Towards a lightweight CEP engine for embedded systems

Paweł Pietrzak, Per Lindgren, Henrik Mäkitaavola
Department of Computer Science, Electrical and Space Engineering
Luleå University of Technology,
971 87 Luleå, Sweden
Email: pawel.pietrzak@ltu.se, per.lindgren@ltu.se, henrik.makitaavola@ltu.se

***Abstract*—Industrial process automation systems are adopting event based communication. Pushing control loops towards low-level devices implies a need for lightweight embedded devices that are able to recognize and to react to events. Atomic events however, such as a value read by an individual sensor exceeding certain value, do not separately suffice to capture scenarios where a reaction should occur to a sequence of low-level events matching certain pattern, rather than to a single atomic event. Therefore, it becomes desirable that resource-constrained low-level devices are equipped with some, possibly lightweight, form of event filtering and processing. In this paper we propose to implement a lightweight complex event processing using the concurrent reactive objects (CRO) model. A core feature of the CRO model is its s ability to react to atomic events. Between the reactions, which basically are function executions, the system remains idle, and thus does not occupy the CPU and is energy-efficient. Additionally, CRO models can be executed in an efficient and predictable manner onto resource constrained platforms and offers low-overhead real-time scheduling through exploiting underlying interrupt hardware according to given time constraints.**

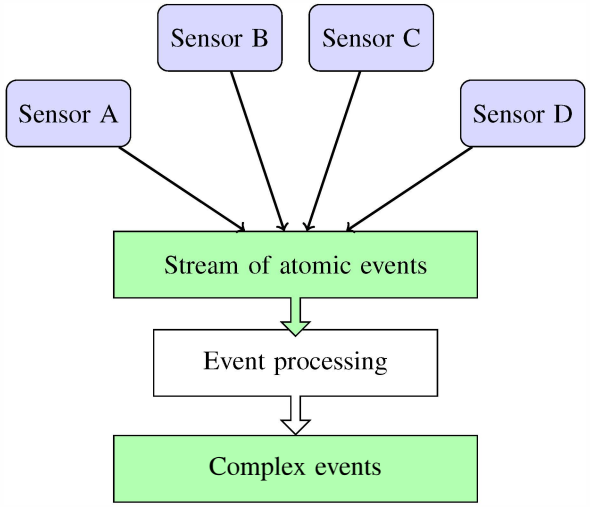


Fig. 1. Basic event processing

I. Introduction

Process industry systems become nowadays larger and more complex than before. They often consist of heterogeneous, distributed components, which might themselves be subsystems. In order to cope with this increasing complexity hardware-independent, asynchronous communication methods have to be employed. Such a flexible mechanism can be built based on the concepts of services [1], [2] and events [3]. We are focused on events and (complex) event processing (CEP), which has become especially popular in the domain of business process management. It is a technology to derive and analyze higher-level information out of low-level, or atomic, events. The feature set for CEP spans from event extraction, sampling, filtering correlation and aggregation to event enrichment, content based routing, event compositions (and not limited to these).

Normally, complex events are created by abstracting from low-level events, which can be thought of as sensor readings, as depicted in Figure 1. The processing of events is expressed within a specific language in terms of rules. Unfortunately, the set of features and the way to express the rules differ from platform to platform.

At the same time, as computing devices have become cheaper and more powerful, there is a growing tendency in process industry to push control loops down towards the hardware, so that the decisions are to be taken at low-level, and communication bandwidth (and energy for that matter) can be spared. As a consequence of pushing this control functionality downwords, lightweight embedded devices should be able to recognize and to react to events. One approach of interchanging events between devices is the use of standardized service interfaces, as shown by Kyusakov et al. [4], and the programming paradigm of Service Oriented Architecture. Atomic events however, such as a value read by an individual sensor exceeding certain value, do not separately suffice to capture scenarios where a reaction should occur to a sequence of low-level events matching certain pattern, rather than to a single atomic event. Therefore, it becomes desirable that resource-constrained low-level devices are equipped with some, possibly lightweight, form of event filtering and processing.

In this paper we show an ongoing effort that aims at implementing a lightweight CEP using the concurrent reactive objects (CRO) model. As explained in Section II-B, a core feature of the CRO model is its ability to react to atomic events. Between the reactions, which basically are function executions, the system remains idle, and thus does not occupy the CPU and is energy-efficient. Additionally, CRO models

978-1-4673-2421-2/12/$31.00 ©2012 IEEE

can be executed in an efficient and predictable manner onto resource constrained platforms and offers low-overhead real-time scheduling through exploiting underlying interrupt hardware according to given time constraints.

