	VC	SYSTEM
`Priority`	2	2	1	7
`Period`	40	40	60	20
`Budget`	10	10	15	4

Table 1: Servers used to test the CC and ACC systems behaviors.

A `System` server is generated to provide communication among the runnable virtual nodes. It has the highest priority of all the other servers, i.e. 7 (there are 8 different server priorities: from lowest priority 0 to the highest 7). Note that higher number means higher priority for both servers and tasks. The `System` server contains two tasks: a `Sender` and a `Receiver` task; whose functionality is to send and receive the data shared among virtual nodes respectively.

An `Idle` server is generated in the system with the lowest priority of all the other servers, i.e. 0, containing an idle task in it. All the other servers in the system have the priority higher than 0. This server is useful to check the temporal separation among servers.

The CC system contains two more servers in addition to `System` and `Idle` server: a `CC` server and a `VC` server associated with CC and VC virtual nodes respectively. The ACC system also contains four servers: an `ACC` server associated with the ACC virtual nodes. It reuses the `VC`, `System`, and `Idle` servers from the CC system. The priorities, periods and budgets for these servers are given in Table 1.

All the servers in both examples are *idling periodic* means that the tasks in the server execute and use the server's capacity until it is depleted. If server has the capacity but there is no task ready then it simply idles away its budget until a task becomes ready or the budget depletes. If a task arrives before the budget depletion, it will be served. An *idle task* per server is also generated that has the lowest priority and runs

when its server has budget remaining but none of its task are ready to execute. Task properties and their assignments to the servers are given in Tables 2 and 3.

Tasks	CCT1	CCT2	ACCT1	ACCT2	ACCT3	VCT1	VCT2	VCT3
Server	CC	CC	ACC	ACC	ACC	VC	VC	VC
Priority	2	1	2	2	1	1	1	2
Period	40	60	40	40	60	60	60	40

Table 2: Tasks in the two servers.

Tasks	Sender	Receiver
Server	$System$	$System$
Priority	2	2
Period	20	20

Table 3: Tasks in the System server.

Once all the platform dependent user code is finalized, all runnable virtual nodes that are to be deployed on the same physical node are integrated with a real-time time scheduler, the platform dependent final binaries are generated and downloaded on an ECU. Currently the PRIDE tool is evolving and the automatic synthesis part is not fully mature. Hence few parts of these experiments were synthesized manually, but it is not relevant for our experiments and does not effect our results.

5.3 Evaluation and Discussion

We have performed the experimental evaluation of the case study on an AVR-based 32-bit `EVK1100` board [15]. The `AVR32UC3A0512` micro-controller runs at the frequency of `12MHz` and its tick interrupt handler at `1ms`(milli seconds). The FreeRTOS operating system with its HSF implementation is used on the micro-controller using idling periodic servers and FPPS scheduling policy at both levels. Its tick-handler runs at the rate of `1ms`.

Our evaluation focuses mainly to evaluate the timing properties of the real-time components during their integration and the reuse of the components in different systems. We tested the real-time components for: (i) temporal isolation among the components that leads to (ii) the predictable integration and (iii) increased reusability of the components. The experiments are described below and the results are presented in the form of visualization of servers executions in Figures 8, 9 and 10.

5.3.1 Testing temporal isolation and predictable integration

To test the temporal isolation among runnable virtual nodes and their predictable integrations, the CC system is synthesized with the previously described four servers and task sets belonging to those servers. The final binaries are executed on the micro-controller and the traces of executions are visualized. The servers executions (according to their resource reservations) along with their task sets are presented in Figure 8 and Figure 9.

In these Figures, the horizontal axis represents the execution time starting from 0. In the task's visualization, the arrow represents task arrival and a gray rectangle means task execution. In the server's visualization, the numbers along the vertical axis are

