João M. P. Cardoso · Michael Hübner
Editors

Reconfigurable Computing

From FPGAs to Hardware/
Software Codesign

Springer

João M.P. Cardoso • Michael Hübner
Editors

Reconfigurable Computing

From FPGAs to
Hardware/Software Codesign

Springer

Editors

João M.P. Cardoso
Departamento de Engenharia Informática
Faculdade de Engenharia (FEUP)
Universidade do Porto
Rua Dr. Roberto Frias, s/n,
4200–465 Porto, Portugal
jmpc@acm.org

Michael Hübner
Institut für Technik der
Informationsverarbeitung, Fakultät für
Elektrotechnik und Informationstechnik
Karlsruher Institut für Technologie (KIT)
Kaiserstr. 12
Karlsruhe, Germany
michael.huebner@kit.edu

ISBN 978-1-4614-0060-8 e-ISBN 978-1-4614-0061-5
DOI 10.1007/978-1-4614-0061-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011933471

Preface

Dr. Panagiotis Tsarchopoulos

The objective of the European research programme in Information and Communication Technologies (ICT) is to improve the competitiveness of European industry and enable Europe to master and shape future developments in ICT. ICT is at the very core of the knowledge based society. EU research funding has as target to strengthen Europe's scientific and technology base and to ensure European leadership in ICT, help drive and stimulate product, service and process innovation and creativity through ICT use and value creation in Europe, and ensure that ICT progress is rapidly transformed into benefits for Europe's citizens, businesses, industry and governments.

Over the past years, the European Commission has constantly increased the amount of funding going to research in computing architectures and tools through the European research programme in Information and Communication Technologies. In this context, the European Commission has funded a number of European research projects in the area of reconfigurable computing. Results from these projects are being presented in this book providing a valuable reference point, which describes the efforts of several international research teams.

Reconfigurable Computing is a fascinating alternative to mainstream computing. But is it always going to remain just an alternative occupying a market niche? The potential for reconfigurable computing has not yet been fully unleashed although there have been notable successes – mostly for 'fine-grain' reconfigurability. Now there are technological developments and market opportunities that suggest breakthroughs in the future for coarse-grain reconfigurability – a field in which Europe has particular strengths. As a matter of fact, the 'coarse-grain' market is showing increasing potential. Tile-based architectures, for example, offer a balance of flexibility and ease of programming, drawing on libraries of pre-defined functionality. Europe has a considerable track-record in research into coarse-grained reconfigurability, and this offers a re-entry route for Europe into the Reconfigurable Computing market, based upon an integrated approach of hardware together with development systems for specific application domains.

In reconfigurable computing, one important observation – that is also becoming reality in mainstream computing with the advent of multicore architectures – is that parallelism is omnipresent. Most reconfigurable computing exploits the potential for parallel processing as much as possible using different "flavours" of parallelism. The exploding interest in parallelism presents another opportunity for reconfigurable computing.

If it is to be effective, any European strategy for RTD in Reconfigurable Computing *must* be set in the context of its potential use by applications developers and systems designers. It must take account of the market – the market for supply of Reconfigurable Computing technologies; the evolution of the general purpose computing market; and the markets of the users. And for embedded systems applications, it must also take account of the evolution of the methodologies ad requirements of the users. Technology is not enough.

The markets for Reconfigurable Computing may be divided into two – High Performance Computing and Embedded Systems. These markets – and the technological solutions appropriate to them – are quite different. However, they share one very important property: the fundamental obstacle to take-up of Reconfigurable Computing is the difficulty of programming. While localised solutions might be devised for specific technologies, such solutions are generally not viable, given their limited markets.

The highest priority need for RTD is therefore to enable commercially viable programmability of Reconfigurable Computing technology. This requires coherent, integrated (or "integrable") suites of processes, methods and tools spanning:

- application level support for reconfigurability that supplements existing design methodologies, including support for verification and validation of reconfigurable behaviour and reconfigurability properties of the system so as to satisfy qualification requirements;
- mapping from the output of application design to reconfigurable hardware via intermediate layer(s) of abstraction with standard libraries of functions based on open and widely accepted standards; and
- run-time support for reconfiguration, typically through OS extensions for resource allocation, scheduling, and discovery; debugging and monitoring; and fast re-layout of reconfigurable units.

Future European RTD in these topics must recognise the need for compatibility with development paradigms and processes, methods and tools in the applications sectors. Indeed, RTD in Reconfigurable Computing should be application-driven. Application sectors where Europe could gain particular advantage include embedded healthcare, (multi)physical system modeling, biomedical, cognitive radio, portable consumer devices, automotive/avionics, infotainment, and user-driven reconfigurable products.

