An event algebra extension of the triggering mechanism in a component model for embedded systems

Jan Carlson [1,2] and Mikael Åkerholm [1]

Department of Computer Science and Electronics
Mälardalen University, Sweden

Abstract

In this article we present how the component triggering in SaveCCM, a component model intended for embedded vehicular systems, can be extended by means of an event algebra. The extension allows components to be triggered by complex event patterns, and not only by clock signals or single external events.

Separating the detection of triggering conditions from the definition of the triggered services permits more general components and thus improves component reusability. Providing event detection mechanisms within the component model means that triggering conditions are explicitly available for system analysis at design time.

An event algebra is used to define the complex triggering conditions. This algebra has a relatively simple declarative semantics and well documented algebraic properties, which facilitates formal and informal reasoning about the system. The algebra also ensures that detection of triggering conditions can be efficiently implemented with limited resources, which is critical in embedded applications.

Key words: Component-based software architecture, event detection, embedded systems.

1 Introduction

SaveCCM [1] is a component model intended for development of software for vehicular systems. The model is restrictive compared to most general component models, due to the high demands of predictability and run-time efficiency in the intended domain. We have extended the triggering mechanism of SaveCCM with an event algebra, which allows components to be triggered

[1] Email: {jan.carlson,mikael.akerholm}@mdh.se

[2] Funded by CUGS (the national graduate school in computer science).

by complex event patterns, rather than just a clock signal or a single external event.

As a running example, we consider a system with external events including a button B, and internal warning events P and T generated by components monitoring pressure and temperature, respectively. The system is supposed to perform a service provided by a third component whenever the button is pressed twice within two seconds, unless either of the alarm components signal in between.

One way to achieve this is to design a new component that is responsible for detecting this particular situation, and to trigger the component under the right circumstances. This means that the triggering condition is not visible on a system design level, and thus not easily available for analysis.

The alternative outlined in this paper is to provide detection of complex event patterns as a part of the component model. Complex triggering conditions are specified on a high level, with well-defined formal semantics, which supports formal analysis at design-time when the access to component source code is limited. At compile-time, code that detects the specified situations is automatically generated.

Triggering conditions are specified by expressions from an event algebra. For example, the situation described above could be defined by the expression $(\mathsf{B};\mathsf{B})_2 – (\mathsf{P} \vee \mathsf{T})$. The algebra is designed to be as intuitive as possible, under the restriction that it should be effectively implementable with limited resources, since we primarily target embedded applications. The operators have intuitive, declarative semantics, and we present a number of algebraic laws that facilitate formal and informal reasoning.

The rest of the paper is organised as follows: Section 2 surveys related work and Section 3 gives an overview of SaveCCM. A description of the proposed extension, including an informal introduction to the event algebra, is given in Section 4. In Section 5 the event algebra is presented more formally, and we outline how the properties of this particular algebra impact on the extended component model. Finally, Section 6 concludes the paper.

2 Related Work

Recently component technologies for different classes of embedded systems have been developed both in academia and industry. In relation to our proposal some of them support different triggering types, specification of advanced real-time constrains, and run-time flexibility. In this section we will briefly describe a sample from the automotive, consumer electronics, and automation domains.

The Rubus Component technology [10] is a commercial technology that is used in the automotive industry. It is shipped, and tightly integrated, with the Rubus operating system. Rubus components are statically scheduled, and sophisticated timing requirements can be specified, e.g., release-time, deadline,

worst case execution time and period time. The main limitation is that only periodic activation of components is possible.

Koala [14] is a component technology developed and used internally by Philips. Component binding flexibility can be achieved with switches, as in SaveCCM. Switches choose between interfaces offered by different components at run time, with possible static reduction at compile-time. ROBOCOP [5] is a continuation to enhance the Koala model with, e.g., support for real-time constrains and analysis.