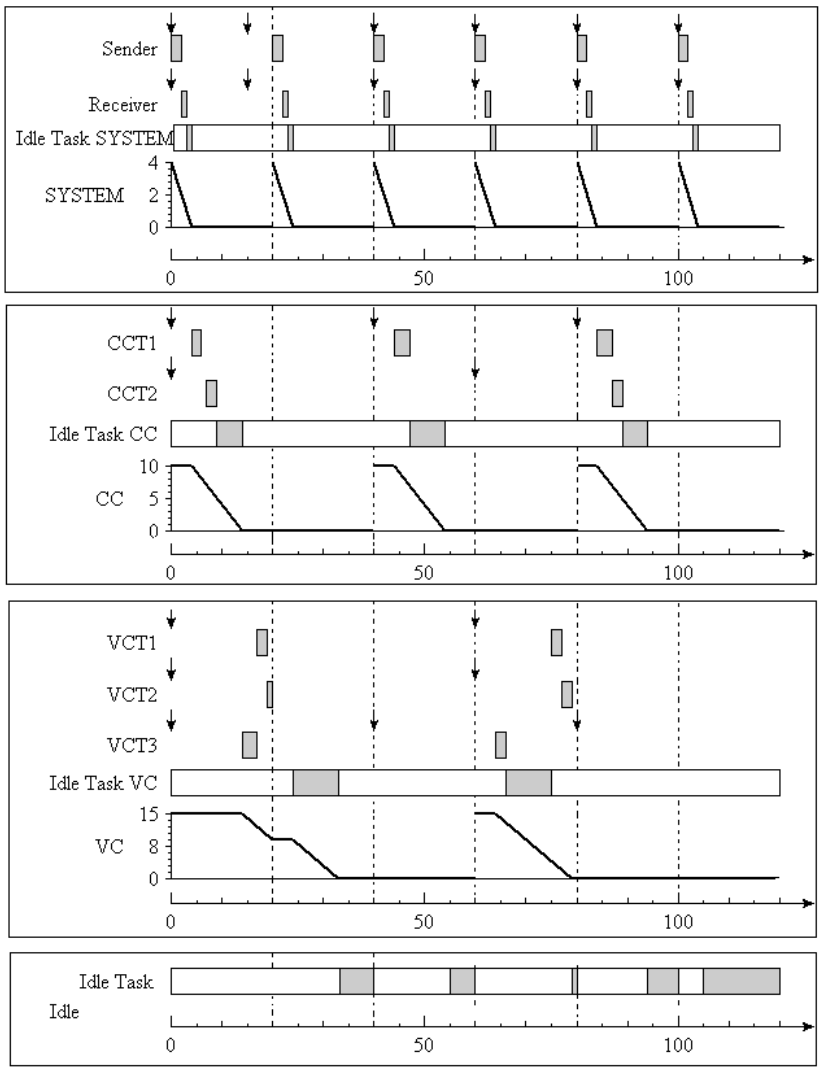

Figure 8: The trace for servers in the CC system during normal load

the server's capacity, the diagonal line represents the server execution while the horizontal line represents either the waiting time for the next activation (when budget has depleted) or the waiting for its turn to execute (when some other server is executing). Since these are idling periodic servers, all the servers in the system executes till budget depletion, if no task is ready then the idle task of that server executes till its budget depletion.

Figure 8 demonstrates the system execution under the normal load situation. The system's behavior is also tested during the overload situation to test the temporal isolation among the runnable virtual nodes. For example, if one server (runnable virtual node) is overloaded and its tasks miss deadlines, it should not affect the behavior of other servers in the system.

The same example is executed to perform this test but with the increased utilization

