Supporting independent development, deployment and co-operation of autonomous objects in distributed control systems.

Jörg Kaiser, Sebastian Zug, Michael Schulze
Otto-von-Guericke-University Magdeburg, Germany
kaiser@ivs.cs.uni-magdeburg.de

Leandro Buss Becker
Federal University of Santa Catarina, Brazil
lbecker@das.ufsc.br

Abstract:

Autonomous, active components like smart sensors and actuators offer the capabilities of spontaneous behaviour, concurrent computations and well-defined communication interfaces. The perspective of building a system from such active blocks however has an impact on modelling and deployment of the components and supporting their interaction at run-time. The paper presents a modelling approach for dynamically interacting autonomous objects addressing the problems of independent development, deployment and incremental extension. Compared to related approaches our concept further complements the modelling by an adequate middleware that supports the abstractions of the model during run-time and performs the respective dynamic binding and co-operation between objects. This opens a wide range of possibilities for dynamically integrating hardware and software components and support virtual sensors and actuators.

Keywords: smart sensors, autonomous components, event-based system, distributed systems, object-oriented modelling, UML

1 Introduction

Autonomous vehicles like cars and mobile robots are composed from a large number of smart networked components comprising hardware, software and, sometimes, mechanical components. These autonomous building blocks are equipped with a computational core and may exhibit spontaneous behaviour. Rather than just being transducers in terms of raw physical data, they offer the capabilities of information processing entities, actively providing application-oriented information via a network interface. Fig. 1 shows one of our robots which is equipped with eight motors, gyros and acceleration sensors, a compass, odometry sensors and distance sensors all equipped with their own processor and connected to the popular automotive CAN-Bus [Bosch1991]. This forms the reactive system layer of the robot. Additionally, a more powerful computer performs the complex processing on the deliberative level and connects to a wireless network. In fact, the robot constitutes a distributed system and it would be highly desirable to purchase the components like navigation modules, environment perception equipment, or specific computing and signal processing engines from third party providers. Then, the programmer has just configuring an application along the information processing chain from sensors to actuators. The perspective of building a system from such active blocks however has a couple of consequences on modelling and deployment of the components and supporting their interaction at run-time.

Using active networked components has many benefits concerning the encapsulation of functional and temporal properties [Kop1998] [CKV2007]. At the interface of such a component a set of application specific data is available that is actively disseminated for further processing. All internal processes and computations are hidden. Also, the respective local control software usually may not be available as source code. Object-oriented modelling usually emphasizes a class hierarchy that supports design time issues. If third party components are used and source code is not available, this may be of limited value. The system development has to be based on the provided interfaces and the interactions between the components become the major concern. Secondly, the concurrent operation of smart components and the active nature of dissemination lead to a data centric view on the system as it has been suggested by the ADS approach [Mor1993].

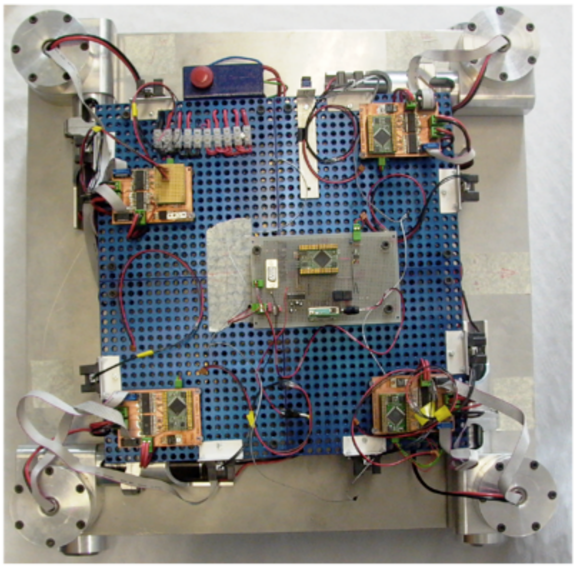


Fig.1: A network of smart components: The “Q” Robot

This means that computations are triggered by the availability of data rather than by an artificial superimposed control flow. Data flows through a chain of processing stages from the sensors to the actuators. This motivates a departure from the conventional synchronous invocation-based object interaction that basically is driven by a single control flow. A producer-consumer-based model that is oriented towards a distributed data flow model [KPB2001], [LeN2005] seems to be more appropriate. The concurrent operation of components corresponds to an actor model [Agh1986]. However, while actors communicate via messages addressing communication targets explicitly in a point-to-point fashion, the more data centric approach of ADS favours an anonymous communication in which all coordination only is performed via the shared data (the data field). The information is routed by contents or topic rather than by addresses. The advantage is that the binding between a producer of information and the respective consumer has not to be specified when the respective object is defined and can be modified without changing any object code. This supports independent development and dynamic deployment of components. If supported by adequate middleware, binding can be postponed to run-time and adds dynamic co-operation facilities to the system that enable an easy on-line modification and extension.

