An architecture to support interaction via Generic Events *

Paulo Veríssimo
U.Lisboa
pjv@di.fc.ul.pt

Jörg Kaiser
U.Ulm
kaiser@informatik.uni-ulm.de

António Casimiro
U.Lisboa
casim@di.fc.ul.pt

1 Introduction

Classical event/object models are usually software oriented. As such, when transported to a real-time, embedded systems setting, their harmony is cluttered by the conflict between, on the one side, send/receive of "software" events (message-based), and on the other side, input/output of "hardware" or "real-world" events, register-based. In a recent paper we introduced the **Generic-Events Architecture (GEAR)**, which allows the seamless integration of physical and computer information flows. The GEAR architecture introduces the unifying concept of *generic event*, be it derived from the boolean indication of a door opening sensor, from the electrical signal embodying a network packet (at the WLAN aerial) or from the arrival of a temperature event message. In this paper we review some of the key issues of GEAR, in particular those related with information flow and support for real-time sentient applications.

GEAR defines an event layer that is responsible for event propagation in the whole system, through several *Event Channels (EC)*. In fact, this layer plays a fundamental role in securing the functional and non-functional (e.g. reliability and timeliness) properties of the envisaged applications. This paper proposes the **COSMIC (COoperating SMart devICes)** middleware for the event layer, showing how it integrates with the GEAR architecture. A particularly interesting issue, also discussed in this paper, is that COSMIC event channels can be of several synchrony classes, ranging from non real-time to hard real-time classes.

2 System Model and Communication Abstractions

In this paper we consider a component-based system model that incorporates some of the ideas developed in the context of the IST CORTEX project. A fundamental idea underlying the approach is that applications can be composed of a (possibly large) number of smart components that are able to sense their surrounding environment and interact with it. These components are referred to as *sentient objects*, a metaphor elaborated in CORTEX [7] and inspired on the generic concept of *sentient computing* introduced in [2]. Sentient objects accept input events from a variety of different sources (including sensors, but not constrained to that), process them, and produce output events, whereby they actuate on the environment and/or interact with other objects. In consequence, it is quite natural to base the communication and interaction among sentient objects on an event-based communication model. Moreover, typical properties of event-based models, such as anonymous and non-blocking communication, are highly desirable in systems where sentient objects can be mobile and where interactions are naturally very dynamic. On the other hand, the need to address timeliness or real-time requirements becomes a challenging issue, which, in our opinion, can only be faced with the use of appropriate architectural constructs.

However, before describing such constructs, let us clarify some fundamental ideas underlying the construction of systems composed of sentient objects.

2.1 Component-based system construction

Sentient objects can take several different forms: they can simply be software-based components, but they can also comprise mechanical and/or hardware parts, amongst which the very sensorial apparatus that substantiates "sentience", mixed with software components to accomplish their task. We refine this notion by considering a sentient object as an encapsulating entity, a component with internal logic and active processing elements, able to receive, transform and produce new events. This interface hides the internal hardware/software structure of the object, which may be complex, and shields the system from the low-level functional and temporal details of controlling a specific sensor or actuator.

Furthermore, given the inherent complexity of the envisaged applications, the number of simultaneous input events and the internal size of sentient objects may become too large and difficult to handle. Therefore, it should be possible to consider the hierarchical composition of sentient objects so that the application logic can be separated across as few or as many of these objects as necessary. On the other hand, composition of sentient objects should normally be constrained by the actual hardware component's structure, preventing the possibility of arbitrarily composing sentient objects. This is illustrated in Figure 1, where a sentient object is internally composed of a few other sentient objects, each of them consuming and producing events, some of which only internally propagated.

To give an example of such *component-aware* object composition we consider a scenario of cooperating robots. Each robot is made of several components, corresponding, for instance, to axis and manipulator controllers. Together with the control software, each of these controllers may be a sentient object. On the other hand, a robot itself is a sentient object, composed of the several controller objects inside his body.

*This work was partially supported by the EC, through project IST-2000-26031 (CORTEX), and by the FCT, through the Large-Scale Informatic Systems Laboratory (LaSIGE) and project POSI/1999/CHS/33996 (DEFEATS).

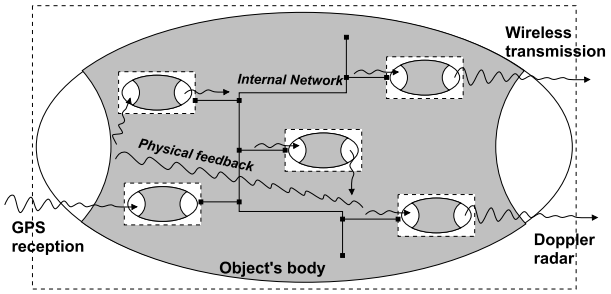

Figure 1: Component-aware sentient object composition.