Event patterns to be recognized by a CEP system are specified in a query language, such as the CEDR language presented in [5]. We propose a scheme to implement CEDR queries using the CRO model. Semantic similarities between the two make it possible to design a fully automatic translation. The technique proposed in the paper is based on compile-time translation, whose aim is to create tailored highly efficient and predictable CEP system for resource constrained platforms. With the proposed concept we are able to build a scalable event processing pipeline, enabling event processing on several levels with full control of data traffic and latency.

There have been a lot of efforts on processing data on resource-constrained devices, typically on sensor nodes in wireless sensor networks, see e.g. SwissQM [6] or TinyDB [7], [8]. These approaches however differ from ours in that nodes actively query for data, whereas in our approach nodes react on external events. Event-based programming is a core concept behind Contiki - a tiny operating system targeted for wireless sensor nodes [9].

An event notification service in distributed, heterogeneous systems based on the publish/subscribe paradigm, is proposed in the system READY [10]. Asynchronous communication over wireless ad hoc networks is addressed by Yoneki [11]. An event algebra specialized to recognize event patterns in resource-constrained devices is discussed by Carlson and Lisper in [12].

II. Background

A. Events and event processing

An event is a record of something that has or has not occurred. Basic events, also referred to as *atomic events*, carry raw unprocessed information. In the context of process automation systems one can think of atomic events as of sensor readings, for instance. Atomic events form an *event stream* which is a subject of *event processing*. Event processing, matches event pattern in the stream, filters interesting events and possibly transform the detected pattern into a new (complex) event. Events can be pattern-matched with respect to their temporal properties or values that they carry in their payloads. Event patterns are expressed by an event query language, an example of which is CEDR [13], [5].

1) The CEDR query language: Below we outline the CEDR query language especially the subset of the language which is a subject of our interest.

CEDR is a declarative language to express queries over event streams. A CEDR query captures event types and order, possibly combined with time and/or value correlation and optional lifetime. As a part of value correlation, an CEDR query can filter (or filter out) certain instances of events. A CEDR query has the general form:

EVENT	$< query\ name >$
WHEN	$< event\ expression >$
WHERE	$< correlation\ expression >$
OUTPUT	$< instance\ transformation\ conditions >$

Throughout this paper we are mostly interested in the **WHEN** clause of *event filtering*. The $< event\ expression >$ part consists of events (event types) and operators. Operators evaluate to True or False and output an event in case of True. Examples of operators are event sequencing operators (E_i is an event expression):

- $\texttt{ALL}(E_1, \ldots, E_k)$ evaluates to True iff every of event patterns E_i occurs with no particular order imposed; outputs all instances of $E_1, \ldots, E_k$ that occurred at a given time point;
- $\texttt{ANY}(E_1, \ldots, E_k)$ evaluates to True iff any of E_i occurs; it outputs E_i that occurred;
- $\texttt{SEQUENCE}(E_1, \ldots, E_k)$ evaluates to True iff all the events E_i occur in the specified order, although arbitrary events may occur between any E_i and E_{i+1};

In some scenarios event filtering system recognizes non-occurrence of an event. An example of related operator is

- $\texttt{UNLESS}(E_1, E_2, t)$ detects an occurrence of E_1 followed by a non-occurrence of E_2 within a timespan t.

The **WHERE** clause expresses correlation between values in events' payload, where event instances matched by corresponding patterns the **WHEN** clause are assigned to local variables by means of **AS** operator, somewhat similarly to **AS** operator in SQL. We access the payload as expressions' attributes, using common dot-notation. Consider the following example query:

EVENT	$Master\_alarm$
WHEN	$\texttt{UNLESS}(\texttt{ANY}(High\_temp, High\_press)$ **AS** $x,$ $Button\_pressed, 10$ **seconds**$)$
WHERE	$x.Tank\_Id = 'Tank\#1'$ **OR** $x.Tank\_Id = 'Tank\#3'$
OUTPUT	$Alarm\ x.Tank\_Id$

The query expresses a pattern of events in which a human operator fails to press the button within 10 seconds after any of the events *High_temp* or *High_press* on Tank #1 or Tank #3 has occurred. Note the role of the **WHERE** clause is used to verify the tank number, assuming that tank number is a part of the payload, and that the matched instance of an event is bound to x by the **AS** operator.