This paper addresses two challenges: Firstly, it describes how to extend object-oriented modelling by incorporating dynamic communication links relying on a subject-based publish subscribe scheme. Secondly, the paper shows how to exploit our COSMIC (CoOperating SMart devICes) publish-subscribe middleware to maintain the desirable property of independent component specification, implementation, deployment and co-operation at run-time. Finally, as a case study, we show how our approach can be exploited to integrate domain-specific tools like Simulink (Mathworks) or LabView (National Instruments).

2 Modelling a control application with autonomous components

Let us start with a simple example of a control system that may be used in a mobile robot application (Fig. 2). Distance sensors perceive the environmental conditions. Their outputs are sampled by a further component performing obstacle detection. Sensors determining and controlling the operation of the robot itself complement these environment perception components, like acceleration and odometry sensors that are fused to compute a reliable speed. A speed control component manages the drives based on the set point received by some command component and operational and environmental conditions.

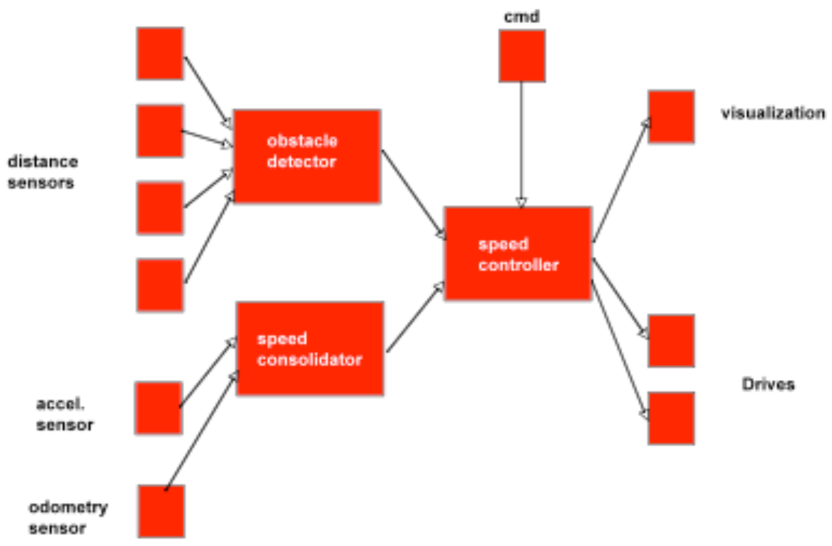


Fig.2: A simple control example of a mobile robot

The output of the speed controller may be visualized for remote inspection. We can observe an information flow from the sensors to the actuators in which each component may operate independently driven by the availability of information. Because the components are equipped with a processing element, they are able to perform their tasks concurrently. Designing a control program for such a robot often artificially superimposes a central sequential control structure as depicted in Fig.3.