Port Based Objects (PBO) [13] is a component technology specialised on reconfigurable robotics applications, from the Advanced Manipulators Laboratory at Carnegie Mellon University. The technology has support for modelling output response from given inputs of closed or open loop systems by applying transfer functions. Real-Time analysis is also supported.

PECOS [15] is a collaborative project between ABB and academia, with the aim to develop a technology adjusted for field-devices. Pecos support different trigger-types associated with components; they can be of passive, active or event-type. Passive components do not have their own execution thread, and have to be triggered by other types of components. Active components have their own thread that is periodically triggered. Event components are components that are triggered by an external event and have a thread of control.

Outside the domain of component-based architecture, event detection mechanisms of various kinds are used in a wide range of areas. For example, some large distributed systems have an architecture based on event subscribers and publishers. In such a system, rather than having subscribers register their interest in simple event types, and perform their own filtering and pattern detection, this functionality can be provided by the publisher. The subscribers register event patterns, specified for example in an event algebra. The publisher performs the event detection and notifies the individual subscribers when their pattern is detected. Many systems of this type has been proposed, e.g., the READY event notification service by Gruber et al. that contains a simple event algebra for registering event patterns [9].

In middleware platforms, event detection techniques are used to handle high volumes of event occurrences by allowing consumers to subscribe to certain event patterns rather than to single event types. For example, Sánchez et al. present an event correlation language where event expressions are translated into nested Petri net like automata [12].

The operators of the event algebra we use, as well as the interval-based semantics and the concept of restricted detection, are influenced by work in the area of active databases. Snoop [4], Ode [8] and SAMOS [7] are examples of active database systems where an event algebra is used to specify the reactive behaviour. These systems differ primarily in the choice of detection mechanism. SAMOS is based on Petri nets, while Snoop uses event graphs. In Ode, event definitions are equivalent to regular expressions and can be detected by

state automata.

Galton and Augusto have shown that associating occurrences of complex event patterns with a single time instant results in unintended semantics for some operation compositions [6]. They also present the core of an alternative, interval-based, semantics to handle the problem. We use a similar semantic base for our algebra, but extend it with a restriction policy to allow the algebra to be implemented with limited resources while retaining the desired algebraic properties. To the best of our knowledge, no existing event algebra provides assistance to the developer in terms of algebraic properties or an event expression equivalence theory, while at the same time ensuring that detection can be correctly performed with limited resources.

3 The SaveComp component model

SaveCCM is based on a textual syntax, but a somewhat modified subset of the component diagrams of UML2 is used as a graphical notation. In this paper, we present only the graphical notation, and refer the reader to [1] for details on the textual syntax.

3.1 Architectural elements

Systems are built from interconnected *components*, i.e., units of encapsulated behaviour with well defined interfaces defined in terms of input- and output *ports*, which are points of interaction between the component and its external environment. The model distinguishes between two aspects of ports: the data flow and the control flow. The former is captured by *data ports*, i.e., one element buffers where data of a given type can be written and read. Control flow is defined in terms of *triggering ports* that control the activation of components. Finally, a port can have both triggering and data functionality. The notation for these port types is shown in Fig. 1.

Basic components are associated with an executable through an entry function. Optionally, quality attributes can be given to specify particular properties of the component, e.g., worst case execution time. In addition to basic components, the model contains two more component types: *Assemblies* are encapsulated subsystems. The internal interconnections and components are hidden from the rest of the system, and can be accessed only through the ports of the assembly. *Switches* are lightweight components used to dynamically change the component interconnection structure. The switch specifies a number of connection patterns, i.e., partial mappings from input to output ports. Each connection pattern is guarded by a logical expression, possibly over the data available at the input ports, that defines the condition under which that pattern is used. Switches perform no computation other than the evaluation of connection pattern guards.

