# Supporting Engineering Requirements in the Rubus Component Model

Kaj Hänninen*†, Jukka Mäki-Turja*, Mikael Nolin*‡,
Mats Lindberg†, John Lundbäck† and Kurt-Lennart Lundbäck†
*Mälardalen Real-Time Research Centre (MRTC), Västerås, Sweden
†Arcticus Systems, Järfälla, Sweden
‡CC Systems, Uppsala, Sweden

***Abstract*— In this paper we present a component model for development of distributed real-time systems. The model is developed to support development of embedded control systems for ground vehicles. The model aims at supporting three important activities in real-time development, (i) design, (ii) analysis and (iii) synthesis. These activities emphasise different and sometimes conflicting requirements that need to be balanced. For example, developers desire freedom in designing to solve complex tasks, analysis tools require the design to be formal enough for analysis and synthesis need to be efficient for low run-time footprint. We have considered industrial requirements for these activities and developed the RubusCMv3 component model. The model has been developed in close cooperation with industrial partners and it is currently being evaluated on real systems.**

## I. Introduction

The industrial requirements on embedded computer systems are constantly evolving. With the flexibility offered by software, the complexity of system designs and the amount of advanced computer controlled functionality in products is increasing. In the automotive domain, for example, systems are typically characterised by having a mix of requirements ranging from hard real-time, soft and even non real-time whilst operating in a resource constrained environment. Historically, developers of embedded real-time systems have used low level programming languages to guarantee full control of the system behaviour. Hence, many embedded real-time systems have become overly complex and hard to manage during functionality or technology shifts. The variety of functionality in today's embedded systems requires development methods and tools that support flexible and efficient development.

In resent years, Component Based Development (CBD), has shown to be successful in development of complex desktop applications. It is an emerging development discipline in which software systems are built by assembling pieces of software units, components, into larger systems. Components are self contained units that provide natural units of reuse by encapsulating functionality into manageable blocks of software. In general, components provide possibilities for high level architectural descriptions, hence, serving as descriptive entities and reusable logic.

Component based engineering has had a tremendous impact in the office-/Internet-area. Today, there exists several commercial component technologies for the desktop- and Internet-market, e.g., COM/DCOM [1], Corba [2] [3], Java Beans/EJB [4] [5], .NET [6] are readily available and used by developers on a day-to-day basis. However, these technologies are typically not suitable for embedded control systems [7]. In the embedded systems domain, CBSE is still only perceived as a promising future technology. Several component models and technologies for embedded systems have been proposed (e.g. Koala [8], PECOS [9] [10], MetaH [11], VEST [12], the control server [13], ReFlex [14] and [15] etc.). Projects such a Space4U [16], its predecessor Robocop [17], DECOS [18], SAVE [19] and PROGRESS [20] are targeting CBSE for embedded systems. Ever still, there is an apprehension that current tools and methods for embedded CBSE are lacking one or more key-properties to support industrial requirements such as:

- giving the software developer suitable level of expressiveness and/or abstraction
- enabling development of both safety critical and flexible functionality in a system
- enabling code and architecture reuse
- supporting development of predictable and resource efficient systems

To summarise, we recognise three important aspects of component based software development for resource constrained and predictable real-time systems: (a) the aspects of the developer, (b) the aspects of the analysis framework, and (c) the aspect of the run-time system. These three different aspects emphasise different, and sometimes conflicting, requirements for design, analysis and synthesis. For example, the developers must have sufficient methods and tools to design the overall software architecture. The analysis framework needs the architecture to be formal enough for automated analysis of important properties. And finally, the solution must be able to execute resource efficiently in the run-time platform. Any development strategy, for resource constrained and predictable real-time systems, has to take into account and balance these aspects to gain industrial usefulness.

In this paper we present a novel component model, RubusCMv3, for development of embedded control systems with a mix of hard, soft and non real-time requirements. The model is developed as a joint effort between industrial partners and the MultEx project [21] at Mälardalen Real-Time research Centre. The objective of the model is to support and balance

common requirements in design, analysis and synthesis of embedded real-time control systems.