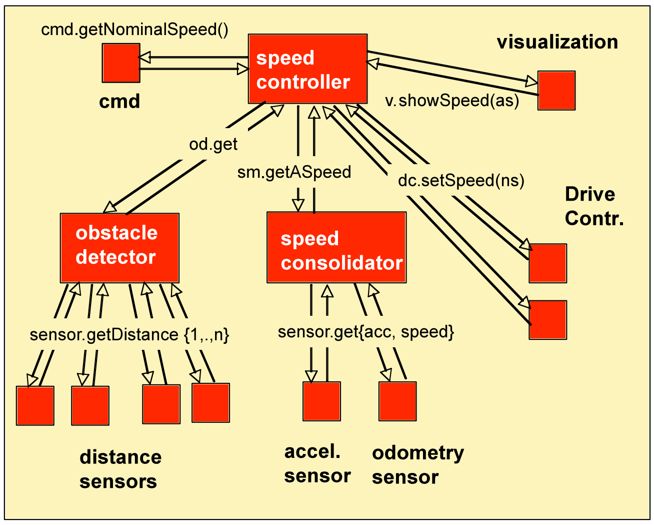


Fig. 3: Sequential control flow in a conventional control program

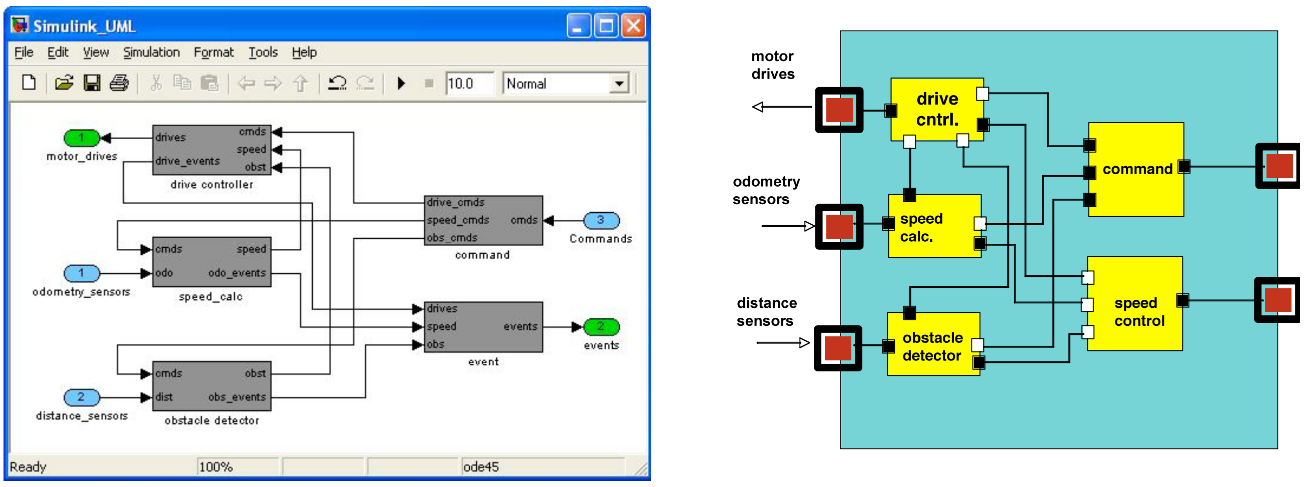
The C++ program that reflects the structure in Fig.3 would start with defining the motor controller operation. This includes collecting the required information to perform the task. The information is acquired by invoking the respective functions on the various components and would look like the program fragment presented in Fig.4. Essentially, we create a thread of sequential control flowing through the objects [LeN2005]. In a distributed control system this creates undesired control (flow-) dependencies because of the synchronous invocations [KPB2001], [EFG2001] and a centralized computational model. Even more important, specifying a control program in such a way will sacrifice all the inherent concurrency available in a distributed system architecture composed from these active entities. The data centric approach seems to be a more appropriate way of modelling interaction characterized by the flow of information though the systems from the sensors to the compute engines and actuators.

```
voidSpeedController::control() {
    ns = cmd.getNominalSpeed();
    if (od.get()) {
          ns=0;
    }
    as=sm.getASpeed();
    if (as != ns) {
          dc1.setSpeed(ns);
          dc2.setSpeed(ns);
    }
    v.showSpeed(as);
```

Fig. 4: C++ code of the sequential control flow control program

The components accept information at their interfaces, modify it and generate new information items for the downstream components in the chain. If we assume that the system is composed from independent networked components the inherent concurrency of operation is fully preserved. In general, we cannot assume that every component has its own processor and a component may be software-only. In the example, the obstacle detector and the speed consolidator are software components, which may or may not run on dedicated hardware. The middleware provides the encapsulation and a transparent communication model masking the details of deployment.

The popular domain specific modelling tools for control applications like Simulink and LabView are examples of how to specify such data-flow structures. Fig. 5 depicts the respective Simulink model for the presented example. The blocks correspond to the respective components of Fig. 1a while the connections are annotated with the data that is disseminated via these links. In the software engineering domain, a similar component-oriented modelling approach is available in the notations provided by ROOM [Sel1996] and RT-UML [SeR1998], [Sel1999] which allow specifying capsules, ports and protocols in a communication diagram. Fig 5b depicts the respective diagram describing the blocks and interactions of the example.



a.) b.)

