Nicolas Guelfi
Gianna Reggio
Alexander Romanovsky (Eds.)

LNCS 3409

Scientific Engineering of Distributed Java Applications

4th International Workshop, FIDJI 2004
Luxembourg-Kirchberg, Luxembourg, November 2004
Revised Selected Papers

Springer

Nicolas Guelfi Gianna Reggio
Alexander Romanovsky (Eds.)

Scientific Engineering of Distributed Java Applications

4th International Workshop, FIDJI 2004
Luxembourg-Kirchberg, Luxembourg, November 24-25, 2004
Revised Selected Papers

Springer

Volume Editors

Nicolas Guelfi
University of Luxembourg
Faculty of Science, Technology and Communication
6, rue Richard Coudenhove-Kalergi, 1359 Luxembourg-Kirchberg, Luxembourg
E-mail: nicolas.guelfi@uni.lu

Gianna Reggio
University of Genoa
Department of Informatics
Via Dodecaneso 35, 16146 Genoa, Italy
E-mail: reggio@disi.unige.it

Alexander Romanovsky
University of Newcastle upon Tyne
School of Computing Science
Newcastle upon Tyne, NE1 7RU, UK
E-mail: alexander.romanovsky@ncl.ac.uk

Library of Congress Control Number: 2005921148

CR Subject Classification (1998): D.2, H.4, H.3, H.5.3-4, C.2.4, D.1.3

ISSN 0302-9743
ISBN 3-540-25053-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11398066 06/3142 5 4 3 2 1 0

Preface

FIDJI 2004 was an international forum for researchers and practitioners interested in the advances in, and applications of, software engineering for distributed application development. Concerning the technologies, the workshop focused on "Java-related" technologies. It was an opportunity to present and observe the latest research, results, and ideas in these areas.

All papers submitted to this workshop were reviewed by at least two members of the International Program Committee. Acceptance was based primarily on originality and contribution. We selected, for these post-workshop proceedings, 11 papers amongst 22 submitted, a tutorial and two keynotes.

FIDJI 2004 aimed at promoting a scientific approach to software engineering. The scope of the workshop included the following topics:

- design of distributed applications
- development methodologies for software and system engineering
- UML-based development methodologies
- development of reliable and secure distributed systems
- component-based development methodologies
- dependability support during system life cycle
- fault tolerance refinement, evolution and decomposition
- atomicity and exception handling in system development
- software architectures, frameworks and design patterns for developing distributed systems
- integration of formal techniques in the development process
- formal analysis and grounding of modelling notation and techniques (e.g., UML, metamodelling)
- supporting the security and dependability requirements of distributed applications in the development process
- distributed software inspection
- refactoring methods
- industrial and academic case studies
- development and analysis tools

The organization of such a workshop represents an important amount of work. We would like to acknowledge all the program committee members, all the additional referees, all the organization committee members, the University of Luxembourg, Faculty of Science, Technology and Communication administrative, scientific and technical staff, and the Henri-Tudor public research center.

FIDJI 2004 was mainly supported by the "Ministère de l'enseignement supérieur et de la recherche" and by the "Fond National pour la Recherche au Luxembourg."

November 2004

Nicolas Guelfi
Gianna Reggio
Alexander Romanovsky

Organization

FIDJI 2004 was organized by the University of Luxembourg, Faculty of Science, Technology and Communication.

Organizing Committee

Amza, Catalin	University of Luxembourg/DISI, Genoa, Italy
Berlizev, Andrey	University of Luxembourg, Luxembourg
Capozucca, Alfredo	University of Luxembourg, Luxembourg
Guelfi, Nicolas	University of Luxembourg, Luxembourg
Mammar, Amel	University of Luxembourg, Luxembourg
Perrouin, Gilles	University of Luxembourg, Luxembourg
Pruski, Cédric	University of Luxembourg, Luxembourg
Reggio, Gianna	DISI, Genoa, Italy
Ries, Angela	University of Luxembourg, Luxembourg
Ries, Benoît	University of Luxembourg, Luxembourg
Sterges, Paul	University of Luxembourg, Luxembourg

Additional Referees

Hnatkowska, Bogumila
Sterges, Paul

Sponsoring Institutions

uni.lu
UNIVERSITÉ DU LUXEMBOURG

fonds national de la recherche

This workshop was supported by the University of Luxembourg, the Ministry for Culture, Higher Education and Research, and the National Research Fund.