**Paper outline.** The reminder of this paper is organised as follows. In section II we outline some common requirements in development of embedded software. In section III we describe the main objectives of the RubusCMv3 component model. In section IV the architectural elements of RubusCMv3 are presented. In section V we show a simple design of an oil pressure supervision to illustrate features of RubusCMv3. In section VI we discuss the key-properties supported by RubusCMv3. Finally, in section VII we conclude the work described in this paper.

## II. Engineering requirements on RubusCMv3

The fact that more and more mechanical solutions are replaced with software, results in an increasing system complexity. Today's embedded systems are typically characterised by having a mix of functionality with requirements ranging from hard real-time, soft and even non real-time. Many of these systems operate in resource constrained environments that need to satisfy requirements on dependability and efficient resource usage. For example, in a recent study [22], we discovered that vehicular systems are rapidly evolving and contain more heterogeneous functionality than before. Control applications, information handling and entertainment supplies, are nowadays supported in software by a mix of hard and soft real-time requirements. In addition, developers predict an increase in information intensity and a continuing increase in diversity of functionality. Safety and reliability are of utmost importance and considered key properties in development. This implies that development models must be able to support development and analysis of safety critical functionality, as well as development of flexible and resource efficient functionality.

## III. Objective of RubusCMv3

This section describes the main objectives of the Rubus component model. The objectives are derived from a recent study of industrial requirements in development of embedded systems in the vehicle domain [22]. One of the key objectives of the RubusCMv3 model is to support an overall descriptive view of the system functionality, i.e., serving as a system description facilitating reasoning about the functionality at a high level. Furthermore, abstraction mechanisms should be supported through hierarchical decomposition. This allows reasoning on different levels of abstraction. Besides having different levels of abstraction a user should be able to see the system through different views, highlighting different aspects. For example, a developer might be interested in the functional view when first designing the system, and later on, focus on the real-time temporal aspects, hiding unnecessary details of the functional aspects. Thus it must be possible to express both real-time requirements and real-time properties of the design. The overall purpose of the component model should be to express the infrastructure of software functions, i.e., the interaction between software functions in terms of data- and control-flow. One important principle is to separate code and infrastructure, i.e., explicit synchronisation or data access should all be visible in the infrastructure level. Separating code and infrastructure facilitates analysis and reuse of components in different contexts.

The component model should have a formal syntax and semantics and thus lend itself to formal analysis at an early stage. This allows timing errors to be revealed at an early stage in development. Real-time attributes on components can be expressed as budgets or estimates, enabling analysis of, for example, memory consumption and temporal attributes. These budgets can then serve as implementation requirements at a later stage in development.

The resulting architecture must be efficiently mapped to a run-system. With the diverging type of functionality in today's embedded systems, suitable execution models, such as hybrid scheduling [23], have to be supported. Hence, the user should be able to express events in terms of clocks and internal/external events.

## IV. The RubusCMv3 component model

The component model is developed as a part of the Rubus concept [24]. Rubus emanates from Basement [25], and was first introduced for industrial use in 1996. Throughout the years Rubus been used by a number of companies, e.g., [26]–[29] for development of real-time software.

The RubusCMv3 model is intentionally simple, still giving enough expressiveness for development and analysis of resource constrained systems with mixed real-time requirements. It is intended for development of software architectures expressing data-flow and synchronisation between software entities in single and multi-node systems.

In the following sections, we describe the architectural elements for hierarchical decomposition of software logic and assignment of real-time properties in RubusCMv3.

### A. Software logic

*Software circuits*

Software circuits (SWCs) are the basic unit of hierarchical decomposition in RubusCMv3. The primary purpose of an SWC is to encapsulate functions, hence a SWC can have multiple behaviours, each one represented by a specific entry function serving as the starting point of execution. Each SWC is defined by its behaviour, interface and an internal state data. Interfaces manage interaction between SWCs through ports. Two types of ports are supported, data ports for data flow and trigger ports for control flow. A SWC receives data(D)/triggering(T) on its input ports(I) and produces data/triggering to its output ports(O). Fig. 1 shows the graphical notation of an SWC in RubusCMv3. OTU denotes unconditional triggering, i.e., a trigger signal that is always produced, whereas UTC denotes conditional triggering meaning that the trigger signal on the UTC port may be produced, depending on conditions within the SWC. Data transfer/triggering between two SWCs require that output ports of the transmitting component are connected to input ports on the receiving component. If a data output

port and a trigger output port from the same component is connected to one other component, then the run-time environment must guarantee that the data will arrive to the destination before it is triggered by the source.

