A Component Model for Control-Intensive Distributed Embedded Systems [1]

Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš,
Jan Carlson, and Ivica Crnković

Mälardalen University, Västerås, Sweden.
{severine.sentilles, aneta.vulgarakis, tomas.bures,
jan.carlson, ivica.crnkovic}@mdh.se

Abstract. In this paper we focus on design of a class of distributed embedded systems that primarily perform real-time controlling tasks. We propose a two-layer component model for design and development of such embedded systems with the aim of using component-based development for decreasing the complexity in design and providing a ground for analyzing them and predict their properties, such as resource consumption and timing behavior. The two-layer model is used to efficiently cope with different design paradigms on different abstraction levels. The model is illustrated by an example from the vehicular domain.

1 Introduction

A special class of embedded systems are control-intensive distributed systems which can be found in many products, such as vehicles, automation systems, or distributed wireless networks. In this category of systems as in most embedded systems, resources limitations in terms of memory, bandwidth and energy combined with the existence of dependability and real-time concerns are obviously issues to take into consideration.

Another problem when developing such systems is to deal with the rapidly increasing complexity. For example in the automotive industry, the complexity of the electronic architecture is growing exponentially, directed by the demands on the driver's safety, assistance and comfort [3]. In this class of systems, distribution is also an important aspect. The architecture of the electronic systems is distributed all over the corresponding product (car, production cell, etc.), following its physical architecture, to bring the embedded system closer to the sensed or controlled elements.

In this paper, we propose a new component model called ProCom with the following main objectives: (i) to have an ability of handling the different needs which exist at different granularity levels (provide suitable semantics at different

[1] This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre Progress.

levels of the system design); (ii) to provide coverage of the whole development process; (iii) to provide support to facilitate analysis, verification, validation and testing; and (iv) to support the deployment of components and the generation of an optimized and schedulable image of the systems. The focus of this paper is on the component model itself, described as means for designing and modelling system functionality and as a framework that enables integration of different types of models for resource and timing analysis.

The component model is a part of the PROGRESS approach [7] that distinguishes three key activities in the development: design, analysis and deployment. The *design* activity provides the architectural description of the system compliant with the semantic rules of the component model presented in this paper and enables the integration analysis and deployment capabilities. *Analysis* is carried out to ensure that the developed embedded system meets its dependability requirements and constraints in terms of resource limitations. The proposed component model provides means to handle and reuse the different information generated during the analysis activity. The *deployment* activity is specific for control-intensive embedded systems; due to timing requirements and resource constraints, the execution models can be very different from the design models. Typically, execution units are processes and threads of tasks.

The main focus of this paper is oriented towards system design. The two supplementary activities (analysis and deployment) are outside the scope of the paper. A component model that enables a reusable design, takes into consideration the requirements' characteristics for control-intensive embedded systems, and is used as an integration frame for analysis and deployment, is elaborated in the subsequent sections.

The ideas underlying ProCom emanate partly from the previous work on the SaveComp Component Model (SaveCCM) [1] within the SAVE project, such as the emphasis on reusability, a possibility to analyse components for timing behavior and safety properties. Several other concepts and component models have inspired the ProCom Design. Some of them are the Rubus component model [2], Prediction-Enabled Component Technology (PECT) [10], AUTOSAR [3], Koala [9], the Robocop project [8], and BIP [4].

2 The ProCom two layer component model

In designing our component model, we have aimed at addressing the key concerns which exist in the development of control-intensive distributed embedded systems. We have analyzed these concerns in our previous work [6], with the conclusion that in order to cover the whole development process of the systems, i.e. both the design of a complete system and of the low-level control-based functionalities, two distinct levels of granularity are necessary.

Taking into consideration the difference between those levels, we propose a two-layer component model, called *ProCom*. It distinguishes a component model used for modelling independent distributed components with complex functionality (called *ProSys*) and a component model used for modelling small parts of

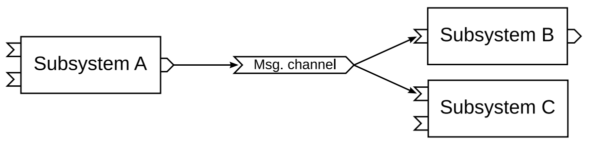


Fig. 1. Three subsystems communicating via a message channel.

control functionality (called *ProSave*). ProCom further establishes how a ProSys component may be modelled out of ProSave components. The following subsections describe both of the layers and their relation. The complete specification of ProCom is available in [5].