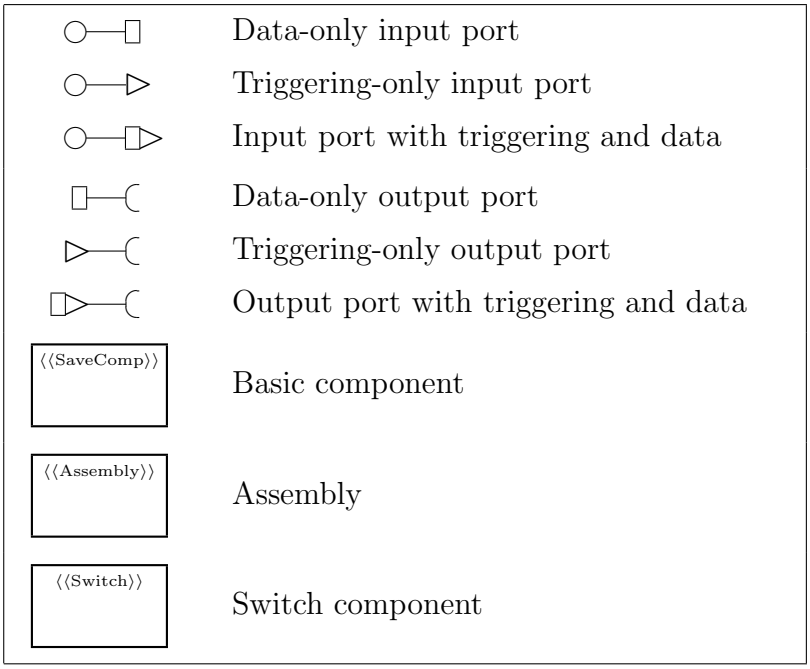

Fig. 1. The graphical notation of SaveCCM.

3.2 Execution model and run-time framework

On a system level, execution can be triggered by clocks or external events. Then, the control propagates through the system according to the triggering port connections. A component is triggered once all of its input triggering ports have been activated. If more than one output triggering port is connected to the same input triggering port, this input port is activated as soon as one of the connected output ports are activated.

When a component is activated, it first reads from all its input ports and then performs the associated computation. Then, output is written to output ports, which includes activating all output triggering ports of the component. Finally, all input triggering ports are reset to a non-active state.

The application domain of embedded systems requires a small runtime-framework that fits the current practice of this field. E.g., the prototype generates code for the RTXC real-time operating system [11], where a system is implemented as a set of periodic tasks with known worst case execution times, deadlines, and priorities governing the execution order.

At compile time, the components are allocated to tasks in such a way that triggering conditions, precedence relations and component communication are preserved. An analysis phase derives task properties from component attributes and from the system architecture, and performs further analysis based on these, e.g., response time and schedulability analysis. Finally, target specific code is generated for each task, where calls to the component entry function are interleaved with code that handles data exchange between components within the task and with other tasks.

4 Extended triggering

We propose an extension to the triggering mechanism that allows more elaborate triggering conditions to be specified. This functionality is provided in the form of an event algebra, i.e., a number of operators from which expressions can be constructed that represent complex triggering conditions.

First, we give an informal description of the algebra operators, and show how the event algebra is incorporated into SaveCCM. The formal semantics and a number of important properties of the algebra is discussed in Section 5. For a detailed description of the algebra, including implementation details, the reader is referred to [2] or [3].

4.1 Operators

Expressions are built recursively from primitive events, represented by input port names, and the operators of the algebra. Fig. 2 lists the operators, together with an informal description of their meaning.

Operator	Notation	Informal meaning
Disjunction	$A \vee B$	occurs when A or B (or both) occurs.
Conjunction	$A + B$	occurs when A and B have occurred (in any order, and possibly not simultaneously).
Negation	$A - B$	occurs when there is an occurrence of A, during which B does not occur.
Sequence	$A;B$	occurs when an occurrence of A is followed by an occurrence of B.
Within	A_τ	occurs when there is an occurrence of A shorter than τ time units.

Fig. 2. Informal description of the algebra operators.

As an example, the triggering condition from in the introductory example, that B occurs twice within two time units and neither P nor T occurs in between, would be defined by the expression $(\mathsf{B};\mathsf{B})_2 - (\mathsf{P} \vee \mathsf{T})$.