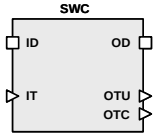


Fig. 1. Graphical description of a software circuit

A SWC becomes eligible for execution when its trigger condition is true, i.e., when the input trigger port receives a trigger signal. Each SWCs executes with a run to completion semantics as shown in Fig. 2, i.e., a SWC is not allowed to have synchronisation primitives defined within its behaviour (i.e., in the source code of the component), meaning that all synchronisations must visible in the design and represented by synchronisation objects. Before an SWC becomes executing it first reads the data on the data input ports. When the SWC terminates, it produces data/trigger on its output ports and reverts to its idle state.

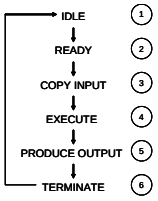


Fig. 2. Run-cycle of software circuit

A SWC may preserve its internal state data between executions. The internal state of an SWC must be initialised by a constructor in the SWC and cleaned up by a destructor in the SWC. The constructor must be called by the run-time environment at system start up, and the destructor must be called when the system is shutdown in an orderly fashion.

*Assemblies and composites*

Assemblies and composites (ASMs/CMPs) provide ways to connect a set of SWCs. They also provide means for hierarchical decomposition of SWCs and their connections. ASMs/CMPs provides no semantics, hence the purpose of ASMs/CMPs is to provide structure and increased abstraction, i.e., to abstract details of the software architecture. Assemblies and composites communicate through a set in- and output ports, similar to SWCs. Composites differ from assemblies in the sense that a composite object can be divided and parts of it can be deployed on different nodes, whereas assembly objects are un-dividable objects that cannot be split during deployment, i.e., an assembly object can only be deployed as a whole objects to a single node. Fig. 3 shows the graphical notation of an ASM in RubusCMv3

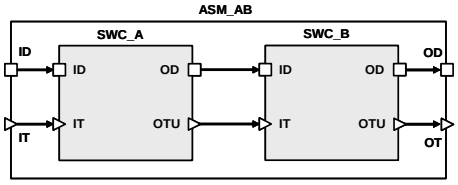


Fig. 3. Graphical description of an assembly

*Modes*

Modes are means to distinguish different states or conditions of a system. Each mode describes the functionality that is relevant for that mode. For example, a system may execute a certain type of functionality during start-up, and other type of functionality when in operational mode. Mode transitions are specified in order to show the transitions that are legal in the system. This is illustrated by a high level state diagram describing switches between different system functionality. In RubusCMv3, the mode objects are treated as self contained applications, realising the operational conditions of a system. Modes are semantically seen as synchronised only within a single node. Fig. 4 shows an example of modes and transitions within an ECU.

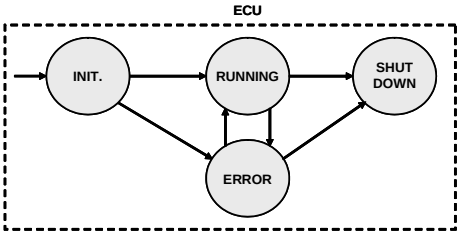


Fig. 4. Example of modes and mode transitions in an ECU

*System*

A system is the top level hierarchical entity, describing the software logic and architecture for a complete, possibly distributed, system. A system contains no assignments to platforms.

*Logic objects for data and triggering*

RubusCMv3 also defines the following software logic items for data and triggering (Fig. 5):

- *Source* items are used to, (i) define constant values on data input ports, (ii) indicate an unconnected trigger input port, i.e., a port that will not be triggered.
- *Sink* items are terminators of data- and trigger output ports. The sink object indicates that the data or trigger from the output port will be terminated, i.e., the control or data flow from the port terminates at the sink.
- *Named data* items can be described as collectors of data. The item has blackboard semantics, i.e., any of the architectural element (described above), can write and read from a named data. Moreover, entities that are external to the model may read and write named data.