Table of Contents

Component-Based Design of Embedded Software: An Analysis of Design Issues

Christo Angelov, Krzysztof Sierszecki, and Nicolae Marian

Mads Clausen Institute for Product Innovation, University of Southern Denmark
Grundtvigs Alle 150, 6400 Soenderborg, Denmark
{angelov,ksi,nicolae}@mci.sdu.dk

Abstract. Widespread use of embedded systems mandates the use of industrial production methods featuring model-based design and repositories of prefabricated software components. The main problem that has to be addressed in this context is to systematically develop a software architecture (framework) for embedded applications, taking into account the true nature of embedded systems, which are predominantly real-time control and monitoring systems. There are a great number of design issues and unresolved problems with existing architectures, which have to be carefully analyzed in order to develop a viable component-based design method for embedded applications. Such an analysis is presented in this paper, which focuses on a number of key issues: specification of system structure; specification of system behaviour; component scheduling and execution; program generation vs. system configuration. The analysis has been used to formulate the guidelines used to develop *COMDES* – a software framework for distributed embedded applications.

1 Introduction

The widespread use of embedded systems (including time-critical and safety-critical systems) poses a serious challenge to software developers in view of diverse, severe and conflicting requirements, e.g. reduced development and operating costs and reduced time to market, as well as specific issues that are particularly important for embedded systems: dependable operation through reliable and error-free software; predictable and guaranteed behaviour under hard real-time constraints; open architecture supporting software reuse and reconfiguration; architectural support for software scalability, including both stand-alone and distributed applications.

The above requirements cannot be met by currently used software technology, which is largely based on informal design methods and manual coding techniques. Recently, there have been successful attempts to overcome the above problem through model-based design and computer-aided generation of embedded software from high-level specifications. However, this approach has a serious drawback: it does not provide adequate support for dynamic (in-site and on-line) reconfiguration since it requires the generation and compilation of new code, which has to be subsequently downloaded into the target system. The ultimate solution to the above problem can be characterized as computer-aided *configuration* of embedded software using formal frameworks and pre-fabricated *executable* components. The latter may be implemented as re-locatable silicon libraries stored in non-volatile memory (ROM).

The main problem to be solved in this context is to develop a comprehensive framework that would reflect the true nature of embedded systems, which are predominantly real-time control and monitoring systems [16]. Developing such a framework and the associated software design method is a highly complex engineering task, which is currently in the focus of attention of many research groups but so far, there has been no widely accepted solution [16-26]. This is due to a number of factors: very high complexity, great diversity of applications and the absence of common approach towards embedded software development, which is further aggravated by the lack of previous research [17].

There are a great number of design issues and unresolved problems with existing architectures, which have to be carefully analyzed in order to develop a viable component-based design method for embedded applications. Such an analysis is presented in this paper, which focuses on a number of key issues: specification of system structure (section 2); specification of system behaviour (section 3); component scheduling and execution (section 4); program generation vs. system configuration (section 5). The analysis carried out has been used to define guidelines used to develop *COMDES* – a software framework for distributed embedded systems [2-4] whose main features are summarized in section 6.

2 Specification of System Structure

A great number of embedded systems use a *process-based* configuration specification in the context of static and/or dynamic process scheduling. A process-based system is conceived as a set of interacting processes (tasks) running under a real-time kernel or a static schedule. Process-based specifications are criticized for emphasizing the functional rather than the structural decomposition of real-time systems. Such specifications address naturally the problems of scheduling and schedulability analysis but the resulting solutions are usually far from being open and easily reconfigurable (especially in the case of static process scheduling).

Conversely, *object-based* specifications emphasize structural decomposition, which facilitates the implementation of open and reconfigurable systems, e.g. industrial software standards such as IEC 61131-3 [12] and IEC 61499 [13]. In that case the system is conceived as a composition of interacting components, such as function blocks and port-based objects, which are then mapped onto real-real-time tasks [13, 20], or alternatively – executed under a static schedule [24, 26]. Unfortunately, object-based design methods often disregard the problems of timing behaviour and schedulability analysis, which are of paramount importance for hard real time systems (e.g. the above two standards). There are some notable exceptions, however, e.g. HRT-HOOD, which has a sound foundation in modern Fixed-Priority Scheduling Theory [5].

Object-based design uses a number of fundamental principles such as encapsulation, aggregation and association of objects (components). Therefore, it is inherent to component-based design methods. However, a major problem that has to be overcome is the informal and largely ad-hoc definition of application objects. This can be observed in many software design methods, where it is left to the system designer to define system objects and object classes for each particular application. That applies not only to component functionality and interfacing, but also – to the way components are mapped onto real-time tasks (e.g. one component mapped onto one task [21, 24], several components mapped onto one task [13, 20], several tasks mapped onto one component [8, 11]).

It can be argued that a software design method should incorporate both types of specification into a hierarchical model, taking into account both the structural and computational aspects of system architecture. This has resulted in the development of hybrid architectures and design methods such as ARTS [8] and SDL [11] that are both object and process-based. That is, the system is conceived as a composition of active objects, each of them encapsulating one or more threads of control. Threads invoke the operations of passive objects and the latter may invoke the operations of other objects, etc. This approach results in well-structured systems, featuring a well-specified hierarchy of active and passive objects, but once again, these are defined by the system designer for each particular case.