Since we are focused on detecting temporal correlation between events, in the subsequent parts we shall omit elements related to event instances, value correlation or validity time window. These aspects can be easily incorporated into our execution model.

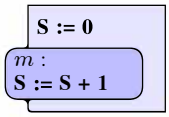
B. Concurrent reactive objects

1) The model: The concurrent reactive object (CRO) model is the execution and concurrency model of the Timber programming language, a general-purpose object-oriented language that primarily targets real-time systems [14], [15], [16]. A subset of C implementing the core features of Timber and also using the CRO model as its execution model is called

TinyTimber [17]. The CRO model facilitates reactivity; object-orientation with complete state encapsulation; object-level concurrency with message passing between objects, the ability to specify timing behavior of a system, and the abstraction to components.

2) Reactivity: Reactivity is the defining property of the CRO model, which makes it particularly suitable for embedded system, since functionality of most, if not all, embedded systems can be expressed in terms of reactions to external stimuli and timer events. A reactive system can be described as follows: initially the system is idle, an external stimulus (originating in the system's environment) or a timer event triggers a burst of activity, and eventually the system returns to the idle state. A reactive object is either actively executing a method in response to an external stimulus or a message from another object, or passively maintaining its state. Since initially the system is idle, some external stimulus is needed to trigger activity in the system.

3) Objects and state encapsulation: The CRO model specifies that all system state is encapsulated in objects $O_1, \ldots, O_n$. Each object has a number of methods, and the encapsulated state is only accessible from the object's methods. This is also known as a complete state encapsulation. A name of a method m can be fully expanded as $O_i : m$, where O_i is an object owning the method. Below we show a graphical representation of an object with a state variable **S** whose initial value is set to 0, and with a method m.



Methods of two objects can be executed concurrently, but each method is granted an exclusive access to its object's state, so only one method of an object can be active at any given time. Coupled with a complete state encapsulation, this provides a mechanism for guaranteeing state consistency under concurrent execution. The source of concurrency in a system can either be two external stimuli that are handled by different objects, or an asynchronous message sent from one object to another (more about message passing below).

To ensure that execution is reactive in its nature, each method must follow run-to-end semantics [18], i.e., it is not allowed to block execution awaiting external stimulus or a message. An example of this would be an object representing a queue: if a *dequeue* method is invoked on an empty queue, it is not allowed to wait until data becomes available, but it must instead return a result indicating that the queue is empty.

4) Message passing and specification of timing behavior: In the CRO model objects communicate by passing messages. Each message specifies a recipient object (O) and a method (m) of this object that will be invoked. A message is either synchronous ($\mathsf{SYNC}(O, m)$) or asynchronous ($\mathsf{ASYNC}(O, m)$). The sender of a synchronous message blocks waiting for the invoked method to complete, while the sender of an asynchronous message can continue execution concurrently with the invoked method. Thus asynchronous messages introduce concurrency into the system. An asynchronous message can also be delayed by a specific amount of time.

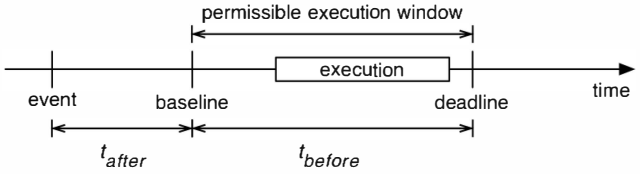


Fig. 2. Permissible execution window for a message.

Timing behavior of a system can be specified by defining a baseline and a deadline for an asynchronous message (a synchronous message always inherits the timing specification of the sender). The *baseline* specifies the earliest point in time when a message becomes eligible for execution, which for an external stimulus corresponds to its "arrival time" and for a message sent from one object to another is defined directly in the code. If the defined baseline is in the future, this corresponds to delaying the delivery of the message. The *deadline* specifies the latest point in time when a message must complete execution, which is always defined relative to the baseline, see Figure 2 for the permissible execution window of a message. Whenever we talk about timing behavior of asynchronous messages, we shall extend the notation to $\mathsf{ASYNC}(O, m, B, D)$, where B and D are respectively a baseline and a *relative* deadline of the message. As mentioned above, synchronous messages always inherit their timing specification from the initial asynchronous message.