- *Clock* items define periodic triggering. A clock object has an trigger output port activated with a specified frequency. In addition, a clock object may define a possible delay (offset time) specifying the minimum delay, from that the clock produces a trigger out, until the first logic object connected to the trigger output port on the clock, may be activated for execution.
- *Interrupt and event* items define external interrupts, internal and external events. These items are used to define events generated either by hardware or other software items. Events are specified by a minimum interarrival time (MINT) and priority. MINT specifies the shortest time between two consecutive activation of an event. The priority denotes the priority of the event (for events that correspond to interrupts, the priority denotes the interrupt priority).
- *Down sampling* items may be used to alter the frequency of triggering. For example, consider a control flow between SWCs (SWC_1 and SWC_2), then, without a down sampling object, a trigger signal from an output port of SWC_1 is immediately delivered to the input port of SWC_2. When attaching a down sampling item to the control flow, the delivery of the trigger signal from the output port of SWC_1 to the input port of SWC_2 may be altered in the following way: If DOWNSAMPLE-FACTOR is $x$ then the receiving port (input port on SWC_2 in this case) is triggered no more than every $x^{th}$ time.
- *AND/OR* items are logic objects for triggering. AND/OR objects have trigger input and trigger output ports. In essence, they define the boolean conditions that has to be true on the objects trigger input ports for the object to activate its trigger output port. AND/OR objects may be used to synchronise the control flow of a design.

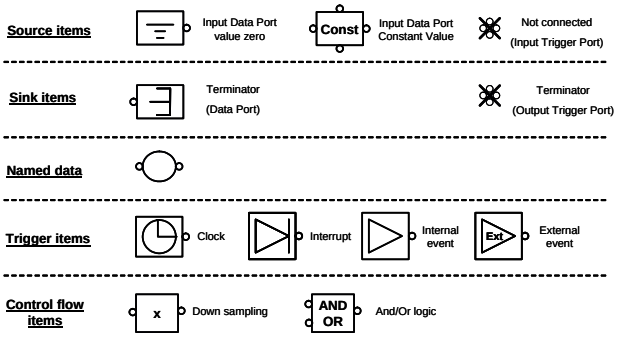


Fig. 5. Logic items for data and triggering

### B. Real-time properties in RubusCMv3

The software logic defined by the constituents of RubusCMv3 may be the associated with real-time execution properties defining actions that trigger executions, requirements on execution and timing properties of SWCs. To enable real-time analysis, each SWC is associated with a run-time profile describing the execution-time and memory consumption on different platforms. Two types of real-time requirements are currently supported in RubusCMv3 (i) deadlines on completion and (ii) jitter bounds. Deadlines are specified as bounds for a completed sequence and jitter bounds are specified as the maximum difference between the longest and shortest delay for a completed sequence.

## V. System example

In this section, we show a simple design of an oil pressure supervision (see Fig. 6) to illustrate some features of RubusCMv3. Assume that the requirements for the example are as follows:

- Correct oil pressure is important for the longevity of an combustion engine, hence the oil pressure of an engine should be regularly monitored. In case of abnormal oil pressure levels, the pressure level should be logged and an alarm, notifying the operator, must be raised within 5 seconds.

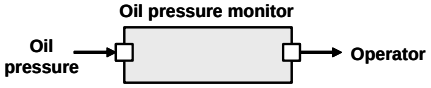


Fig. 6. Example, oil pressure supervision

A typical development scenario using RubusCMv3 includes identifying and composing the software architecture, deploying the architecture and assigning real-time properties to the design.

For this example, we assume that a pressure sensor is used to measure the oil pressure and a LED to inform the operator of abnormal pressure conditions, hence, the software parts in this example are illustrated by the constituents of the oil pressure monitor in Fig. 7. The monitor consist of an object for supervision of the oil pressure, an object logging abnormal conditions and an alarm object informing the operator of abnormal oil pressure.