2.1 ProSys — the upper layer

In ProSys, a system is modeled as a collection of concurrent, communicating *subsystems*, possibly developed independently. Some of those subsystems, called *composite subsystems*, can in turn be built out of other subsystems, thus making ProSys a hierarchical component model. This hierarchy ends with the so-called *primitive subsystems*, which are either subsystems coming from the ProSave layer or non-decomposable units of implementation (such as COTS or legacy subsystems) with wrappers to enable compositions with other subsystems. From a CBSE perspective, subsystems are the “components” of the ProSys layer, i.e., design or implementation units that can be developed independently, stored in a repository and reused in multiple applications.

The communication between subsystems is based on the asynchronous message passing paradigm which allows transparent communication (both locally or distributed over a bus). A subsystem is specified by typed input and output *message ports*, expressing what type of messages the subsystem receives and sends. The specification also includes attributes and models related to functionality, reliability, timing and resource usage, to be used in analysis and verification throughout the development process. The list of models and attributes used is not fixed and can be extended.

Message ports are connected via *message channels* — explicit design entities representing a piece of information that is of interest to several subsystems — as exemplified in Fig. 1. The message channels make it possible to express that a particular piece of shared data will be required in the system, before any producer or receiver of this data has been defined. Also, information about shared data such as precision, format, etc. can be associated with the message channel instead of with the message port where it is produced or consumed. That way, it can remain in the design even if, for example, the producer is replaced by another subsystem.

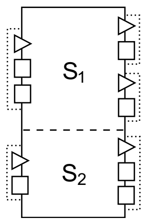


Fig. 2. A ProSave component with two services; S_1 has two output groups and S_2 has a single output group. Triangles and boxes denote trigger- and data ports, respectively.

2.2 ProSave — the lower layer

The ProSave layer serves for the design of single subsystems typically interacting with the system environment by reading sensor data and controlling actuators accordingly. On this level, components provide an abstraction of tasks and control loops found in control systems.

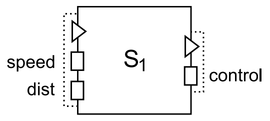
A subsystem is constructed by hierarchically structured and interconnected ProSave *components*. These components are encapsulated and reusable design-time units of functionality, with clearly defined interfaces to the environment. As they are designed mainly to model simple control loops and are usually not distributed, this component model is based on the pipes-and-filters architectural style with an explicit separation between data and control flow. The former is captured by *data ports* where data of a given type can be written or read, and the latter by *trigger ports* that control the activation of components.

A ProSave component is of a collection of services, each providing a particular functionality. A service consists of an *input port group* containing the activation trigger and the data required to perform the service, and a set of *output port groups* where the data produced by the service will be available. Fig. 2 illustrates these concepts. The data of an output group are produced at the same time, at which the trigger port of that group is also activated. Having multiple output groups allows the service to produce time critical parts of the output early.

ProSave components are *passive*, i.e. they do not contain their own execution threads and cannot initiate activities on their own. So each service remains in a passive state until its input trigger port has been activated. Once activated, the data input ports are read in one atomic operation and the service switches into an active state where it performs internal computations and produces data on its output ports. Before the service returns to the inactive state again, each of its output groups should be written exactly once.

Input data ports can receive data while the service is active, but it would only be available the next time the service is activated. This simplifies analysis by ensuring that once a service has been activated it is functionally (although not temporally) independent from other components executing concurrently.

A component also includes a collection of structured *attributes* which define simple or complex types of component properties such as behavioural models,



```
typedef struct {
  int *speed;
  float *dist;
} in_S1;
typedef struct {
  int *control;
} out_S1;
void init();
void entry_S1(in_S1 *in, out_S1 *out);
```

Fig. 3. A primitive component and the corresponding header file.

resource models, certain dependability measures, and documentation. These attributes can be explicitly associated with a specific port, group or service (e.g. the worst case execution time of a service, or the value range of a data port), or related to the component as a whole, for example a specification of the total memory footprint. New attribute types can also be added to the model.

The functionality of a component can either be realized by code (*primitive component*), or by interconnected sub-components (*composite component*). For primitive components, in addition to a function called at system startup to initialise the internal state, each service is implemented as a single non-suspending C function. Fig. 3 shows an example of the header file of a primitive component.