4.2 Extended notation

To incorporate the algebra into the component model, the SaveCCM notation is extended with a new element called *event*. An example of a this construct is shown in Fig. 3. An event element has a number of named input ports and one output port. To illustrate that the semantics of these ports differ from the ordinary ports, they are not represented graphically. The event element

is also associated with an algebraic expression defining the event pattern it is responsible for detecting.

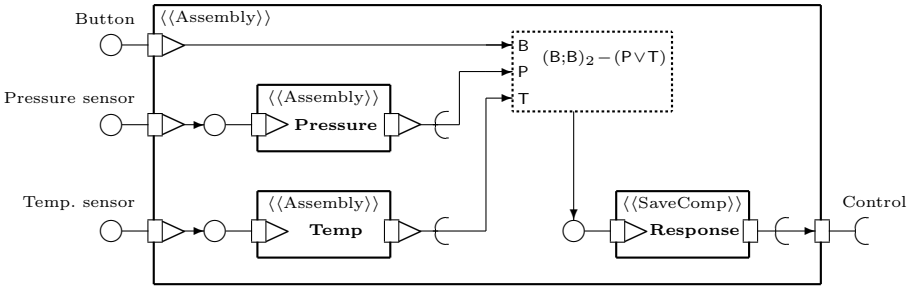

Fig. 3. An example of a system design that includes an event element.

Currently, the semantics of the event element is defined in terms of the original notation. The main motivation for this is that the extension can be included in the prototype tool with minimal effort. In the future, we intend to develop a more direct semantics for the event element, and handle them explicitly in the analysis and code generation phases to avoid unnecessary overhead.

Definition 4.1 An event element with n input ports is viewed as syntactic sugar for a collection of a basic component B, a switch S and n auxiliary basic components, as outlined in Fig. 4.

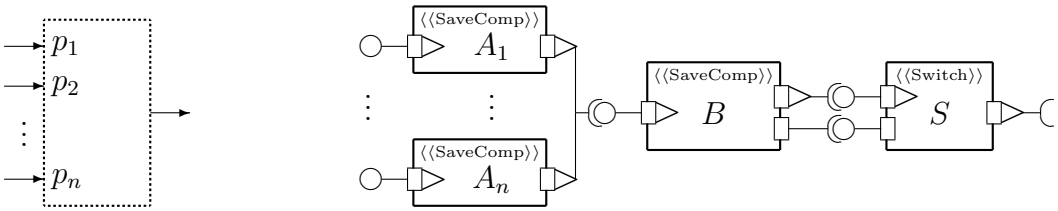

Fig. 4. An event element (left), and its counterpart in the original notation (right).

The basic component B is responsible for the computation and state information needed to detect the event pattern correctly. Code for this component is automatically generated from the expression of the event element, following the algebra implementation presented in [2].

The switch S determines, based on the output of B, if the triggering should be forwarded or not. The connection pattern condition is independent of the expression to be detected.

Additionally, an auxiliary component A_i is generated for each of the n input ports of the event element. These components are very small, and only serve to tag the data with a timestamp and the corresponding port name before relaying them to B. To ensure that an activation of one of the input ports is processed by B before it is overwritten by an activation of another port, we require that B is given a higher priority than the auxiliary components.

We believe that the overhead introduced by the auxiliary components is reasonably low. They will typically be allocated at the end of existing tasks, and the component code can be inlined by the compiler since it is only accessed from a single point. Explicit handling of event elements in the code generation phase would remove the need for these auxiliary components.

5 Event algebra details

From the perspective of the algebra, the input ports of the event element are viewed as event sources. Conversely, to the rest of the system the output port acts as an event source, emitting a triggering signal whenever the incoming event sequence matches the pattern defined by the algebraic expression. Note that the SaveCCM triggering signals are instantaneous, but the algebra associates event instances, i.e., both port activations and detections of subexpressions, with time intervals to ensure the desired algebraic properties.