Fig. 5: Modelling the example as a Simulink (a) and ROOM (b) diagrams

Domain specific languages and specification tools are very strong in defining the functionality of blocks. They offer a large spectrum of mathematical functions of all areas and also in providing an environment to simulate the specified design. Object-oriented modelling tools have the advantage to support the structural aspects of a design to enable automatic code generation, easy reuse and extension. Languages like Ptolemy [BLL2007] combine the advantages of both worlds and there are also other efforts to marry domain specific languages with UML [BON2007], [Shi2007]. However, all these approaches are mainly intended to run on a single machine. Usually, a single code module can be generated from a design and compiled to run on a powerful target machine. The mapping to a system structure consisting of independent tiny hardware/software components is not well supported. Although there is the possibility to distribute Simulink blocks on different machines and use explicit communication between these parts, all distribution is static and communication is message-based and low level only. Similarly, ports in UML are a step in the direction to postpone the binding decision between components. However, they do not adequately reflect the needs of a many-to-many communication and also are not intended to support run-time binding issues. Section 3 treats these problems in detail.

Our approach aims to provide a thin middleware layer that handles communication at a higher level and integrate the developed communication modules into the respective modelling and simulation environments. The proposed middleware supports a subject-based publish/subscribe scheme that allows specification of interaction patterns in terms of the exchanged data. This opens a number of desirable properties. Firstly, because communication is specified in terms of which data is needed to perform a certain function and which data is produced, no references have to be specified at design time and thus independent development of components is supported. This is particularly useful when thinking about hardware/software components supplied by a third party. Secondly, because communication is bound to content rather than based on addresses, items can be moved to other nodes without causing any change in the remaining system. This property is exploited to create mixed reality systems in which some sensor or actuator components may be simulated while others are real hardware components. It also allows distributing simulations to multiple nodes transparently. Obviously, this requires some middleware mechanism to perform the routing of a certain data item to the target dynamically. The important point here is that the application has not to be aware of these changes. These issues are handled by the COSMIC (CoOperating SMart devICes) middleware, which is described below.

Exploiting event-based middleware to shift configuration and binding issues to deployment time has been investigated in [EDS2004]. In this paper the focus is on factoring out QoS issues and provide separate descriptions that are used when deploying the software. As a middleware basis, Ciao (Component Integrated Ace Orb) is used that is a rather heavyweight middleware not suitable for tiny systems like smart sensors and actuators with a limited CPU performance and memory footprint. An approach that is very close to our goals and intentions is presented in the DRIVE tool [CCC2008]. In DRIVE the entire life cycle of a sensor/actuator application is supported including the run-time issues. The main difference firstly is that DRIVE is very much based on Java, which again is not suitable for small systems. Secondly, DRIVE is a unified, integrated system while we tried to incorporate many standard tools and run-time systems by just providing a thin layer of a unified interaction mechanism.

3 Computational model of the COSMIC middleware

COSMIC is based on two basic concepts, autonomous objects that are active entities performing all computations and events that constitute typed communication objects disseminating data [VCC2002], [CKV2007]. The term "event" is used to denote a typed communication object and expresses the publish/subscribe-based model of communication where nodes are notified when data is available. It does not refer to any synchrony model as e.g. in real-time systems [KoK1991] nor to a specific programming model as e.g. in TinyOS [HSW2000]. All interactions between autonomous objects are performed via typed communication objects called events. The main property of such a system is that communication relations are not determined by statically defined addresses but that conceptually, events are broadcasted and every sentient object locally decides to use an event based on its contents.

The benefit is that no communication structure is superimposed to the set of objects at design time but that every sentient object can autonomously and even dynamically decide which events to receive.

A very important aspect for this paper is the concept of event channels. An event channel guarantees dissemination properties. Because any form of predictable dissemination and QoS needs some form of resource reservation before communication, event channels are created prior to communication and setup the data structures and resource reservations according to their specification. An event channel is a unidirectional communication channel connecting multiple publishers to multiple subscribers. It is related to a specific class of events by a subject and only disseminates those events. COSMIC supports hard real-time (HRT), soft real-time (SRT) and non-real-time (NRT) channels [KBM2004] as abstraction of the physical network in terms of addressing mechanism and QoS properties. It has to be noted that the capacity of an event channel and its latency attributes, however, are dependent on the respective real network [KBM2004]. It may not always be possible to meet the desired QoS. This however will be recognized when a channel is bound to an underlying communication network. The necessary knowledge to detect these situations comes on the one hand from the channel specifications described in the next section and on the other hand from the knowledge about the properties of the underlying network. In a system that is statically configured and all network properties are known, it is easy to detect a mismatch. E.g. an emergency stop component may need distance information at precise time intervals. Thus the channel (to which the component will subscribe) is defined to be a hard real-time channel. If the network doesn't provide this quality, this is obviously detected. In a dynamic scenario the information about the network has to be kept and evaluated in some place. Gateways connecting heterogeneous networks would be an ideal place for this.