Concurrent reactive objects can be used to model the system itself and its interaction with its environment (e.g., via sensors, buttons, keyboards, displays). Events in the physical world (such as pushing a button) result in an asynchronous message being sent to a handler object, and system output (e.g., flashing an LED) is represented as messages sent from an object to the environment.

5) Implementation of CRO: The implementation of an object instance can be either in software (e.g, synthesized from the class definition in the Timber language, or from a TinyTimber definition in C) or provided by the environment. This allows incorporating hardware interactions and legacy code (typically external software libraries) in the model, as long as their interface is compliant with the concurrent reactive object model.

III. Event processing in CRO

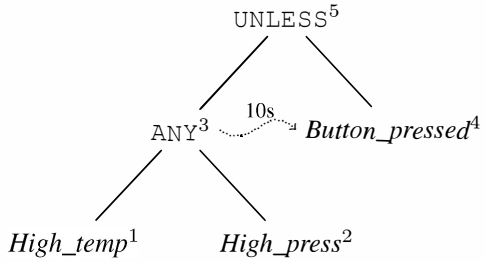
In this section we explain how to apply the CRO model for implementing event pattern filtering algorithms. Our proposal is based on translating a CEDR query into a CRO program, which can further be directly expressed in C [19], [17] Consider the example query from section II-A.

EVENT $Master\_alarm$
WHEN UNLESS(ANY($High\_temp, High\_press$) **AS** x, $Button\_pressed$, 10 **seconds**)
WHERE $x.Tank\_Id = 'Tank\#1'$ **OR** $x.Tank\_Id = 'Tank\#3'$
OUTPUT $Alarm\ x.Tank\_Id$

Its event recognizing **WHEN** clause is shown below. In order to avoid ambiguity every subexpression p in a **WHEN** clause is labeled with a label ℓ, written as p^ℓ.

$$\textbf{WHEN}\quad \texttt{UNLESS}(\texttt{ANY}(\mathit{High\_temp}^1, \mathit{High\_press}^2)^3\ \textbf{AS}\ x,\ \mathit{Button\_pressed}^4, 10\ \texttt{seconds})^5$$

An event pattern like the one above can be decomposed into subexpressions, each of them being an event pattern itself. The decomposition is illustrated below.



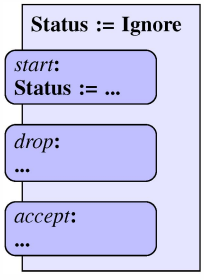
Every node corresponds to an event pattern, with possible further subexpressions. Atomic events are leaves in the pattern expression.

A process of recognizing an event can be in one of the four states:

- **Ignore** when the recognizer is not activated to filter the atomic events and listen to the input stream;
- **Await** when the even pattern is under recognition process, but it is not completed as more input is needed;
- **Accept** when the event pattern has been successfully accepted
- **Fail** when the pattern definitely is not occurring, regardless of subsequent events that will follow.

A similar set of states that an event pattern recognition process can be in has been defined in [20] within the formal model called *reactive machine*.

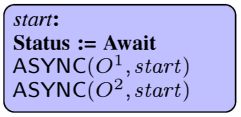
In this work we propose using the reactive objects paradigm. Our design principle is that with every event pattern p^ℓ (composite or atomic) we assign a reactive object, denoted O^ℓ, whose purpose is to provide main functionality of filtering p. Such an object contains one state variable **Status** and methods *start*, *accept*, and optionally *drop*.



The variable **Status** maintains the status of the recognizing process. The initial value of **Status** is **Ignore**. The object's methods are to used control behavior of the pattern recognizing objects and to provide communication, by means of asynchronous messages, between objects filtering event patterns and their sub-patterns. The method *start* initiates recognition of the corresponding pattern. So it always turns the status from **Ignore** to **Await**, and possibly triggers more actions, like initializing recognition of sub-patterns. In the context of embedded systems, atomic events manifest themselves as hardware interrupts, whose handlers are the *accept* methods of corresponding objects. In a case of a composite events the *accept* method is invoked by one of the sub-pattern objects according to the structure of the entire query. As an illustration assume the subexpression

$$\texttt{ANY}(\mathit{High\_temp}^1, \mathit{High\_press}^2)^3$$

of the working example, containing three event patterns - two atomic and one composite. The method O^3: *start* sets up the value of **Status** and asynchronously, yet without delay, activates the two components of the ANY operator.



where object O^1 recognizes *High_temp* and O^2 does so for *High_press*.