Before we consider the algebra semantics, a few basic concepts have to be defined that connects the algebra with concepts in the component model.

Definition 5.1 Let the temporal domain $\mathcal{T}$ be the set of all natural numbers, and let $\mathcal{P}$ be a set containing the names of the input ports of the event element. For each $p \in \mathcal{P}$, let type(p) denote the data type of p.

5.1 Input ports

An activation of an input port is characterised by the port name, occurrence time and the associated data. Formally, we represent each activation as a singleton set to allow uniform treatment of primitive and complex event instances. Together, all activations of a certain port during the system lifetime form an event stream.

Definition 5.2 If $p \in \mathcal{P}$, $v \in \text{type}(p)$ and $\tau \in \mathcal{T}$, then the singleton set $\{\langle p, v, \tau\rangle\}$ is a *primitive event instance*. A *primitive event stream* is a set of primitive event instances all of which are labelled with the same port name and with different timestamps.

An interpretation is a formal representation of a single execution scenario, as it defines one of the possible ways in which the input ports of the event component can be activated.

Definition 5.3 An *interpretation* is a function mapping each input port $p \in \mathcal{P}$ to a primitive event stream with instances labelled with p.

As an example, let $\mathcal{P} = \{\mathsf{T}, \mathsf{P}\}$, type($\mathsf{T}$) = $\mathbb{N}$ and type(P) = {high, low}. Now $S = \{\{\langle \mathsf{T}, 12, 2\rangle\},\ \{\langle \mathsf{T}, 14, 3\rangle\},\ \{\langle \mathsf{T}, 8, 5\rangle\}\}$ and $S' = \{\{\langle \mathsf{P}, \text{low}, 4\rangle\}\}$ are examples of primitive event streams. The interpretation $\mathcal{I}$ such that $\mathcal{I}(\mathsf{T}) = S$ and $\mathcal{I}(\mathsf{P}) = S'$ represents a scenario where T is activated at times 2, 3 and 5, and P at time 4.

5.2 Event expressions

Event expressions are built from input port names and the operators of the algebra, as outlined in Section 4. As a first step of defining the meaning of an expression, we extend the concepts of instances and streams, defined for input ports above, to event expressions.

Definition 5.4 An *event instance* is a union of n primitive event instances, where $0 < n$, and an *event stream* is a set of event instances. For an event instance a we define:

$$\begin{aligned} \text{start}(a) &= \min(\,\{\tau \mid \langle p, v, \tau\rangle \in a\}\,) \\ \text{end}(a) &= \max(\,\{\tau \mid \langle p, v, \tau\rangle \in a\}\,) \end{aligned}$$

Informally, an event instance represents a number of input port activations that together match the event pattern described by the expression. We associate with each instance a an interval $[\text{start}(a), \text{end}(a)]$ that can be thought of as the smallest interval which contains all input port activations that caused a.

Note that a primitive event instance is an event instance, and if a is a primitive instance then $\text{start}(a) = \text{end}(a)$. Similarly, a primitive event stream is an event stream, just as the names suggest.

As an example, let $a = \{\ \langle \mathsf{T}, 12, 2\rangle,\ \langle \mathsf{P}, \text{low}, 4\rangle,\ \langle \mathsf{T}, 8, 5\rangle\ \}$. Then a is an event instance, with $\text{start}(a) = 2$ and $\text{end}(a) = 5$.

5.3 Operator semantics and the restriction policy

The interpretation represents input port activations by mapping each port name to an event stream, and the role of the algebra semantics is to extend this mapping so that a given event expression is mapped onto an event stream with exactly those instances that match the pattern described by the expression.