2.2 The role of smart sensors

Besides the advantages of structuring large scale sentient computations, the component-based model of sentient objects is also motivated by the availability of smart sensor and actuators. These devices encapsulate hardware, software and mechanical components and provide a set of well-specified functions which are closely related to the interaction with the environment. Different from a general-purpose node which can execute any program, smart components are dedicated to a certain function which complies to their sensing and actuating capabilities. The built-in computational components network attachment enable the implementation of a well-defined high-level interface that does not just provide raw transducer data, but a pre-processed, application-related set of events, which can also be disseminated.

Smart components translate events of the environment to an appropriate form available at the event layer or, vice versa, transform a system event into an actuation. For smart components we can assume that:

- Smart components have dedicated resources to perform a specific function.
- These resources are not used for other purposes during normal real-time operation.
- No local temporal conflicts occur that will change the observable temporal behaviour.
- The functions of a component can usually only be changed during a configuration procedure which is not performed when the component is involved in critical operations.
- An observation of the environment as a ⟨time,value⟩ pair can be obtained with a bounded jitter in time.

Many predictability and scheduling problems arise from the fact, that very low-level timing behaviours have to be handled on a single processor. Here, temporal encapsulation of activities is difficult because of the possible side effects when sharing a single processor resource. Consider the control of a simple IR-range detector which is used for obstacle avoidance. On a single central processor, this critical activity has to be coordinated with many similar, possibly less critical functions. In a smart component, all the low-level timing behaviour can be optimized and encapsulated. Thus we can assume temporal encapsulation similar to information hiding in the functional domain. The main responsibility to provide timeliness guarantees is shifted to the event layer where these events are disseminated. Smart sensors thus lead to network centric system model as presented in the following sections.

2.3 Encapsulation and scoping

Now an important question is about how to represent and disseminate events in a large scale networked world. It is clear that the event layer should be universal, i.e. potentially every sentient object can talk to every other sentient object using the event channel (EC) dialect. However, there are substantial obstacles originating from the components heterogeneity in such a large-scale setting. Firstly, the components may have severe performance constraints, particularly because we want to integrate smart sensors and actuators in such an architecture. Secondly, the bandwidth of the participating networks may vary largely. Thirdly, the networks may have widely different reliability and timeliness characteristics. At the abstraction level of sentient objects, such heterogeneity is reflected by the notion of *body-vs-environment*. At the network level, we introduced the *WAN-of-CANs* structure to model the different networks.

The notion of body and environment is derived from the recursively defined component-based object model. A body is similar to a cell membrane and represents a quality of service container for the sentient objects inside. On the network level, it may be associated with the components coupled by a certain CAN. A CAN defines the dissemination quality which can be expected by the cooperating objects.

The notions of body-environment and WAN-of-CANs are very useful when defining interaction properties across such boundaries. Their introduction obeyed to our belief that a single mechanism to provide quality measures for interactions is not appropriate.

2.4 Architectural Building blocks for a generic event system

As we mentioned above, in order to apply event-based object-oriented models on real-time cooperative and embedded systems it is necessary to use adequate architectural constructs, which allow the enforcement of certain required properties.

We propose the **Generic-Events Architecture (GEAR)**, depicted in Figure 2, which we briefly describe in what follows (for a more detailed description please refer to [9]). The L-shaped structure is crucial to ensure some of the properties described.

Environment: The physical surroundings, remote and close, solid and etherial, of sentient objects. **Body:** The physical embodiment of a sentient object (e.g., the hardware where a mechatronic controller resides, the physical structure of a car). Note that due to the compositional approach taken in our model, part of what is 'environment' to a smaller object seen individually, becomes 'body' for a larger, containing object. In fact, the body is the 'internal environment' of the object. This architecture layering allows composition to take place seamlessly, in what concerns information flow. **Translation Layer:** The layer responsible for physical event transformation from/to their native form to event channel dialect, between environment/body and an event channel. Essentially one doing observation and actuation operations on the lower side, and doing transactions of event descriptions on the other. **Event Layer:** The layer responsible for event propagation in the whole system, through several *Event Chan-*

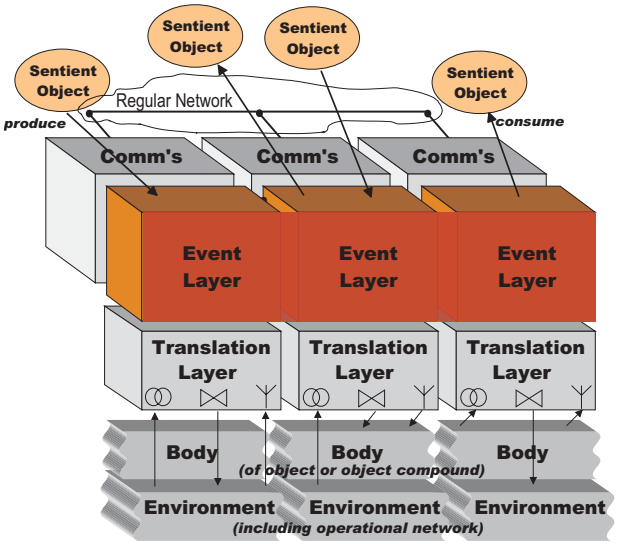

Figure 2: Generic-Events architecture.