The complete set of objects implementing recognition of the working example in depicted in Figure 3. An interesting construct can be found in object O^4 that implements recognition of *Button_pressed*. In fact, what is to be detected in the context of the entire query is a *nonoccurrence* (or *negative occurrence*) of *Button_pressed* within 10 seconds after *High_temp* or *High_press*. Listening to *Button_pressed* is initialized by the pattern $\texttt{ANY}(\mathit{High\_temp}^1, \mathit{High\_press}^2)^3$ and remains active for 10 seconds. This functionality has been implemented with the CRO feature that allows sending messages with timing constraints, and here by sending the *drop* message by O^4 to itself. In effect O^4 stops waiting for the button to be pressed and sends the *accept* message to the parent UNLESS pattern.

IV. Future work

The presented work is an ongoing effort, and there are several directions we plan to continue.

- First of all we plan to develop a robust prototype for compilation a subset of CEDR queries into a CRO. Since TinyOS [21] real-time scheduler matches well with the CRO computation model, our approach can be implemented under TinyOS. Combining CRO and TinyOS is a subject of ongoing work [22]. Alternatively, the proposed approach to CEP can be implemented under the TinyTimber model [19], [17], which does not require a hosting operating system, but just a tiny platform-dependent run-time system.
  Further benchmarking and practical evaluation of the proposed method are needed. We plan to apply our lightweight CEP technique to an industrial case-study.
- Our approach is simplified in the sense that only single instance of an event is assumed. It has to be further studied how to handle multiple instances in terms of