The book that you have in your hands will give you a glimpse of the future: research results that will be coming out of labs towards market introduction; unresolved issues and new research challenges that need to be solved; relentless efforts to produce the last missing piece of magic that will make everything work…. but above all, I am sure, you will feel the enthusiasm and passion of the researchers and engineers that make all this happen.

Dr. Panagiotis Tsarchopoulos
ICT Research Programme
European Commission

Disclaimer: The views expressed are those of the author and do not necessarily represent the official view of the European Commission on the subject.

Contents

Contributors

Tapani Ahonen Tampere University of Technology, Tampere, Finland

José Carlos Alves Departamento de Engenharia Electrótecnica, Faculdade de Engenharia (FEUP), Universidade do Porto, Porto, Portugal

Michel Auguin CNRS, Orsay, France

Juergen Becker Institut fur Technik der Informationsverarbeitung, Fakultat fur Elektrotechnik und Informationstechnik, Karlsruhe Institute für Technology, Karlsruhe, Germany

Jürgen Becker Institut für Technik in der Informationsverarbeitung (ITIV), Karlsruhe Institute of Technology KIT, Karlsruhe, Germany

Koen Bertels Computer Engineering Lab, Faculty Electrical Engineering, Mathematics and Computer Science, Technische Universiteit Delft, TUD, Delft, The Netherlands

Labros Bisdounis INTRACOM, Athens, Greece, (currently: T.E.I. of Patras, Greece)

Umberto Bondi Università della Svizzera italiana, Lugano, Switzerland

Philippe Bonnot Thales Research & Technology, Paris, France

Timon D. ter Braak University of Twente, Enschede, The Netherlands

Lars Braun Institut für Technik der Informationsverarbeitung, Fakultät für Elektrotechnik und Informationstechnik, Karlsruher Institut für Technologie, (KIT), Karlsruhe, Germany

Paul Brelet Thales Research & Technology, Paris, France

Stephen T. Burgess Tampere University of Technology, Tampere, Finland

Marcel van de Burgwal University of Twente, Computer Science, Enschede, The Netherlands

Joan Cabestany Universitat Politècnica de Catalunya, Catalonia, Spain

Fabio Campi STMicroelectronics SRL, Agrate Brianza, Italy

João M.P. Cardoso Departamento de Engenharia Informática, Faculdade de Engenharia (FEUP), Universidade do Porto, Porto, Portugal

Luigi Carro Universidade do Rio Grande do Sul, Passo Fundo, Brazil

Marcelo Cintra University of Edinburgh, Edinburgh, UK

George A. Constantinides Department of Electrical & Electronic Engineering, Imperial College London, London, UK

José Gabriel F. de Coutinho Department of Computing, Imperial College London, London, UK

Martin Danek UTIA AV CR, Ostrava, Czech Republic

Giuseppe Desoli ST Microelectronics, Agrate Brianza, Italy

Jean-Philippe Diguet CNRS, Orsay, France

Pedro C. Diniz Electronic Systems Design and Automation Research Group, INESC-ID, Lisboa, Portugal

Susan Eisenbach Imperial College, London, UK

Fabrizio Ferrandi Dipartimento di Elettronica e Informazione Politechnico di Milano, Milano, Italy

Alberto Ferrante Università della Svizzera italiana, Lugano, Switzerland

João Canas Ferreira Departamento de Engenharia Electrótecnica, Faculdade de Engenharia (FEUP), Universidade do Porto, Porto, Portugal

Paolo Gai Evidence, Edinburgh, Italy

Christian Gamrat CEA, LIST, Centre de Saclay - Point Courrier 94, Gif sur Yvette Cedex, France

Georgi N. Gaydadjiev Computer Engineering, TU Delft, The Netherlands

Richard Geißler Atmel Automotive GmbH, Heilbronn, Germany

Roberto Giorgi Universita' degli Studi di Siena, Siena, Italy

Fernando Gonçalves Coreworks – Projectos de Circuitos e Sistemas Electrónicos S.A., CW, Porto, Lisboa, Portugal

Kim Grüttner OFFIS – Institute for Information Technology, Oldenburg, Germany

Arnaud Grasset Thales Research & Technology Campus Polytechnique1, Palaiseau Cedex, France

Reiner Hartenstein Fachbereich Informatik,
Technische Universität Kaiserslautern, Baden-Baden, Germany

Philipp A. Hartmann OFFIS – Institute for Information Technology, Oldenburg, Germany

Andreas Herrholz OFFIS – Institute for Information Technology, Oldenburg, Germany

Paul M. Heysters Recore Systems, Enschede, The Netherlands

Michael Hübner Institut für Technik der Informationsverarbeitung,
Fakultät für Elektrotechnik und Informationstechnik, Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany