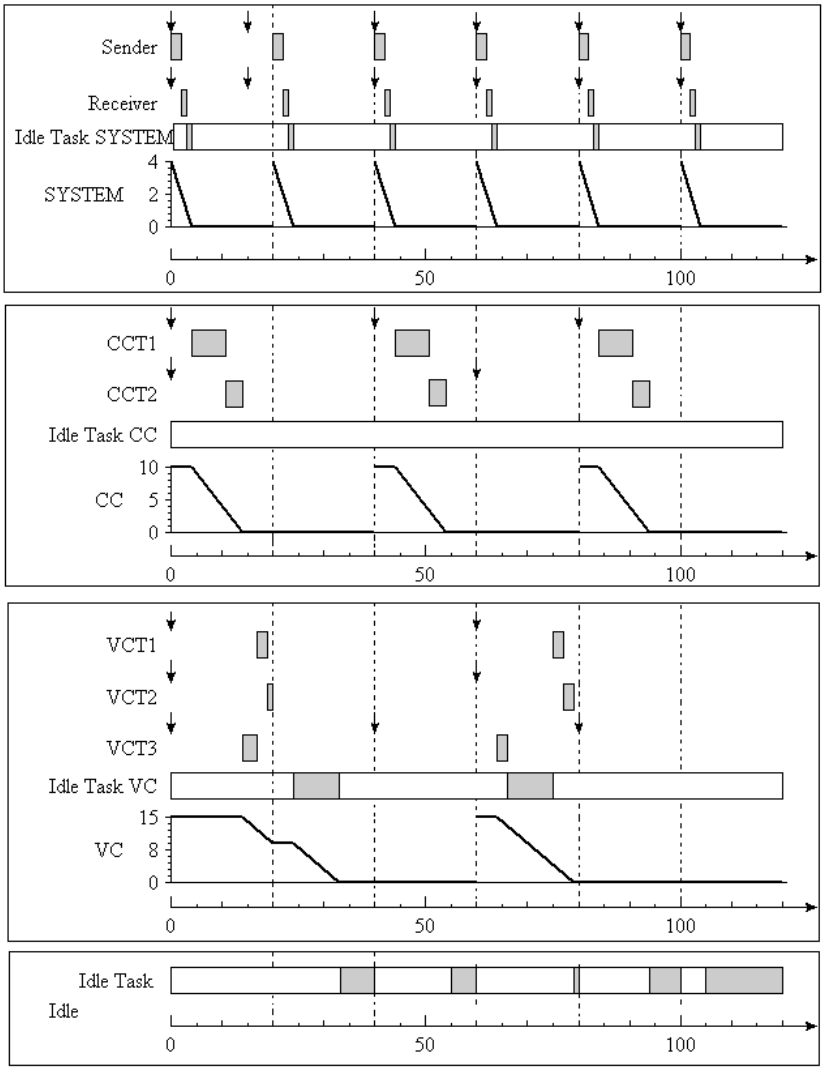

Figure 9: Trace showing temporal isolation during overload situation

of the CC server as shown in Figure 9. The execution times of tasks CCT1 and CCT2 are increased by adding the busy loops, hence making the CC server's utilization greater than 1. Therefore the low priority task CCT2 misses its deadlines at time 54. CCT2 is preempted at time 14 because of the CC server's budget expiration, and starts it's execution again when next time the server is replenished. Further, the CC is never idling because it is overloaded (Idle task of CC server is not executed in Figure 9).

The overload of CC server does not effect the behavior of any other server in the system as obvious from Figure 9. The VC server has a lower priority than the CC, but still it receives its allocated resources and its tasks meet their deadlines. In this manner, the runnable virtual nodes exhibit a predictable timing behaviour that eases their integration. It also manifests that the temporal errors are contained within the faulty runnable virtual node only and their effects are not propagated to the other nodes

in the system.

5.3.2 Testing component's reusability

The purpose of this experiment is to test the reusability of the runnable virtual nodes in a new system. The ACC system is synthesized for this purpose. It also contains four servers: the ACC server is synthesized with its task set while the other three servers are reused from the CC system. The trace of execution is visualized and presented in Figure 10.