Definition 5.5 The meaning of an event expression for a given interpretation $\mathcal{I}$ is defined as follows:

$$\begin{aligned} [\![A]\!]^{\mathcal{I}} &= \mathcal{I}(A) \text{ if } A \in \mathcal{P} \\ [\![A \vee B]\!]^{\mathcal{I}} &= [\![A]\!]^{\mathcal{I}} \cup [\![B]\!]^{\mathcal{I}} \\ [\![A + B]\!]^{\mathcal{I}} &= \{a \cup b \mid a \in [\![A]\!]^{\mathcal{I}} \wedge b \in [\![B]\!]^{\mathcal{I}}\} \\ [\![A - B]\!]^{\mathcal{I}} &= \{a \mid a \in [\![A]\!]^{\mathcal{I}} \wedge \neg\exists b (b \in [\![B]\!]^{\mathcal{I}} \wedge \text{start}(a) \leq \text{start}(b) \wedge \\ &\qquad\qquad \text{end}(b) \leq \text{end}(a))\} \\ [\![A;B]\!]^{\mathcal{I}} &= \{a \cup b \mid a \in [\![A]\!]^{\mathcal{I}} \wedge b \in [\![B]\!]^{\mathcal{I}} \wedge \text{end}(a) < \text{start}(b)\} \\ [\![A_\tau]\!]^{\mathcal{I}} &= \{a \mid a \in [\![A]\!]^{\mathcal{I}} \wedge \text{end}(a) - \text{start}(a) \leq \tau\} \end{aligned}$$

These definitions result in an algebra with simple semantics and intuitive algebraic properties. However, it can not be implemented with limited resources since the conjunction and sequence operators require that instances are stored throughout the system lifetime. To deal with resource limitations, we define a formal restriction policy, and require only that an implementation should compute a valid restriction of the event stream specified by the algebra semantics above.

The restriction policy is defined as a binary relation *rem* over event streams, where $rem(S, S')$ means that S' is a valid restriction of S. For reasons of repeatability, it is typically desirable that an implementation of the algebra is deterministic. From a theoretical point of view, however, we prefer to leave as many detailed design decisions as possible open, and guarantee that any implementation which is consistent with the restriction policy relation have the properties described in the paper.

Definition 5.6 For event streams S and S', $rem(S, S')$ holds if the following conditions hold:

- $S' \subseteq S$
- For any $s \in S$ there exists a $s' \in S'$ such that $\text{start}(s) \leq \text{start}(s')$ and $\text{end}(s) = \text{end}(s')$.
- No instances in S' have the same end time.

Rather than computing $[\![A]\!]^{\mathcal{I}}$ for a given event expression A, an implementation of the algebra should result in an event stream S for which $rem([\![A]\!]^{\mathcal{I}}, S)$ holds. For the user of the algebra, this means that at any time when there is one or more occurrences of A, according to the $[\![A]\!]^{\mathcal{I}}$ semantics, one of them will be detected.

5.4 Properties

In order to investigate the properties of the event algebra, we need a well-defined concept of expression equivalence.

Definition 5.7 For event expressions A and B we define $A \equiv B$ to hold if $[\![A]\!]^{\mathcal{I}} = [\![B]\!]^{\mathcal{I}}$ for any interpretation $\mathcal{I}$.

Trivially, $\equiv$ is an equivalence relation. Moreover, it satisfies the substitutive condition, meaning that if a subexpression is changed into something equivalent, the result is equivalent to the original expression.

The following laws describe a number of important expression equivalences that facilitate reasoning, both formally and informally, about the triggering condition defined by an event element. They also show to what extent the algebra operators behave according to intuition. For a more extensive set of laws, and formal proofs, the reader is referred to [2].