4 Modelling interactions by the notion of event channels

As stated in the previous sections, there are several modelling approaches that can represent components interactions, however none can handle properly the notion of event channels. For instance, the ROOM methodology defines ports, connectors, and protocols to specify the associations among autonomous objects/components. This notation is more recently incorporated by the version 2.0 of UML. Although these concepts suggest a solution to design independent objects with small control flow dependency, they do not support a completely anonymous event-based. Such kind of interaction, that is common in distributed applications, is only supported by what we call totally independent components.

Let us again use ROOM as example (considering that it is the same from UML 2.x). The first problem is that ROOM semantics represent an implicit client-server relationship (possibly bi-directional), which differs considerably from the uni-directional data-flow model (purely event-based) that is involved in this kind of control applications. Another problem not properly tackled by ROOM is the poor capacity to represent one-to-many communication, which is natural in event-based systems. Although ROOM allows the specification of ***replicated ports***, this, however, is simply a different way to represent multiple one-to-one connections. Finally, such as in any OO modelling/programming language, there is no facility to make the deployment of the model in a distributed environment[1]. Thereby, all bindings are defined at design-time. This is suitable in Ethernet-based networks, where nodes are accessed by address. However, this is a nightmare to be solved in broadcast networks, such as CAN-bus and wireless medium.

We can summarize the mentioned problems as follows: (i) there is no proper notation to represent a data-flow relationship (purely event-based) among distributed components; (ii)

[1] One could argue that UML offers the Deployment Diagram, but this is far to be a desirable solution (at most it serves as a sketch of the architecture).

there is no proper notation to represent one-to-many communications; (iii) there are no facilities to support runtime binding among interacting distributed components.

To solve these problems we propose that instead of modelling "relationships" among components, we explicit model data-flows. Our solution is based in first depicting all data exchanged among components in so called event channels. Afterwards it is modelled the data-flow itself, i.e., an explicit representation of which (single) component generates data and which (one or many) components consume these data. The graphical representation or our proposal can be synthesized in what we call an ***Event-Channel Diagram***, as shown in Fig.6. The elements in the middle represent the data exchanged among the independent components, which belong to event-channels. When a component is connected to the channel using a ***P*** tag, it means that it publishes data in the channel. On the other hand, an ***S*** tag means that the component subscribes for the channel data.