nels (EC):. In concrete terms, this layer is a kind of middleware that provides important event-processing services which are crucial for any realistic event-based system. For example, some of the services that imply the processing of events may include publishing, subscribing, discrimination (zoning, filtering, fusion, tracing), and queuing. **Communication Layer:** The layer responsible for 'wrapping' events (as a matter of fact, event descriptions in EC dialect) into 'carrier' *event-messages*, to be transported to remote places. For example, a sensing event generated by a smart sensor is wrapped in an event-message and disseminated, to be caught by whoever is concerned. The same with an actuation event produced by a sentient object, to be delivered to a remote smart actuator. Likewise, this may apply to an event-message from one sentient object to another. **Regular Network:** This is represented in the horizontal axis of the block diagram by the Communication layer, which encompasses the usual LAN, TCP/IP, and real-time protocols, desirably augmented with reliable and/or ordered broadcast and other protocols.

The GEAR introduces some innovative ideas in distributed systems architecture. While serving an object model based on production and consumption of generic events, it treats events produced by several sources (environment, body, objects) in a homogeneous way. This is possible due to the use of a common basic dialect for talking about events and due to the existence of the Translation layer, which performs the necessary translation between the physical representation of a real-time entity and the EC compliant format. Crucial to the architecture is the Event layer, which uses Event Channels to propagate events through Regular Network infrastructures. The Event layer is realized by the COSMIC middleware, as described in the next section.

3 Interaction Model

3.1 General Information Flow

The flow of information (external environment and computational part) is seamlessly supported by the L-shaped architecture. It occurs in a number of different ways, which demonstrates the expressiveness of the model with regard to the necessary forms of information encountered in real-time cooperative and embedded systems.

Smart sensors produce events which report on the environment. Body sensors produce events which report on the body. They are disseminated by the local Event layer module, on an event channel (EC) propagated through the Regular Network, to any relevant remote Event layer modules where entities showed an interest on them, normally, sentient objects attached to the respective local Event layer modules. Sentient objects consume events they are interested in, process them, and produce other events. Some of these events are destined to other sentient objects. They are published on an EC using the same EC dialect that serves, e.g., sensor originated events. Smart actuators, on the other hand, merely consume events produced by sentient objects, whereby they accept and execute actuation commands.

The model offers opportunities to solve a long lasting problem in real-time, computer control, and embedded systems: the inconsistency between message passing and the feedback loop information flow subsystems.

3.2 Events in a real-time mobile scenario

An implementation of the event layer is provided by the **COSMIC (COoperating SMart devICes)** middleware. Events are disseminated in a publisher/subscriber style, which is particularly suitable because it supports generative, anonymous communication and does not create any artificial control dependencies between producers of information and the consumers. This decoupling in space (no references or names of senders or receivers are needed for communication) and the flow decoupling (no control transfer occurs with a data transfer) are crucial properties to maintain autonomy of components and dynamic interactions.

Differently from simple messages, an event includes the context in which it has been generated and quality attributes defining requirements for dissemination. This is particularly important in an open, dynamic environment where an event may travel over multiple networks. An event instance is specified as: $event := \langle subject, context\_attributeList, quality\_attributeList, contents \rangle$. A subject defines the type of event and thus is related to the event contents. It supports anonymous communication and is used to route an event. Attributes are complementary to the event contents. The context attributes describe the environment in which the event has been generated, e.g. a location, an operational mode or a time of occurrence. The quality attributes specify timeliness and dependability aspects in terms of $\langle validity\ interval, omission\ degree \rangle$ pairs. The validity interval defines the point in time after which an event becomes temporally inconsistent [5]. The temporal validity carried in the respective event attribute allows to define a transmission deadline in a synchronous network. If no transmission deadlines can be enforced in a network, it enables a consumer of an event to decide whether an event still is temporally consistent, i.e. represents a valid time-value entity.

Event channels in our system are abstractions of the underlying network. Publishers and subscribers interact via an unidirectional event channel by pushing events in the channel and receiving notifications. An event

channel is defined by: *event_channel* := ⟨*subject, quality_attributeList, handlers*⟩. The subject determines the event types which may be issued to the channel. In contrast to the attributes of an event, which describe the properties of a single individual occurrence of an event, the attributes of the event channel reflect the properties of the underlying communication network and dissemination scheme. These attributes include latency specifications, dissemination constraints and reliability parameters. Our goal is to handle the temporal specifications as ⟨*bound, coverage*⟩ pairs [8] orthogonal to the more technical questions of how to achieve a certain synchrony property of the dissemination infrastructure. A more detailed description of the event channels can be found in [3].