Composite components internally consist of *sub-components*, *connections* and *connectors*. A *connection* is a directed edge which connects two ports (output data port to input data port of compatible types and output trigger port to input trigger port) whereas *connectors* are constructs that provide detailed control over the data- and control-flow. The existence of different types of connectors and the simple structure of components makes it possible to explicitly specify and then analyse the control flow, timing properties and system performance.

The set of connectors in ProSave, selected to support typical collaboration patterns, is extensible and will grow over time as additional data- and control-flow constructs prove to be needed. The initial set includes connectors for *forking* and *joining* data or trigger connections, or *selecting* dynamically a path of the control flow depending on a condition. Fig. 4 shows a typical usage of the selection connector together with *or* connectors.

ProSave follows the push-model for data transfers and the triggered service always uses the latest value written to each input data port. Since communication may eventually be realised over a physical connection, the transfer of data and triggering is not an atomic operation. For triggering and data appearing together at an output group, however, the semantics specify that all data should be delivered to their destinations before the triggering is transferred, to avoid components being triggered before the data arrives.

2.3 Integration of layers — combining ProSave and ProSys

ProCom provides a mechanism for integrating the low-level design of a subsystem described by ProSave into the high-level design described by ProSys. A ProSys

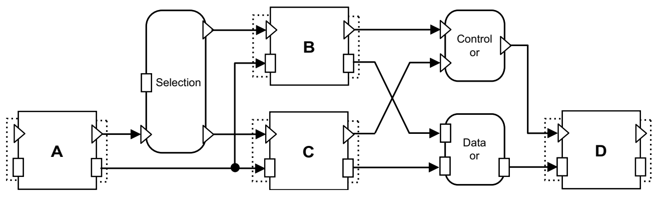


Fig. 4. A typical usage of *selection* and *or* connectors. When component A is finished, either B or C is executed, depending on the value at the selection data port. In either case, component D is executed afterwards, with the data produced by B or C as input.

primitive subsystem can be further specified using ProSave (as exemplified in Fig. 6). Concretely, in addition to ProSave components, connections and ProSave connectors, additional connector types are introduced to *(a)* map the architectural style (message passing used in ProSys to pipes-and-filters used in ProSave, and vice versa), and *(b)* specify periodic activation of ProSave components.

Periodic activation is provided by the clock connector, with a single output trigger port which is repeatedly activated at a given rate. To achieve the mapping from message passing to trigger and data, and vice versa, the message ports of the enclosing primitive subsystem are treated as connectors with one trigger port and one data port when appearing on the ProSave level. An input message port corresponds to a connector with output ports. Whenever a message is received by the message port, it writes the message data to the output data port and activates the output trigger. Oppositely, output message ports correspond to a connector with an input trigger and input data ports. When triggered, the current value of the data port is sent as a message.

These composition mechanisms do not only allow a consistent design of the entire system by integrated pre-existing subsystems but also provide mechanisms for analysis of particular attributes such as timing properties or performance of the entire system using specifications or analysis results of the subsystems.

3 Example

To illustrate the ProCom component model we use as an example an electronic stability control (ESC) system from the vehicular domain. In addition to anti-lock braking (ABS) and traction control (TCS), which aim at preventing the wheels from locking or spinning when braking or accelerating, respectively, the ESC also handles sliding caused by under- or oversteering.

The ESC can be modeled as a ProSys subsystem, as shown in Fig. 5. Inside, we find subsystems for the sensors and actuators that are local to the ESC. There are also subsystems corresponding to specific parts of the ESC functionality (SCS, TCS and ABS). In the envisioned scenario, the TCS and ABS subsystems are reused from previous versions of the car, while SCS corresponds to the added functionality for handling under- and oversteering. Finally, the