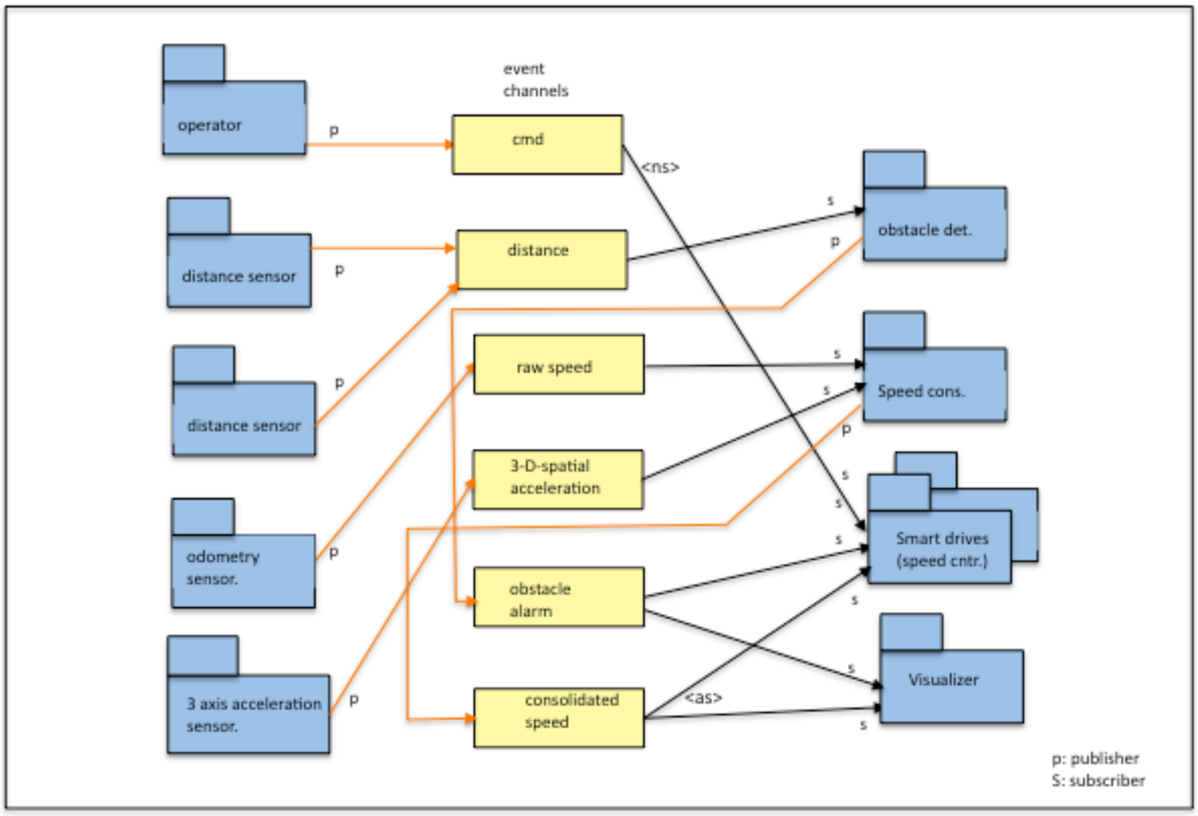


Fig.6: The proposed Event-Channel Diagram.

Using our Event-Channel Diagram one get the following benefits: (i) all interactions are purely event-based (there is no control flow exchange); (ii) it represents easily one-to-many communications; (iii) using COSMIC middleware, all bindings can be performed during runtime, regardless of the components location. When using the Event-Channel Diagram it is also possible to access the detailed features from channels and events. In Fig.7 we show these properties by taking as example the distance channel and distance message from Fig.6. Looking to the channel, the subject determines the event types which may be issued to the channel. Its attributes abstract the properties of the underlying communication network and dissemination scheme. These attributes include latency specifications, dissemination constraints, and reliability parameters. In respect to the event, the context_list describes the environment in which the event has been generated, e.g. a location, an operational mode or a time of occurrence. In this example this is the relative position of the sensor inside a vehicle (rel_pos) and a timestamp. The quality attributes specify the temporal properties of a single event in terms of a validity interval (temp_validity). The validity interval defines the point in time after which an event becomes temporally inconsistent [VR2001].

Event Channel (e.g. Distance)	Event (e.g. distance)

Ntw_interface: CAN-0
Subject: UID
QoSList: {hard_rt, reaction_time, omission_degree}
Handler: exec_h
	
	Subject: UID
	ContextList:{real_pos, time_stamp}
	QoSList: {max_rate, tmp_validy}
	Contents: distance

Fig.7: Detailed features from Channels and Events.

The main feature of the proposed approach relies on the capacity to provide an adequate automatic code generation, taking into consideration the information in the Event-Channel Diagram. Thereby, it is possible to create an application-skeleton that contains code to configure the channels and to make dynamic components bindings. This benefits directly the programmer, as she should worry only with the application itself, and not with the configuration code. Also this reflects in the quality features of the final code, improving features such as readability, maintainability, and portability. For example, the control code presented in Fig.4 would look much simpler if using the proposed scheme, as shown in Fig. 8. As it can be observed, this code is essentially event-based. It includes temporal events to trigger the periodic operations (ex. *controlPeriod()*)., and data events to trigger the aperiodic operations (ex. *refresh_as()*). The middleware is in charge of guaranteeing concurrency and mutual exclusion issues.

```
voidSpeedController::controlPeriod() {
    if (as != ns)
          publish(speedCh,ns);
    publish(visSpeedCh,as);     }

voidSpeedController::refresh_as(p_as) {
    as = p_as;                  }

voidSpeedController::refresh_ns(p_ns) {
    if(!od)  ns = p_ns;
    else ns = 0;            }
```