3.3 Synchrony Classes

The abstraction of event channels allows to specify channel attributes on the level of the event system and in terms of the application requirements. These channel properties have to be mapped to the protocols of the regular networks. Obviously, in a large scale WAN-of-CANs environment, an abstract network layer cannot provide the same quality of service for all message disseminations. Additionally, because harder guarantees require more resources, even on a CAN it is beneficial to distinguish messages with different dissemination requirements. Based on our previous work on predictable protocols for the CAN-Bus [6], we defined an abstract network layer to which the channel classes could be mapped straightforwardly. We distinguish three event channel classes according to their synchrony properties: hard real-time channels, soft real-time channels and non-real-time channels.

Hard real-time channels guarantee event propagation within the validity interval in the presence of a specified number of omission faults. We use a reservation scheme for hard real-time event channels and additionally exploit the CAN priority scheme. Events published in soft real-time channels are scheduled according to deadlines derived from their temporal validity as specified in the event attributes. Soft real-time event messages exploit the priority mechanism of the CAN-Bus [6]. However, deadlines cannot be guaranteed in overload situations. In these cases, the temporal validity of an event allows to detect a temporal inconsistency and provides awareness for the application. Non-real-time channels do not assume any temporal specification and disseminate events in a best effort manner.

The middleware to support dissemination of events is completely distributed. Because we aim at integrating smart components directly in the system, implementation must consider the respective performance and memory constraints. A first version of our COSMIC middleware (COoperating SMart devICes) has been implemented and is running on CAN-Bus networks connected to wireless TCP/IP networks. So far, this version of COSMIC enables seamless interaction via events in a WAN-of-CANs but does not yet support quality attributes of event channels. A new version which includes quality and context attributes has been implemented for RT-Linux. It is currently been tested and evaluated on a CAN-Bus before we will port it on the micro-controllers. Future work will include quality of service enforcement for channels which span multiple CANs.

4 Related work

Many event-based systems have been introduced for large systems, requiring quite complex infrastructures. These event systems do not consider stringent quality aspects like timeliness and dependability issues. Secondly, they are not created to support inter-operability between tiny smart devices with substantial resource constraints. In [1] a real time event system for CORBA has been introduced. The events are routed via a central event server which provides scheduling functions to support the real-time requirements. Such a central component is not available in an infrastructure envisaged in our system architecture and the developed middleware is a quite complex and unsuitable to directly integrating smart devices. The efforts to implement CORBA for the CAN-Bus suffer from lacking any support for timeliness. A new scheme to integrate smart devices in a CORBA environment is proposed in [4] and has lead to the proposal of a standard by the Object Management Group (OMG). In contrast to the event channel model introduced in this paper, all communication inside a cluster relies on a single technical solution of a synchronous communication channel. Secondly, although the temporal behaviour of a single cluster is rigorously defined, no model to specify temporal properties for cluster-to-CORBA or cluster-to-cluster interactions is provided.

References

[1] T. Harrison, D. Levine, and D. Schmidt. The design and performance of a real-time corba event service. In *Proceedings of the 1997 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 184–200, Atlanta, Georgia, USA, 1997. ACM Press.

[2] A. Hopper. The clifford paterson lecture, 1999 sentient computing. *Philosophical Transactions of the Royal Society London*, 358(1773):2349–2358, August 2000.

[3] J. Kaiser, C. Mitidieri, C. Brudna, and C. Pereira. COSMIC: A Middleware for Event-Based Interaction on CAN. In *Proc. 2003 IEEE Conference on Emerging Technologies and Factory Automation*, Lisbon, Portugal, September 2003.

[4] H. Kopetz, M. Holzmann, and W. Elmenreich. A Universal Smart Transducer Interface: TTP/A. *International Journal of Computer System, Science Engineering*, 16(2), March 2001.

[5] H. Kopetz and P. Verssimo. Real-time and Dependability Concepts. In S. J. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 16, pages 411–446. Addison-Wesley, 1993.

[6] M.A. Livani and J. Kaiser. Evaluation of a hybrid real-time bus scheduling mechanism for can. In *7th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'99)*, San Juan, Puerto Rico, April 1999.

[7] René Meier (Ed.). Preliminary definition of cortex programming model. CORTEX project, IST-2000-26031, Deliverable D2, March 2002.

[8] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *Transaction on Computers - Special Issue on Asynchronous Real-Time Systems*, 51(8):916–930, August 2002.

[9] P. Veríssimo and A. Casimiro. Event-driven support of real-time sentient objects. In *Proceedings of the 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, Guadalajara, Mexico, January 2003.