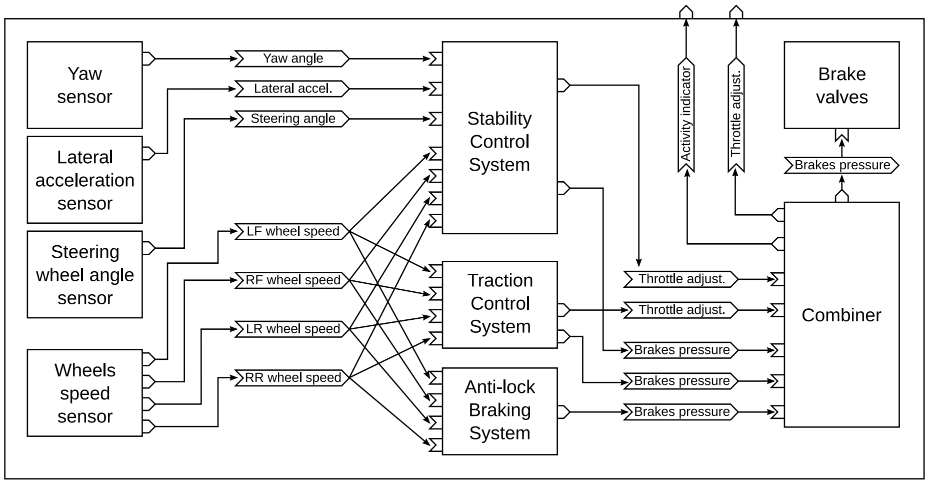


Fig. 5. The ESC is a composite subsystem, internally modelled in ProSys.

"Combiner" subsystem is responsible for combining the output of the three. The internal structure of a SCS primitive subsystem is modeled in ProSave (see Fig. 6). The SCS contains a single periodic activity performed at a frequency of 50 Hz, expressed by a clock connector. The clock first activates the two components responsible for computing the actual and desired direction, respectively. When both components have finished their respective tasks, the "Slide detection" component compares the results (i.e., the actual and desired directions) and decides whether or not stability control is required. The fourth component computes the actual response, i.e., the adjustment of brakeage and acceleration.

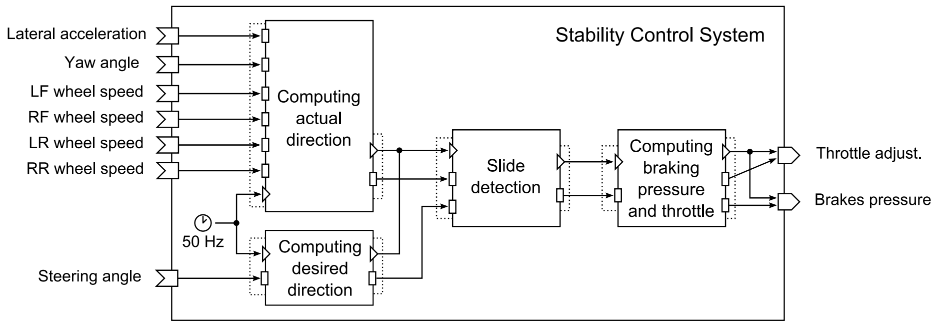


Fig. 6. The SCS subsystem, modelled in ProSave.

4 Conclusions

We have presented ProCom, a component model for control-intensive distributed embedded systems. The model takes into account the most important characteristics of these systems and consistently uses the concept of reusable components throughout the development process, from early design to deployment. A characteristic feature of the domain we consider is that the model of a system must be able to provide both a high-level view of loosely coupled subsystems and a low-level view of control loops controlling a particular piece of hardware. To address this, ProCom is structured in two layers (ProSys and ProSave). At the upper layer, ProSys, components correspond to complex active subsystems communicating via asynchronous message passing. The lower layer, ProSave, serves for modelling of primitive ProSys components. It is based on primitive components implemented by C functions, and explicitly captures the data transfer and control flow between components using a rich set of connectors.

The future work on ProCom includes elaborating on advanced features of the component model (e.g. static configuration, mode shifting, error-handling, etc.), building an integrated development environment and evaluating the proposed approach in real industrial case-studies.

References

1. Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.
2. Arcticus Systems. Rubus Software Components. `www.arcticus-systems.com`.
3. AUTOSAR Development Partnership. Technical Overview V2.2.1, February 2008. Available from `www.autosar.org`.
4. Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *Proc. of the 4th IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE, 2006.
5. 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.
6. Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. A component model family for vehicular embedded systems. In *The Third International Conference on Software Engineering Advances*. IEEE, October 2008.
7. Hans Hansson, Mikael Nolin, and Thomas Nolte. Beating the automotive code complexity challenge. In *National Workshop on High-Confidence Automotive Cyber-Physical Systems*, Troy, Michigan, USA, April 2008.
8. Robocop project page. `www.extra.research.philips.com/euprojects/robocop`.
9. Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.
10. Kurt C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon, 2003.