Ad-hoc specification and design severely limits component reusability. Therefore, it is necessary to develop a formal framework that will allow for a *systematic* specification of reconfigurable components, which will be reusable by definition. The proper way of doing this is to specify software components using decomposition criteria that are derived from the areas of control engineering and systems science rather than human experience and intuition, taking into account that modern embedded systems are predominantly control and monitoring systems [16].

This has been achieved to some extent in industrial software standards (e.g. those mentioned above) but at a relatively low level, i.e. the level of passive objects such as function blocks [12, 13]. However, no provision is made for reconfigurable state machines or hybrid models involving reconfigurable state machines and function blocks. Conversely, there are a few systems featuring reconfigurable state machines, e.g. AIRES [20] and StateWORKS [23], but they do not provide support for function blocks and function block diagrams in the sense of IEC 61131-3 and similar standards.

Component interaction is another major issue that has to be resolved while specifying system configuration. There are a number of interaction patterns, which are widely used in the context of both process-based and object-based systems: client-server, producer-consumer(s) and publisher-subscribers. Client-server is highly popular in the IT community and is also used in industrial computer systems and protocols. However, it has limitations, such as the blocking nature of communication and point-to-point interactions. Therefore, producer-consumer is considered better suited for real-time systems because it is non-blocking and supports one-to-many interactions [7]. Publisher-subscriber combines the useful features of the former two methods and it is also widely used in practice.

The above patterns can be implemented via component interfaces, which are defined in the component interface model. Embedded systems use basically two types of such models: a) the IEC 61131 type of model specifying component interfaces in terms of signal inputs and outputs, whereby a component output is directly linked to one or more inputs of other components (function blocks); b) port-based objects interacting via suitably linked input and output ports that are implemented as shared memory locations. The latter model has been specifically developed for robotics applications [21] but similar models have been widely used in other application domains as well, e.g. ROOM [10], SDL [11], SOFIA [22], PECOS [25], etc. Port-based objects provide a high degree of flexibility but at the same time their use may result in relatively complex models (diagram clutter) because it is necessary to explicitly specify all ports and port connections for a given application.

This problem is overcome in IEC 61131-like models through implicit definition of I/O buffers and softwiring of inputs and outputs resulting in simple and easy to understand models – function block diagrams [12]. In this case I/O buffers are defined within the corresponding execution records of interconnected function block instances, whereby an input variable value can be obtained from the corresponding output using either an I/O assignment statement, or even better – a pointer specifying the source of data for the corresponding input-to-output connection. However, this technique is directly applicable to low-level components such as function blocks but it does not scale up well to distributed applications. It the latter case, it is necessary to use special-purpose components, e.g. service interface function blocks [13], with the resulting loss of transparency.

Distributed applications require higher-level components (function units). These are software agents implementing autonomous subsystems, such as sensor, controller, actuator, etc., that are usually allocated to different network nodes and interact with one another within various types of distributed transactions. Therefore, the basic control engineering approach has to be extended into a *systems engineering approach,* in order to take into account the complexity of real-world applications. Accordingly, it is necessary to extend the softwiring technique, so that function units are connected with each other in a uniform manner and signals are exchanged *transparently* between them, independent of their physical allocation. This would require the development of a special-purpose protocol supporting the transparent exchange of signals, i.e. labeled messages, between interacting function units [1].

3 Specification of System Behaviour

At the operational level of specification, there is a controversy between various paradigms, e.g. event-driven vs. time-driven operation and control flow vs. data flow models. Consequently, some architectures and design methods emphasize event-driven reactive behavior and control flow, whereas others focus on time-driven operation and the data flow between interacting components and subsystems. The former type of software architecture is usually associated with discontinuous event-driven systems, whereas the latter is preferred with continuous control systems. This situation reflects a gap between continuous and discontinuous systems modeling and design (e.g. state machines vs. data flow diagrams), which has been recognized by the control engineering community.

However, such a differentiation of system models is largely artificial and it clearly comes into conflict with the nature of real plants, which are more or less hybrid, even if they are treated as predominantly discrete or continuous. That is even more obvious in the case of complex hybrid control systems and applications. On the other hand, it can be shown that a sequential control system can be represented with a function block diagram (in terms of gates and flip-flops) and likewise – a continuous control system can be entirely specified in terms of state machines, even though these are not perhaps the typical modeling techniques for the above two types of system.

Similarly, a discontinuous event-driven system can be specified and implemented as a time-driven synchronous state machine; on the other hand, time-driven behaviour can be considered a special case of event-driven behaviour, where periodic activities are triggered by regularly arriving timing events, e.g. *timeout* events as defined in Statecharts.