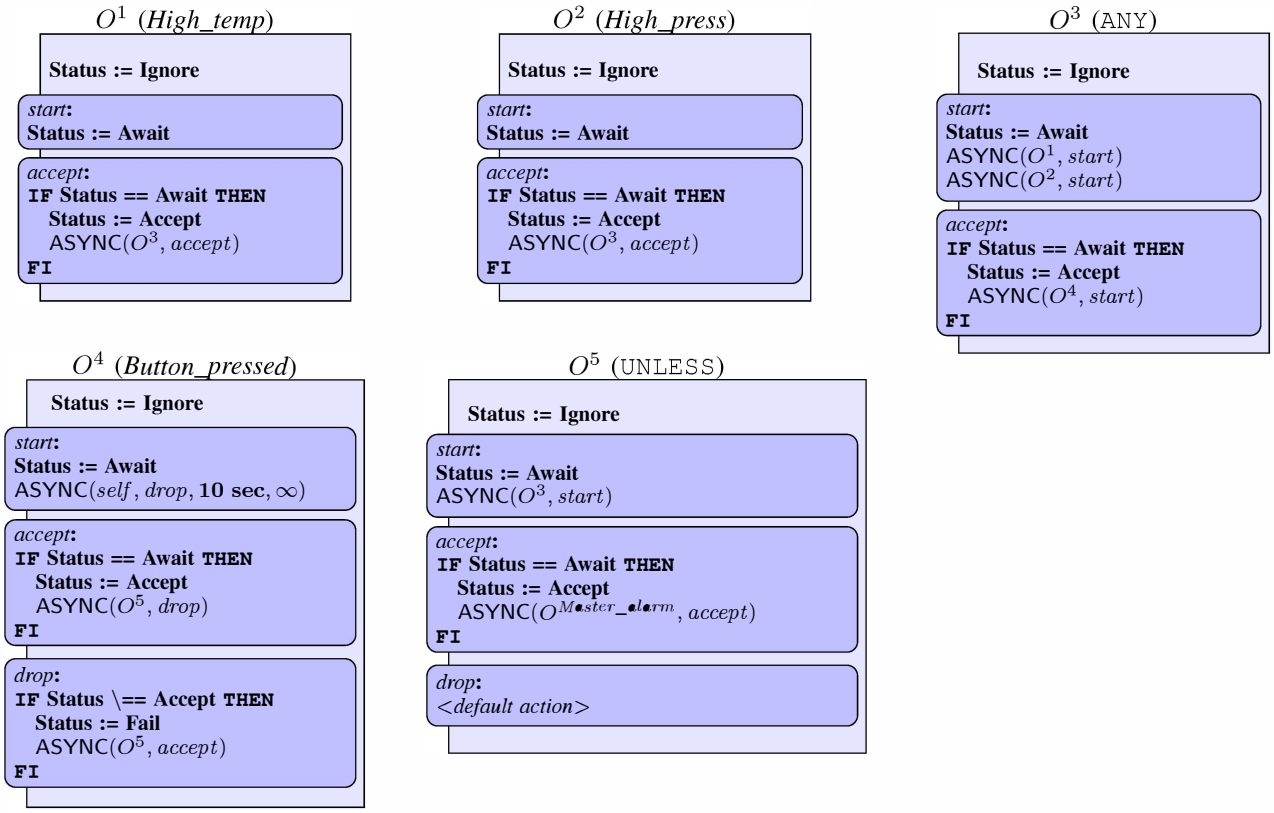


Fig. 3. The CRO implementation of the working example

algorithm details and memory requirements. It seems feasible to assume that in real scenario the maximum number of simultaneously active instances of the same event can be determined a priori, and thus multiple "clones" of an object corresponding to the event can be created at startup time. Alternatively overlapping event instances might be handled by state vectors (with index given as and argument to methods $accept(i)$, $drop(i)$, etc.).

- It is also desirable to carry an insightful comparison of our technique with algorithms in the literature, especially concerning memory and time efficiency (e.g. with [12]).
- Up to now we have studied a translation-based implementation technique of a specific query language into CRO. It would be beneficial to establish formal correctes of such a translation, departing from more mature formal models of event processing, such as [12], [20], or [23]. On the CRO side we can refer to formal model of Timber [14].
- Since our CRO-based implementation is basically a real-time reactive system, one can study its schedulability under various scheduling policies, and possibly link it with temporal characteristics of input event stream as well as different types of queries.
- For our static mapping of CEDR query to a CRO program, where the object hierarchy and the communication channels between the objects are determined at compile-time, it should be feasible to design a suite of program analyses that provide static guarantees of run-time behavior [24]. What is particularly interesting in context of embedded platforms is resource usage, including memory and execution time [25].
  For systems with dynamic memory allocation, heap usage can however be statically predicted (see e.g. [26], [27]), and it would be interesting to study how the characteristics of input event stream (given for instance in terms of minimum and maximun inter-arrival times of atomic events) along with some information about dynamically created queries, can influence predictability and accuracy of static analysis as well as garbage collection [28], [29].
- Since our proposed scheme is primarily based on compile-time translation, it is not suitable for queries created dynamically at runtime. One can however think of techniques based on a form of run-time interpreting, which would accept dynamic queries as well. This will inevitably lead to dynamic memory allocations and difficult performance predictability. A possible middle ground in between the two would be to synthesize a set of pre-allocated CROs for the CEP and allow run-time configuration. In that case the problem of implementing a specific set of rules boils down to finding a configuration (set of parameters and communication patterns) that meets the requirements. That would allow for dynamically changing the functionality without changing the code base running on the node(s) and would allow (for a configuration) to predict performance.

V. Conclusions

In this paper we have demonstrated how to use the concept of concurrent reactive object to implement CEP on small, resource constrained embedded devices. The idea is based on compile-time translation of a CEP query expressed in the CEDR query language, into a set of concurrent reactive object. Interconnections between the objects reflect logical links between subexpressions of the query. This is an ongoing work, therefore, future work has been also outlined.

Acknowledgments

This work is funded by the EU FP7 Project "AESOP". The authors would like to thank partners of the AESOP project for discussions.

References

[1] A. Colombo and S. Karnouskos, "Towards the factory of the future: A service-oriented cross-layer infrastructure," in *ICT Shaping The World - A Scientific View, The European Telecommunications Standards Institute (ETSI)*. John Wiley and Sons Ed., April 2009, ch. 6.

[2] S. Karnouskos, A. Colombo, F. Jammes, J. Delsing, and T. Bangemann, "Towards an architecture for service-oriented process monitoring and control," in *IECON 2010, (The 36th Annual Conference of the IEEE Industrial Electronics Society)*, 2010.

[3] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed Event-Based Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[4] R. Kyusakov, J. Eliasson, J. Delsing, J. van Deventer, and J. Gustafsson, "Integration of wireless sensor and actuator nodes with it infrastructure using service-oriented architecture," *IEEE Transactions on Industrial Informatics*, 2012, Accepted.

[5] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," in *CIDR*, 2007, pp. 363–374.