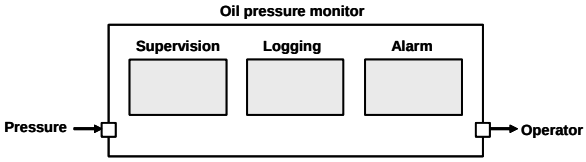


Fig. 7. Example, oil monitor software parts

We are now ready to create a more detailed design of the oil monitor object and applying the requirements on the software. Recall that in RubusCMv3, boxes represent data, triangles represent activation interfaces and the arrows show direction of data and control flow. A clock is assigned to the supervision object for periodic monitoring of the oil pressure. We assume that abnormal pressure conditions are sporadic, hence the logging and alarming of abnormal pressure is activated by events. The real time requirement, describing the maximum time delay (5 seconds) from the occurrence of abnormal oil

pressure until the alarm is raised, is assigned to the design (see Fig. 8).

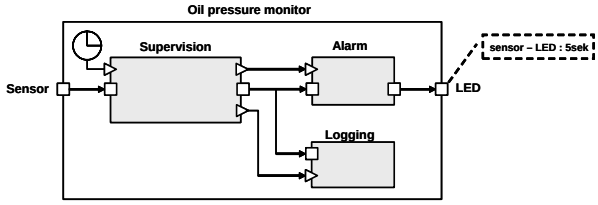


Fig. 8. Example, oil monitor software logic

Figure 8 shows that the data flow and the control flow are separated in the design. Separating data from control flow makes it easier to extract parts from the design and display only the data flow or only the control flow between object. We can now go further in to details and design the functionality for each of the objects in the oil pressure monitor. For simplicity, we only show an detailed example of the supervision object (Fig. 9).

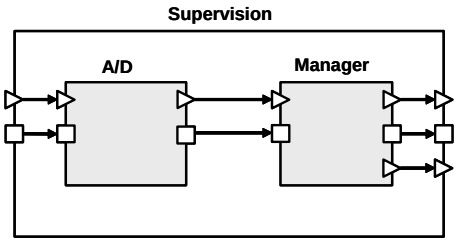


Fig. 9. Example, supervision software logic

A key feature of RubusCMv3 is to allow developers to focus on functionality on system level, without considering the hardware architecture. To this point, the functionality is designed without hardware architectural concerns. However, in reality, the hardware architecture of a system is often established based on the physical conditions and hence conceptually established at an early point in development. In addition, the placement of sensors and other elements for sensing environmental conditions are guiding the placement of the computational hardware. For the rest of this example we assume that the pressure sensor is connected to an engine ECU and that the LED display is connected to a cabin ECU. The engine ECU is physically placed in the engine compartment and the cabin ECU in the cabin. A CAN bus is used for communication between the ECUs (Fig. 10).

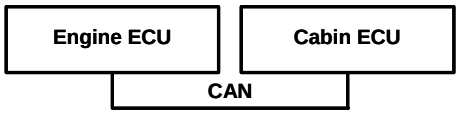


Fig. 10. Example, ECUs

Objects describing ECUs can have a set of operational modes, i.e., states, and transitions between modes within an ECU. Assuming that we need to deploy parts of our functionality to the drive mode of the engine ECU and other parts of our functionality to the drive mode of the cabin ECU, we then need to decide how to design the communication between the ECUs. In our example we assume that we need an object that packages messages into frames (CAN send) and an interrupt that activates the transmission of messages. At the receiving end we assign a CAN receive object that unpacks the messages and activates the software that alarms the operator of abnormal oil pressure conditions. Fig. 11 shows the software logic of each mode.

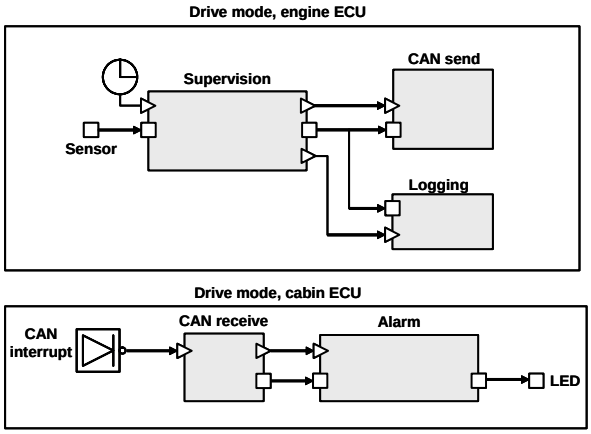


Fig. 11. Example, mode logic

To enable timing- and memory analysis of our example design, the objects need to be assigned properties such as worst case execution times and maximum stack usage etc. At this point, we may or may not have source code for the objects. We can however, assign timing and memory budgets to the objects and perform analysis. We can then choose to write or retrieve the source code to get more accurate figures on timing and memory consumption. For our example, we also need to consider timing in messages transmission on the CAN bus.