Synchronous time-driven implementation is actually preferred by real-time engineers, because such systems have periodic task execution patterns, and this is a major prerequisite for the estimation of task response times using analysis techniques developed in modern Real-Time Scheduling Theory. It is worth noting that synchronous time-driven systems are preferred not only by real-time engineers but also, by control engineers: most practical examples of industrial control systems – both continuous and sequential – are synchronous systems triggered by periodically arriving timing events.

That is also the choice of hardware engineers, who implement sequential circuits as synchronous (clock-driven) state machines. However, this example prompts another interesting observation: hardware systems generate signals (i.e. reactions to timing events) with a very small delay, which can be ignored for the purpose of analysis. That is, it is possible to assume zero delay between the clock event and the corresponding reaction. This is the well-known *synchrony hypothesis*, which has been also adopted for a class of real time systems in a more general event-driven context, assuming that events are clocked and subsequently processed by the system before the next tick arrives. There are a number of architectures and programming languages illustrating this approach, e.g. real-time languages such as ESTEREL, LUSTRE, and SIGNAL [9]. However, these languages use an interleaved model of execution, whereby concurrent processes (state machines) are compiled into sequential programs with the resulting loss of modularity [18].

The above discussion has outlined the duality of various paradigms used to specify system behaviour. Nonetheless, there is seemingly no model combining naturally the reactive and the transformational aspects of system behavior, both of which are inherent to real-world systems and especially – to complex hybrid control systems. There have been attempts to solve this problem, and most notably the MoBIES project being developed at the University of California, Berkeley [18]. That project combines numerous computational models into a hybrid object-oriented framework, whereby the notion of computational model includes not only "pure" operational models such as state machines and data flow diagrams, but also – process interaction and execution models. Unfortunately, this has resulted in an overly complex framework featuring multiple operational domains and "polymorphic" component interfaces, which seems to be too complicated for practical purposes. This framework is supported by an equally complex software engineering environment featuring a multi-stage program generation process.

Another attempt to bridge the gap between event-driven and time-driven behaviour and accordingly – between control flow and data flow, is illustrated with the component model introduced in standard IEC 61499 [13]. In that case reactive (event-driven) and data transformation aspects are combined into low-level components such as function blocks. Whereas this is a big step forward in comparison with the previous standard IEC 61131-3 [12], it has also substantial limitations: for each function block, event-driven behaviour is specified in terms of a limited subset of input and output events that are defined in the interface specification of the function block. Accordingly, event-driven behaviour is specified with a state transition graph, which is "hardwired" in the function block. Hence, it is impossible to reconfigure the state machine, without re-designing and re-implementing the function block.

This limits substantially component reusability, since in the general case a function block may be expected to execute in different contexts, e.g. operational modes of a complex modal controller whose behaviour is specified with a state transition graph that is different from the one encoded in the function block. On the other hand, it might be possible to execute a sequence of function blocks within a given mode of operation. Hence, it is not necessary to replicate the state transition graph in all function blocks involved, which could result in undue overhead.

Instead, state transition logic might be implemented within a higher level of abstraction, i.e. a state machine that would be capable of executing function blocks and/or function block sequences (function block diagrams) within different states/modes of operation. This is essentially a hybrid state machine, which combines in a natural way the reactive and transformational aspects of component behaviour. It can be eventually encapsulated into a function block of class *reconfigurable state machine*, which can be used to implement complex behaviour for a broad range of embedded applications.

4 Component Scheduling and Execution

Component operations are mapped onto real-time processes (tasks) that have to be executed concurrently in a multi-tasking and possibly – multiple-node distributed environment. This is related to another aspect of operational behaviour, i.e. process scheduling and execution, which has to be provided by some kind of operational environment, guaranteeing that processes are executed within specified deadlines. This problem is further complicated when processes are executed as integral part of (possibly complex) sequences – transactions, which have to satisfy the corresponding end-to-end deadlines. It becomes even more complex in the case of distributed transactions, where computational and communication tasks have to be executed in different scheduling domains – network nodes, communication media, etc., observing once again the corresponding end-to-end deadlines. However, in all cases the adopted scheduling mechanism has to provide a safe operational environment for application tasks, i.e. predictable and guaranteed behaviour under hard real-time constraints.

There are basically two approaches to process scheduling for dependable embedded systems: *static scheduling* vs. predictable *dynamic scheduling* using algorithms developed in modern Real-Time Scheduling Theory. Static scheduling is widely used with dependable real-time systems in application areas such as aerospace and military systems, automotive applications, etc., and it is illustrated with the timed-triggered architecture specifically developed for this type of system [7]. It has also been used with a number of component-based design methods as well, see e.g. [24, 26]. However, this approach has a major disadvantage: its use results in closed systems that are