[6] R. Mueller, G. Alonso, and D. Kossmann, "SwissQM: Next generation data processing in sensor networks," in *In: Third Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.

[7] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, Mar. 2005.

[8] S. Madden, W. Hong, M. Franklin, and J. Hellerstein, "Tinydb web page. `http://telegraph.cs.berkeley.edu/tinydb`," 2012.

[9] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462.

[10] R. Gruber, B. Krishnamurthy, and E. Panagos., "The architecture of the READY event notification service," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Middleware Workshop, Austin, TX, USA*, P. Dasgupta, Ed., May 1999.

[11] E. Yoneki, "ECCO: Data centric asynchronous communication," Ph.D. dissertation, University of Cambridge, Computer Laboratory, December 2006.

[12] J. Carlson and B. Lisper, "A resource-efficient event algebra," *Science of Computer Programming*, vol. 75, no. 12, pp. 1215–1234, December 2010.

[13] R. Barga, H. Caituiro-Monge, T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, and J. Wijsen, *Event Correlation and Pattern Detection in CEDR*. Springer Berlin / Heidelberg, 2006, vol. 4254, pp. 919–930. [Online]. Available: http://dx.doi.org/10.1007/11896548_70

[14] M. Carlsson, J. Nordlander, and D. Kieburtz, "The semantic layers of Timber," in *First Asian Symp. on Programming Languages and Systems (APLAS)*, ser. Lecture Notes in Computer Science, vol. 2895. Berlin, Germany: Springer-Verlag, 2003, pp. 339–356.

[15] A. P. Black, M. Carlsson, M. P. Jones, R. Kieburtz, and J. Nordlander, "Timber: A programming language for real-time embedded systems," Oregon Graduate Institute School of Science & Engineering, Tech. Rep., 2002.

[16] The Timber Language. (webpage) Last accessed 2011-04-15. [Online]. Available: http://www.timber-lang.org

[17] J. Eriksson, "Embedded real-time software using TinyTimber : reactive objects in C," Licentiate Thesis, Luleå University of Technology, 2007.

[18] P. Lindgren, J. Nordlander, L. Svensson, and J. Eriksson, "Time for Timber," Luleå University of Technology, Tech. Rep., 2005.

[19] J. Nordlander, "Programming with TinyTimber kernel," Luleå University of Technology, Tech. Rep., January 2012.

[20] C. Sánchez, M. Slanina, H. B. Sipma, and Z. Manna, "The reaction algebra: A formal language for event correlation," in *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, ser. Lecture Notes in Computer Science, A. Avron, N. Dershowitz, and A. Rabinovich, Eds., vol. 4800. Springer-Verlag, 2008, pp. 589–609.

[21] TinyOS Community Forum, "An open-source os for the networked sensor regime `www.tinyos.net`," 2010.

[22] P. Lindgren, H. Mäkitaavola, J. Eriksson, and J. Eliasson, "Leveraging TinyOS for integration in process automation and control systems," in *IECON 2012, (The 38th Annual Conference of the IEEE Industrial Electronics Society)*, 2012.

[23] C. Sánchez, H. B. Sipma, M. Slanina, and Z. Manna, "Final semantics for Event-Pattern Reactive Programs," in *First International Conference in Algebra and Coalgebra in Computer Science (CALCO'05)*, ser. LNCS, J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. Rutten, Eds., vol. 3629. Springer-Verlag, September 2005, pp. 364–378.

[24] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 2005, second Ed.

[25] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[26] M. Kero, P. Pietrzak, and J. Nordlander, "Live Heap Space Bounds for Real-Time Systems," in *Programming Languages and Systems - 8th Asian Symposium, (APLAS'10)*, ser. Lecture Notes in Computer Science, vol. 6461, 2010, pp. 287–303.

[27] E. Albert, S. Genaim, and M. Gómez-Zamalloa, "Parametric Inference of Memory Requirements for Garbage Collected Languages," in *9th International Symposium on Memory Management (ISMM'10)*. New York, NY, USA: ACM Press, June 2010, pp. 121–130.

[28] M. Kero, J. Nordlander, and P. Lindgren, "A correct and useful incremental copying garbage collector," in *8th International Symposium on Memory Management (ISMM'07)*. ACM Press, 2007.

[29] M. Kero and S. Aittamaa, "Scheduling garbage collection in real-time systems," in *CODES*, 2010.