## VI. System development using RubusCMv3

The RubusCMv3 model provides constructs for systems design through hierarchical decomposition of software logic, giving developers the possibility design the logic at different levels of abstraction. By separating data- and control-flow, developers can exploit the benefits of addressing different views, e.g., the logical or the temporal view of a design. The oil pressure example shows the separation of concerns in different phases of development. First, the design of the complete functionality is put in focus, without hardware concerns. Secondly, the data flow and control flow are separated for abstraction reasons. The example also shows different levels of abstraction of a software entity, i.e., the monitor object consist of a supervision object that consists of an A/D converter and an oil manager.

The example also illustrates how a distributed functionality, with hard and non real-time requirements, can be realised by two execution models (EMs). The hard real time requirement,

realised by a time-triggered EM, concerns the maximum delay from abnormal pressure until an alarm is raised, whereas the logging of abnormal pressures, realised by the event-triggered EM, lack explicit timing requirements. Industrial systems often contain a mix of different real-time requirements. Still, many of these systems are developed using only the time triggered EM. The time triggered EM is, of course, suitable for periodic control systems and other type of functionality that need to be managed or polled periodically. However, the mixed requirements on systems indicate that the event-triggered execution model could also be used in systems development. For this reason, RubusCMv3 supports two fundamental execution models, the time- and event-triggered models. Hence, developers can choose different EMs for different subsystems, e.g., static scheduling for critical core functions (which is often desired, and sometimes even mandated by safety standards and certification agencies [30]) while allowing less critical functions to be executed using a less resource demanding and flexible EM.

The fact that EMs are logical objects defined in the infrastructure, instead of being coded into the components, will facilitated component reuse, since components can be developed independent of the EMs. This, in turn, will in a long term perspective decrease the software development costs. Also, for companies that sustain a product line,[1] software reuse is crucial and is an important factor in decreasing the time-to-market for new products.

To formally verify whether the real-time properties of a Rubus design is met, response time analysis such as [31] [32] [33] [34] may be used. Response-time analysis gives an upper bound on the completion time of functionality. It can also be used to analyse message transmission times in communication networks. By comparing the deadline with the response time, the temporal properties of the functionality can be verified. The jitter bound, as defined in RubusCMv3, can also be verified using response time analysis. The jitter bound is defined as the maximum difference between the longest and shortest delay for a completed sequence. This corresponds to the difference between the worst case response time and the best case response time of the functionality. In addition to formal analysis, verification of the logical functionality may be done by, for example, stimulating data and triggering. We have developed a framework for such verification of SWCs and ASMs/CMPs on PCs, see Fig. 12. The framework reads input data from, for example, Matlab. The data is then fed to the simulation process that controls the stimulation of input ports and state variables using probes. The output from the simulation process can be fed back to, for example, Matlab. This gives developers possibilities to test the logical functionality of software elements, even without having the actual hardware at hand. The framework has been successfully tested in development and it is currently used by our industrial partners.

[1]A product line is a series of related, but yet distinct, products. Economical benefits are achieved by synergies in the development and maintenance of the products in the product line.

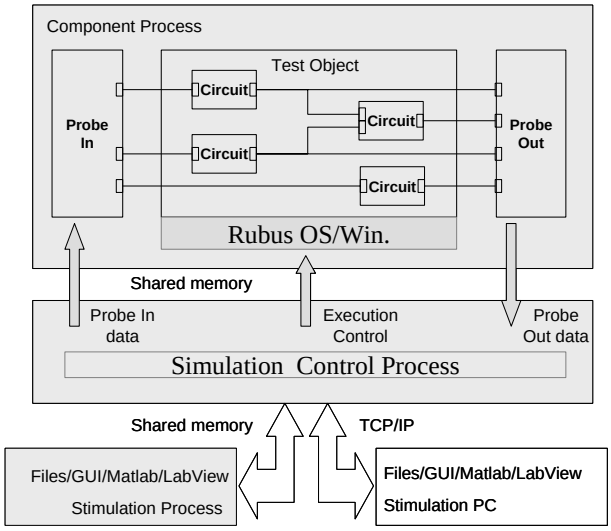


Fig. 12. Framework for testing of logical functionality of Rubus software elements