Fig. 8: C++ code of the modified data flow control program

5 Discussion

The Simulink example from section 2 (Fig. 5a) should highlight the advantages of our approach. Conventionally, the entire system described in Fig. 5a can be compiled to a monolithic code module for a target machine and executed there. However, we may want to distribute the load for the simulation on two machines. If so, we have to decide at design time which modules will go to which machine. Moreover, we have to use explicit low level (TCP/IP) messaging to distribute the data between modules. With the proposed event channel model and the support of the COSMIC middleware we can postpone the decision concerning the final deployment of modules. Because no addresses are used, it is completely transparent from the functional perspective on which machine the information for a module is generated and where the produced data have to be transferred. As a consequence, no change to the code of any module is necessary when some module is migrated or added. As mentioned before, it is not possible to have transparency for the temporal aspects of interaction. In our approach, the event channels as the mediators between the modules provide the possibility to express QoS attributes explicitly when modelling the application. According to these specifications, we can check on a mismatch of timing conditions. Another benefit comes for mixed reality scenarios where some sensors are real and some are just simulated. Simulated sensors and real sensors can be used and interchanged without changing anything in the software of those modules that subscribed to the respective events. Additionally, because we can assign simulated sensors freely to dedicated computational resources, we also may be able to achieve performance figures for a simulated scenario that is close to a scenario with real equipment. The overhead of using event channels is very small. Figure 9 shows the integration of event channels in a Simulink system. We added the operations to set-up a channel (announce), to publish and to subscribe. The "get event" function is provided because Simulink follows a strictly synchronous periodic temporal model while the autonomous objects may spontaneously generate events. Therefore, in a mixed application, the "get event" function checks periodically whether an asynchronous event has arrived and then notifies the respective (simulated) autonomous object. An example of this integration has been discussed in [KCS2008].

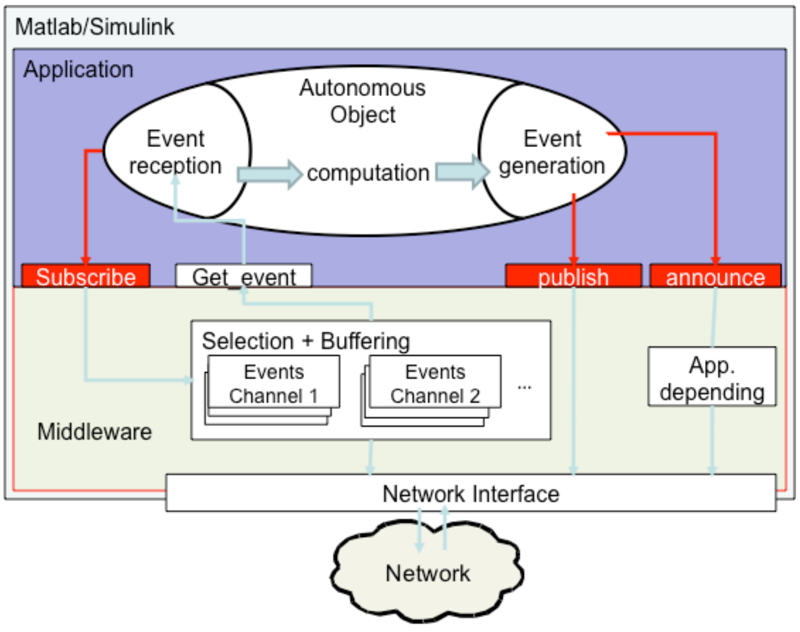


Fig. 9 Integration of the COSMIC middleware and Simulink

6. Conclusion

This work presented a contribution for the design of a distributed system by composing completely autonomous components, which interact by means of events. The core of our approach is the notion of Event-Channels, which can be modelled in a graphical tool using the Event-Channel Diagram. Moreover, the designed model can be used as input for an automatic code generator that can generate applications skeletons that already include all required configuration code. It is important to highlight that this code relies in the COSMIC middleware to provide the required application QoS. As a case study, we included event channels in Simulink, which enable dynamic interactions between Simulink functional blocks and other autonomous components in the network. This enables mixed reality scenarios in which simulated and real components interact.

As future work, we will address the adaptation of our proposed Event-Channel Diagram into UML models. In fact, this would not be modelled as a new UML diagram, but we will use the extensions mechanisms of UML (stereotypes, tag-values) to decorate existing diagrams with Event-Channels and Events information. Moreover, an adequate code generator will be provided. This should be done in the Eclipse platform, as it is a design platform that offers facilities to use the UML metamodel.

7 Acknowledgements

Part of the work has been supported by DAAD and GRICES in the German Portuguese collaboration Scheme (Project DECOMOR).

8 References:

[Agh1986] Gul Agha: “Actors: A model of Concurrent Computation”, MIT Press, 1986

[BLL2007] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Ptolemy II - heterogeneous concurrent modeling and design in Java. Technical Report UCB/EECS-2007-7, EECS Department, University of California, Berkeley, Jan 2007.