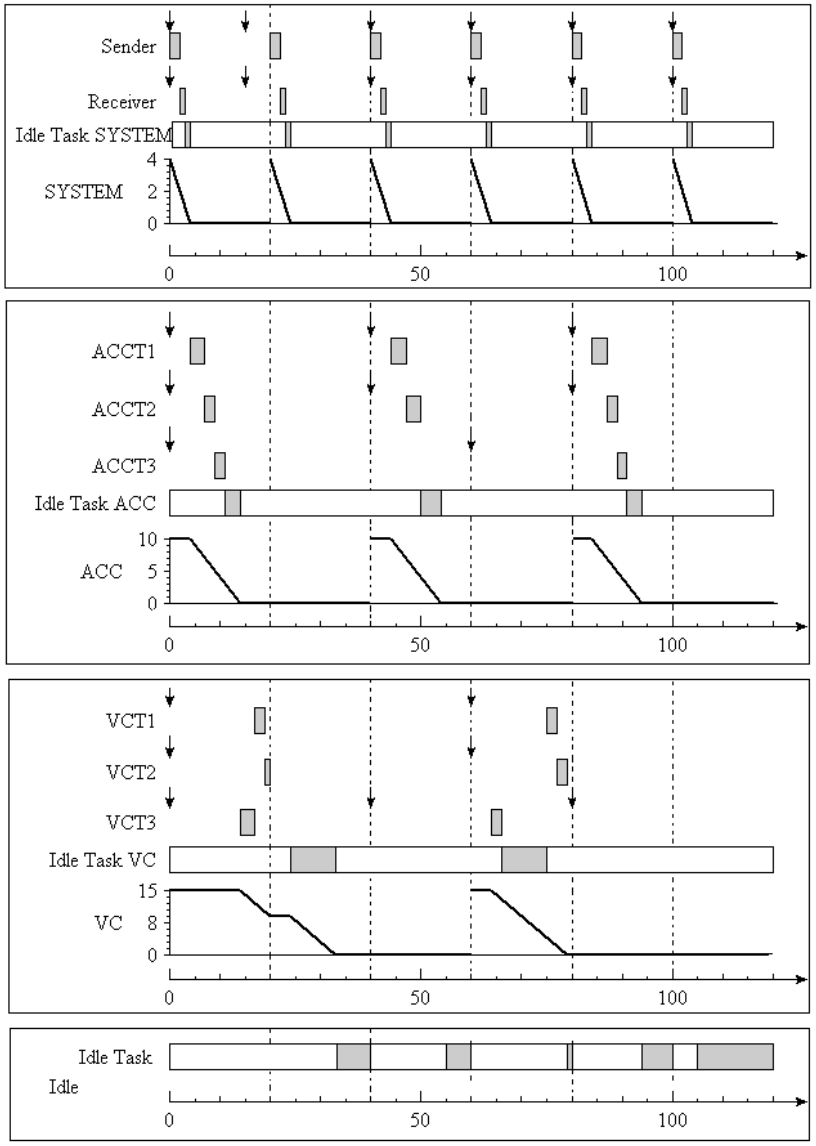

Figure 10: Trace showing reusability of runnable virtual nodes in ACC system

Since the runnable virtual nodes preserve their timing properties within them; there-

fore, their behaviour should not be changed when integrated into a new system, as long as their reserved resources are provided.

The task set for the ACC server is different from that of CC server. It is clear from the Figure 10 that all the three reused servers sustain their timing behaviour. For example, the VC server has a lower priority than ACC, still it's behaviour is not effected at all and remains similar to its behaviour in the CC system. It confirms the predictable integration of real-time components on one hand, and demonstrates their reusability on the other hand. We observed the same results on testing the ACC server with the changed timing properties, i.e. period 40 and budget 15. As long as the allocated budgets to servers (at the modeling level) are provided, the timing properties are guaranteed at the execution.

Hence, by the use of runnable virtual node components and two-level deployment process, the timing requirements are also encapsulated within the components along with their function requirements and the temporal partitioning is provided among the components (using HSF), that results in the increased predictability during component's integration and making the runnable virtual nodes a reusable entity.

6 Conclusions

We present the concept of runnable virtual nodes as a means to achieve predictable integration and reuse of software components using a two-level deployment process in cyber-physical systems. The virtual node is intended as a coarse-grained component for single node deployment and with potential internal multitasking. Each physical node is used to execute one or more virtual nodes. The idea is to encapsulate real-time properties into model-driven reusable components-based systems to achieve predictable integration and reusability of virtual nodes, thereby facilitating the development of complex CPS.

The notion of two-level deployment process encapsulates the timing properties and uses the hierarchical scheduling within runnable virtual nodes that provides temporal isolation and increases the reuse of the nodes in different systems. Hence using runnable virtual nodes, a complex CPS can be developed as a set of well defined reusable components encapsulating functional and timing properties.

A proof-of-concept case study is presented which demonstrates temporal error containment within a virtual node as well as reuse of a virtual node in new environment without altering its temporal behavior. Our work is based on the ProCom component-technology [31] running on FreeRTOS which has been extended with a hierarchical scheduling framework. The case study was executed on an ECU with an AVR based 32-bit micro-controller. However, we believe that our concept is applicable also to commercial component technologies like AADL, AUTOSAR, Rubus [19].

For future work, we plan to support virtual communication-busses using server-based scheduling techniques for e.g. CAN [24] and Ethernet [30]. This will allow development, integration and reuse of distributed components using a set of virtual nodes and buses.

References

[1] Past, Present, and Future of real-time embedded automotive software: a close look at basic concepts of AUTOSAR. In *SAE World Congress and Exhibiion,*

Session: In-Vehicle software, 2006.

[2] Arcticus Systems Web-Page. http://www.arcticus.se.