For resource constrained embedded systems, the resulting design must be resource efficiently mapped to a run time system. The mapping of logical software items to executable threads can of course be done in several ways. However, the RubusCMv3 model provides possibilities to map several components into one executable thread, minimising contexts switch overheads. This is possible since the model provides possibilities to realise functionality as transactions, i.e., a sequential execution of components triggered by a common source such as a clock item. In addition, the run-to-completion semantics of SWCs and the fact that synchronisation is done in the infrastructure, enables efficient use of memory by stack sharing. Stack sharing allows several tasks to share, i.e., execute, on a single stack, even in preemptive systems. It has been shown that significant memory savings can be achieved by stack sharing. Today, there exists a number of methods to formally analyse the amount of stack needed in shared stack systems, e.g., [35] [36] [37].

A run-time environment is essential to provide the infrastructure services that are needed to execute the logic defined by a component model. Furthermore, a run-time environment must also be able to preserve the semantics and support efficient execution of the logics defined by a component model. In general, RubusCMv3 do not restrict the use of a specific run-time environment, in fact, any run-time environment that endorse the semantics of RubusCMv3 can be used. However, to fully support RubusCMv3 model the run-time environment must be able to realise the complete set of logic with the semantics defined by the model. Currently, we have extended the Rubus-RTOS [24] to fully support the RubusCMv3 model.

## VII. Conclusions and Future Work

Today's embedded systems are typically characterised by having a mix of functionality with requirements ranging from hard real-time, soft and even non real-time. Many of these systems operate in resource constrained environments that

need to satisfy requirements on dependability as well as efficient resource usage. Current trends predict a continuing increase in diversity of functionality in systems, resulting in an increasing system complexity. Component Based Development (CBD) is a promising and emerging development discipline for real-time systems, providing many attractive qualities such as encapsulation, high level description and reusable logic.

This paper presented the RubusCMv3 component model, a novel component model for development of resource constrained embedded real-time systems. The model has been developed in close cooperation with industrial partners. It aims at supporting three important activities in real-time development, (i) design, (ii) analysis and (iii) synthesis. These activities emphasise different and sometimes conflicting requirements that need to be balanced. The model provides methods to express the infrastructure of software functions, i.e., the interaction between software functions in terms of data- and control-flow. The resulting architecture is formal enough for analysis of timing and memory properties. The components and the infrastructure allows for a resource efficient mapping onto a run-time structure.

The model is integrated in the next generation of the Rubus tool suite (RubusICE), the model is currently evaluated on real systems. We have successfully converted a traditionally developed industrial system, into a component based system using RubusCMv3. Future work consists of evaluating the component model in other industrial settings.

## REFERENCES


[1] Microsoft, "Microsoft COM Technologies," http://www.microsoft.com/-com/.

[2] OMG, "CORBA Home Page," http://www.omg.org/corba/.

[3] ——, "CORBA Component Model 3.0," June 2002, http://www.omg.-org/technology/documents/formal/components.htm.

[4] SUN Microsystems, "Enterprise Javabeans Technology," http://java.sun.-com/products/ejb/.

[5] ——, "Introducing Java Beans," http://developer.java.sun.com/-developer/onlineTraining/Beans/Beans1/index.html.

[6] Microsoft, ".NET Home Page," http://www.microsoft.com/net/.

[7] A. Möller, M. Åkerholm, J. Fredriksson, and M. Nolin, "Evaluation of Component Technologies with Respect to Industrial Requirements," in *Euromicro Conference, Component-Based Software Engineering Track*, August 2004.

[8] R. van Ommering, *Building Reliable Component-Based Software Systems*. Artech House Publishers, July 2002, ch. The Koala Component Model, pp. 223–236, iSBN 1-58053-327-2.

[9] P. O. Müller, C. M. Stich, and C. Zeidler, *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002, ch. Component Based Embedded Systems, pp. 303–323, iSBN 1-58053-327-2.

[10] "PECOS Project Web Site," http://www.pecos-project.org.