[BON2007] Lisane Brisolara, Marcio Oliveira, Francisco A. Nascimento, Luigi Carro, Flavio R. Wagner: Using UML as a front-end for an efficient Simulink-based multithread code generation targeting MPSoCs, DAC 2007 Workshop, UML-SoC 2007, San Diego Convention Center, San Diego CA, USA, June 3rd, 2007

[CCC2008] H. Chen, P.B. Chou, N. H. Cohen, S.S. Duri, C.W. Jung, “DRIVE: A tool for developing, deploying, and managing distributed sensor and actuator applications”, IBM System Journal, Vol. 47, No. 2, 2008-09-30

[CKV2007] António Casimiro, Jörg Kaiser, Paulo Verissimo, “Generic-Events Architecture: Integrating Real-World Aspects in Event-Based Systems”, Lecture Notes in Computer Science (Architecting Dependable Systems IV), Volume 4615/2007, Springer 2007, pp. 287-315

[EDS2004] G. T. Edwards, G. Deng, D. C. Schmidt, A. S. Gokhale, and B. Natarajan, ‘‘Model-driven Configuration and Deployment of Component Middleware Publish/Subscribe Services,’’ Proceedings of the 3rd ACM International Conference on Generative Programming and Component Engineering, Vancouver, ACM, New York (2004), pp. 337–360

[EFG2001] Eugster, P.T.,Felber, P., Guerraoui, R., Kermarrec, A.M., “ The many faces of publish/subscribe”, Technical Report DSC ID:200104, EPFL, Switzerland (2001)

[HSW2000] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, “System Architecture Directions for Networked Sensors”, ACM SIGPLAN Notices, Volume 35 , Issue 11, November 2000

[KBM2004] J. Kaiser, C. Brudna, C.Mitidieri, “Implementing Real-Time Event Channels on CAN-Bus, IEEE Workshop on Factory Communication Systems 2004 (WFCS'2004), Vienna, Austria, September 22-24, 2004

[KCS2008] J. Kaiser, M. Schulze, S. Zug, C. Cardeira, F. Carreira,, “Sentient Objects for Designing and Controlling Service Robots”, in 17th International Federation of Automatic Control World Congress (IFAC 2008), Seoul, Korea, 2008

[KoK1991] H. Kopetz, K.H. (Kane) Kim, “Real-Time Objects and Temporal Uncertainties in Distributed Computer Systems”, PDCS Technical Report Series, No. 42, January 1991

[Kop1998] H. Kopetz, “Component-based design of large distributed real-time systems”, Journal of IFAC, Pergamon Press, 1998, pp. 53-60

[KPB 2001] J. Kaiser, C.E.Pereira, L.B. Becker, C. Villela, C. Mitidieri, “On Evaluating Interaction and Communication Schemes for Automation Applications based on Real-Time Distributed Objects”, Proc. of the IEEE 4th International Symp. on Object-Oriented Real-Time Distributed Computing (ISORC 2001), Magdeburg, Germany, May 2001

[LeN2005] E. A. Lee and S. Neuendorffer, Concurrent Models of Computation for Embedded Software, IEEE Proceedings Computers and Digital Techniques 152, No. 2, 239-250 (2005).

[Mor1993] K. Mori, “Autonomous Decentralized System: Concept, Data Field Architecture and Future Trends”, Proc. of ISADS 1993, IEEE, Kawasaki, Japan, 1993, pp.28-34

[Shi 2007] Jianlin Shi: Model and Tool Integration in High Level Design of Embedded Systems, Phd Thesis, Department of Machine Design Royal Institute of Technology SE-100 44 Stockholm 2007

[Sel1996] B. Selic, “Real-Time Object-Oriented Modeling (ROOM)”, 2nd IEEE Real Time Technology and Applications Symposium, (RTAS '96), June 10-12, 1996, Boston, MA, USA

[SeR1998] B. Selic, J. Rumbaugh, “Using UML for Modelling Complex Real-Time Systems”, March 11, 1998, available from: http://www.ibm.com/developerworks/rational/library/139.html

[Sel1999] B. Selic, “Using UML in the Real-Time domain”, Communications of the ACM, Oktober 1999

[VCC2002] P.Verissimo, V.Cahil, A.Casimiro, K.Cheverst, A.Friday and J.Kaiser, “Cortex: Towards supporting autonomous and cooperating sentient entities”, In Proceedings of European Wireless 2002, Florence, Italy, Feb 2002.

[VR2001] Paulo Veríssimo, Luís Rodrigues, “Distributed Systems for System Architects”, Kluwer Academic Publishers, Boston, January 2001