Theorem 5.8 *These laws hold for event expressions A, B and C, and $\tau \in \mathcal{T}$.*

$$
\begin{array}{rcl}
A \vee A & \equiv & A \\
A \vee B & \equiv & B \vee A \\
A + B & \equiv & B + A \\
A \vee (B \vee C) & \equiv & (A \vee B) \vee C \\
A + (B + C) & \equiv & (A + B) + C \\
A ; (B ; C) & \equiv & (A ; B) ; C \\
(A \vee B) + C & \equiv & (A + C) \vee (B + C) \\
(A \vee B) ; C & \equiv & (A ; C) \vee (B ; C) \\
A ; (B \vee C) & \equiv & (A ; B) \vee (A ; C)
\end{array}
\qquad
\begin{array}{rcl}
A & \equiv & A_\tau \quad \text{if } A \in \mathcal{P} \\
(A_\tau)_{\tau'} & \equiv & A_{\min(\tau,\tau')} \\
(A \vee B)_\tau & \equiv & (A_\tau) \vee (B_\tau) \\
(A + B)_\tau & \equiv & (A_\tau + B)_\tau \\
(A - B)_\tau & \equiv & (A_\tau) - B \\
(A - B)_\tau & \equiv & (A - (B_\tau))_\tau \\
(A ; B)_\tau & \equiv & (A_\tau ; B)_\tau \\
(A ; B)_\tau & \equiv & (A ; B_\tau)_\tau \\
(A - B) - C & \equiv & A - (B \vee C) \\
(A \vee B) - C & \equiv & (A - C) \vee (B - C)
\end{array}
$$

The laws identify expressions that are semantically equivalent, but in order to handle resource limitations we expect an implementation of the algebra to compute an event stream S such that $rem([\![A]\!]^{\mathcal{I}}, S)$, rather than computing $[\![A]\!]^{\mathcal{I}}$. As a result, detecting A might yield a different stream than detecting A', even when $A \equiv A'$. Consequently, it should be clarified to what extent the laws presented above are still applicable when restriction is applied.

Theorem 5.9 *If $A \equiv A'$ and $rem([\![A]\!]^{\mathcal{I}}, S)$ holds, then $rem([\![A']\!]^{\mathcal{I}}, S)$ holds.*

Proof. This follows trivially from Definition 5.7, since $A \equiv A'$ implies that $[\![A]\!]^{\mathcal{I}} = [\![A']\!]^{\mathcal{I}}$. □

Thus, $A \equiv A'$ ensures that the result of an implementation detecting A is always a valid result for A'. Any reasoning based on the algebra semantics and the restriction policy, and not on the details of a particular detection algorithm, will be equally valid for equivalent expressions.

5.5 Analysis

An argument for extending the component model with an event algebra is that it facilitates analysis on a system design level, compared to developing a new component for each triggering condition. The algebraic laws presented above can be used to rewrite expressions into a form that can be more efficiently detected, e.g., as illustrated by the transformation algorithm presented in earlier work [2]. From the component model point of view, we want to be able to infer a number of properties of an event element based on the expression and properties of the components connected to it.

Memory requirement
The memory requirements of an event element can be directly determined from the expression and the types of the input ports. Unlike earlier work on the event algebra, the current implementation can detect any expression with limited memory.

Worst case execution time
The code generated for the detection component is characterised by a very simple control flow. For example, there are no nested loops and no function calls. Once the memory analysis derives the required storage structure sizes, all loops are trivially bounded. This means that the code could be analysed with standard worst case execution time techniques.

Alternatively, it should be fairly straightforward to determine, directly from the expression, an abstract worst case execution time in terms of the number of assignments, comparisons, arithmetical operations, etc.

Triggering frequencies
In order to guarantee timely responses when parts of the system activities are non-periodic, it is essential to have information about how often a given component is triggered. In real-time scheduling theory, the term *sporadic* is used for activities for which a lower bound on the time between two consecutive triggerings (minimum interarrival time) is known.

If the minimum interarrival time is known for the ports connected to the event element, this property can be derived for the output port as well. However, operators like disjunction and conjunction result in expressions with zero minimum interarrival time. We have defined a more general notion of maximum occurrences, computed by a function occ that guarantees that during any interval of length τ, there are at most $\mathrm{occ}(A, \tau)$ detections of the expression A. This can be computed for an arbitrary expression and interval length if minimum interarrival times, or occ values, are known for the input ports of the event element [2].