[11] S. Vestal, "Support for Real-Time Multi-Processor Avionics," in *Proc. $18^{th}$ IEEE Real-Time Systems Symposium (RTSS)*, December 1997, pp. 11–21.

[12] J. A. Stankovic, "VEST — A toolset for constructing and analyzing component based embedded systems," *Lecture Notes in Computer Science*, vol. 2211, pp. 390–402, 2001. [Online]. Available: citeseer.nj.nec.com/stankovic00vest.html

[13] A. Cervin and J. Eker, "The Control Server Model: A Computational Model for Real-Time Control Tasks," in *Proc. of the $15^{th}$ Euromicro Conference on Real-Time Systems*, July 2003.

[14] A. Wall, "Architectural Modeling and Analysis of Complex Real-Time Systems," Ph.D. dissertation, Mälardalen University, Dept. of Computer Science and Engineering, September 2003.

[15] M. Díaz, D. Garrido, L. Llopis, F. Rus, and J. Troya, "Integrating Real-Time Analysis in a Component Model for Embedded Systems," in *Proceedings of the $30^{th}$ Euromicro Conference*. Rennes, France: IEEE Computer Society, September 2004, pp. 14–21.

[16] "Space4U project home-page," http://www.extra.research.philips.com/-euprojects/space4u.

[17] "Robocop project home-page," http://www.extra.research.philips.com/-euprojects/robocop/index.htm.

[18] "DECOS - Dependable Embedded Components and Systems," http://-www.decos.at.

[19] SAVE, "SAVE Project Page," http://www.artes.uu.se/++/SAVE/.

[20] "PROGRESS Web-Page," http://www.mrtc.mdh.se/progress/.

[21] "MultEx - Software development using multiple execution models, Web-Page," http://www.mrtc.mdh.se/projects/multex/.

[22] K. Hänninen, J. Mäki-Turja, and M. Nolin, "Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain," in *Proceedings of the $13^{th}$ International Conference and Workshop on the Engineering of Computer Based Systems*. Potsdam, Germany: IEEE Computer Society, March 2006, pp. 139–147.

[23] J. Mäki-Turja, K. Hänninen, and M. Nolin, "Efficient Development of Real-Time Systems Using Hybrid Scheduling," in *International conference on Embedded Systems and Applications (ESA)*, June 2005.

[24] "Arcticus Systems Web-Page," http://www.arcticus-systems.com.

[25] H. Hansson, H. Lawson, and M. Strömberg, "BASEMENT a Distributed Real-Time Architecture for Vehicle Applications," *Journal of Real-Time Systems*, vol. 3, no. 11, pp. 223–244, November 1996.

[26] "BAE Systems," Web page, http://www.baesystems.com.

[27] "Knorr-bremse," Web page, http://www.knorr-bremse.com.

[28] "Mecel," Web page, http://www.mecel.se/.

[29] "Volvo Construction Equipment," http://www.volvoce.com.

[30] I. E. Commission, "IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems."

[31] O. Redell, "Response time analysis for implementation of distributed control systems," Ph.D. dissertation, KTH, Department of Machine Design, 2003, series: TRITA-MMK 2003:17.

[32] J. Mäki-Turja, "Engineering Strength Response-Time Analysis, A Timing Analysis Approach for the Development of Real-Time Systems," Ph.D. dissertation, Mälardalen University, Department of Computer Science and Electronics, 2005.

[33] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[34] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei, "Timing Analysis of the FlexRay Communication Protocol," in *Proc. of the $18^{th}$ Euromicro Conference on Real-Time Systems*, July 2006.

[35] K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlson, and M. Nolin, "Determining Maximum Stack Usage in Preemptive Shared Stack Systems," in *Proc. $27^{th}$ IEEE Real-Time Systems Symposium (RTSS)*, December 2006, pp. 445–453.

[36] M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Nolin, "Safe Shared Stack Bounds in Systems with Offsets and Precedences," Mälardalen Real-Time Research Centre (MRTC), Tech. Rep. MRTC no. 221, January 2008.

[37] R. Davis, N. Merriam, and N. Tracey, "How embedded applications using an RTOS can stay within on-chip memory limits," in *Proc. of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.