[3] Arcticus Systems. The Rubus Operating System. http://www.arcticus.se.

[4] AUTOSAR GbR. Specification of Operating System, 2008.

[5] AUTOSAR Partnership. Specification of RTE V2.0.1 R3.0 Rev 0001 , 2008. http://www.autosar.org/.

[6] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[7] Moris Behnam, Thomas Nolte, Mikael Sjödin, and Insik Shin. Overrun Methods and Resource Holding Times for Hierarchical Scheduling of Semi-Independent Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 6(1), February 2010.

[8] Etienne Borde and Jan Carlson. Towards verified synthesis of procom, a component model for real-time embedded systems. In *14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE)*. ACM, June 2011.

[9] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[10] Jan Carlson, Juraj Feljan, Jukka Mäki-Turja, and Mikael Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *Proceedings of the 36^{th} Euromicro Conference on Software Engineering and Advanced Applications (SEAA' 10)*, September 2010.

[11] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

[12] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority preemptive systems. In *IEEE Real-Time Systems Symposium (RTSS'06)*, pages 389–398, 2006.

[13] Julien Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon. Validate, simulate, and implement arinc653 systems using the aadl. *Ada Lett.*, 29(3):31–44, 2009.

[14] Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *IEEE Real-Time Systems Symposium(RTSS'97)*, pages 308–319, 1997.

[15] ATMEL EVK1100 product page. http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=4114.

[16] FreeRTOS web-site. http://www.freertos.org/.

[17] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus Component Model for Resource Constrained Real-Time Systems. In *3rd International Symposium on Industrial Embedded Systems*, 2008.

[18] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Trans. Embed. Comput. Syst.*, 7(4):1–25, 2008.

[19] Rafia Inam, Jukka Mäki-Turja, Jan Carlson, and Mikael Sjödin. Using temporal isolation to achieve predictable integration of real-time components. In *22nd Euromicro Conference on Real-Time Systems (ECRTS10) WiP Session*, pages 17–20, July 2010.

[20] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, S. M. H. Ashjaei, and Sara Afshar. Hierarchical Scheduling Framework Implementation in FreeRTOS. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*.

[21] Rafia Inam, Jukka Mäki-Turja, Mikael Sjödin, and Moris Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 51–60, Porto, Portugal, 2011.

[22] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. *OCARINA : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications*. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-01923-4.

[23] Thomas Nolte. Compositionality and CPS from a Platform Perspective. In *Proceedings of the 1st International Workshop on Cyber-Physical Systems, Networks, and Applications (CPSNA'11), satellite workshop of 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, August 2011.

[24] Thomas Nolte, Mikael Nolin, and Hans Hansson. Real-Time Server-Based Communication for CAN. *IEEE Transaction on Industrial Electronics*, 1(3):192–201, April 2005. Citations=33.

[25] Thomas Nolte, Insik Shin, Moris Behnam, and Mikael Sjödin. A Synchronization Protocol for Temporal Isolation of Software Components in Vehicular Systems. *IEEE Transactions on Industrial Informatics*, 5(4):375–387, November 2009.

[26] Object Management Group. Deployment and Configuration of Component-based Distributed Applications Specification, 2006. v4.0.

[27] Patricia Lopez Martinez and Cesar Cuevas and Jose M. Drake. RT-D&C: Deployment Specification of Real-time Component-based Applications. In *36th EUROMICRO Conference on Software Engineering an dAdvanced Applications (SEAA'10)*, pages 147–155, 2010.

[28] PRIDE Team. PRIDE: the PROGRESS Integrated Development Environment, 2010. "http://www.idt.mdh.se/pride/?id=documentation".

[29] SAE International. AADL specification. http://www.sae.org/technical/standards/AS5506/1.

[30] Rui Santos, Paulo Pedreiras, Moris Behnam, Thomas Nolte, and Luis Almeida. Hierarchical server-based traffic scheduling in ethernet switches. In *3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'10)*, pages 69–70, December 2010.

[31] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In *11th International Symposium on Component Based Software Engineering*, pages 310–317, October 2008.

[32] L. Sha, T. Abdelzaher, K-E. rzn, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2/3):101–155, 2004.

[33] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 181–191, 1986.

[34] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, pages 16–25, June 1995.

[35] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments. *IEEE Transactions on Computers*, 44(1), 1995.