6 Conclusions and future work

We have presented how the component triggering in SaveCCM, a component model intended for embedded vehicular systems, can be extended by means of an event algebra. Using the algebra operators, complex patterns of triggering port activations can be defined, and whenever the activations match this pattern, the associated components are triggered.

Separating the detection of triggering conditions from the definition of the triggered service permits more general components and thus improves component reusability. Providing event detection as a part of the component model, rather than implementing it within an ordinary component, means that it is available for analysis at design-time.

The event algebra has been developed with two main considerations in mind: It should comply with laws that intuitively ought to hold for the algebra operators, and there should be an implementation that correctly detects any expression with limited memory. The simple operator semantics, and the expression equivalence laws, facilitates formal and informal reasoning about the triggering conditions in a system.

Our ongoing work includes defining a more direct semantics for the event element, rather than defining it in terms of other architectural elements. The compile-time phases of the prototype tool should be modified accordingly, so that the event detection activities are implemented as efficiently as possible. Then, case studies should be carried out to evaluate the usefulness of the method and to identify possible improvements.

References

[1] Åkerholm, M., *A software component technology for vehicle control systems*, Licentiate thesis No. 44 (2005), Mälardalen University, Sweden.

[2] Carlson, J., *An intuitive and resource-efficient event detection algebra*, Licentiate thesis No. 29 (2004), Mälardalen University, Sweden.

[3] Carlson, J. and B. Lisper, *An event detection algebra for reactive systems*, in: *Proc. 4th ACM Int Conference on Embedded Software (EMSOFT)* (2004).

[4] Chakravarthy, S. and D. Mishra, *Snoop: An expressive event specification language for active databases*, Data Knowledge Engineering **14** (1994), pp. 1–26.

[5] de Jonge, M., J. Muskens and M. Chaudron, *Scenario-based prediction of run-time resource consumption in component-based software systems*, in: *Proc. 6th Int. Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction* (2003).

[6] Galton, A. and J. C. Augusto, *Two approaches to event definition*, in: *Proc. 13th Int. Conference on Database and Expert Systems Applications*, Lecture Notes in Computer Science **2453** (2002).

[7] Gatziu, S. and K. R. Dittrich, *Events in an active object-oriented database system*, in: *Proc. 1st Int. Workshop on Rules in Database Systems* (1993).

[8] Gehani, N., H. V. Jagadish and O. Shmueli, *COMPOSE: A system for composite specification and detection*, in: *Advanced Database Systems*, Lecture Notes in Computer Science **759** (1993).

[9] Gruber, R., B. Krishnamurthy and E. Panagos, *The architecture of the READY event notification service*, in: *Proc. 19th IEEE Int. Conference on Distributed Computing Systems, Middleware Workshop*, Austin, TX, USA, 1999.

[10] Lundbäck, K.-L., J. Lundbäck and M. Lindberg, *Development of dependable real-time applications* (2004).
URL http://www.arcticus.se

[11] Quadros Systems Inc, *RTXC kernel users guide* (2004).
URL http://www.quadros.com

[12] Sánchez, C., S. Sankaranarayanan, H. Sipma, T. Zhang, D. Dill and Z. Manna, *Event correlation: Language and semantics*, in: *Proc. 3rd Int. Conference on Embedded Software*, Lecture Notes in Computer Science **2855** (2003).

[13] Stewart, D. B., R. A. Volpe and K. Khosla, *Design of dynamically reconfigurable real-time software using port-based objects*, IEEE Transactions on Software Engineering **23** (1997), pp. 759–776.

[14] van Ommering, R., F. van der Linden, K. Kramer and J. Magee, *The Koala component model for consumer electronics software*, Computer **33** (2000), pp. 78–85.

[15] Winter, M., T. Genßler, C. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, M. Mueller, C. Stich and S. Schoenhage, *Components for embedded software — the PECOS approach*, in: *Proc. 2nd Int. Workshop on Composition Languages*, 2002.