Heikki Hurskainen Tampere University of Technology, Tampere, Finland

Chris Jesshope University of Amsterdam, Amsterdam, The Netherlands

Jiri Kadlec UTIA AV CR, Ostrava, Czech Republic

Stefanos Kaxiras Industrial Systems Institute, Patras, Greece

Hans G. Kerkhoff University of Twente, Enschede, The Netherlands

André B. J. Kokkeler University of Twente, Enschede, The Netherlands

Ralf König Institut für Technik in der Informationsverarbeitung (ITIV),
Karlsruhe Institute of Technology KIT, Karlsruhe, Germany

Kamil Krátký Advanced Technology Europe, Honeywell International,
Brno, Czech Republic

Matthias Kühnle Institut für Technik der Informationsverarbeitung,
Fakultät für Elektrotechnik und Informationstechnik,
Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany

Georgi Kuzmanov Computer Engineering Lab, Faculty Electrical Engineering, Mathematics and Computer Science, Technische Universiteit Delft, TUD, Delft, The Netherlands

Eric Lenormand THALES, Paris, France

Wayne Luk Department of Computing, Imperial College London,
London, UK

Giacomo Marchiori Dipartimento di Fisica Università di Ferrara, Ferrara, Italy

Debora Motta Matos Universidade do Rio Grande do Sul, Passo Fundo, Brazil

Sally A. Mckee Chalmers University, Gothenburg, Sweden

Philippe Millet Thales Research & Technology, Paris, France

Razvan Nane Computer Engineering Lab, Faculty Electrical Engineering, Mathematics and Computer Science, Technische Universiteit Delft, TUD, Delft, The Netherlands

Jari Nurmi Tampere University of Technology, Tampere, Finland

Bryan Olivier ACE Associated Compiler Experts b.v., Amsterdam, The Netherlands

Frank Oppenheimer OFFIS – Institute for Information Technology, Oldenburg, Germany

Juha Pärsinnen VTT, Espoo, Finland

Zlatko Petrov Advanced Technology Europe, Honeywell International, Brno, Czech Republic

Jean-Marc Philippe CEA, LIST, Paris, France

Wolfram Putzke-Roeming Deutsche Thomson OHG, Hanover, Germany

Nikola Puzovic Universita' degli Studi di Siena, Siena, Italy

Jussi Raasakka Tampere University of Technology, Tampere, Finland

Gerard Rauwerda Recore Systems, Enschede, The Netherlands

Jean-Luc Roux ACIES, Paris, France

Mateus Rutzig Universidade do Rio Grande do Sul, Passo Fundo, Brazil

Axel Schneider Alcatel-Lucent Deutschland AG, Stuttgart, Germany

Alex Shafarenko University of Hertfordshire, Hatfield, UK

Eberhard Schüler PACT XPP Technologies AG, Munich, Los Gatos, Germany

Vlad Mihai Sima Computer Engineering Lab, Faculty Electrical Engineering, Mathematics and Computer Science, Technische Universiteit Delft, TUD, Delft, The Netherlands

Gerard J.M. Smit University of Twente, Computer Science, Enschede, The Netherlands

Hans van Someren ACE Associated Compiler Experts b.v., Amsterdam, The Netherlands

Ioannis Sourdis Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden

Kim Sunesen Recore Systems, Enschede, The Netherlands

Benoit Tain CEA, LIST, Paris, France

Florian Thoma Institut für Technik der Informationsverarbeitung, Fakultät für Elektrotechnik und Informationstechnik, Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany

Raffaele Tripiccione Dipartimento di Fisica Università di Ferrara, Ferrara, Italy

Bart Vermeulen NXP Semiconductors, Eindhoven, The Netherlands

Nikolaos S. Voros Department of Telecommunication Systems & Networks, Technological Educational Institute of Mesolonghi, Mesolonghi, Greece

Stephan Wong Technische Universiteit Delft, Delft, The Netherlands

Ayal Zaks IBM, Haifa, Israel

Xiao Zhang University of Twente, Enschede, The Netherlands

Henk van Zonneveld Thales Netherlands, Hengelo, The Netherlands

Chapter 7
AETHER: Self-Adaptive Networked Entities: Autonomous Computing Elements for Future Pervasive Applications and Technologies

Christian Gamrat, Jean-Marc Philippe, Chris Jesshope, Alex Shafarenko, Labros Bisdounis, Umberto Bondi, Alberto Ferrante, Joan Cabestany, Michael Hübner, Juha Pärsinnen, Jiri Kadlec, Martin Danek, Benoit Tain, Susan Eisenbach, Michel Auguin, Jean-Philippe Diguet, Eric Lenormand, and Jean-Luc Roux

Abstract The ÆTHER project has laid the foundation of a complete new framework for designing and programming computing resources that live in changing environments and need to re-configure their objectives in a dynamic way. This chapter contributes to a strategic research agenda in the field of self-adaptive computing systems. It brings inputs to the reconfigurable hardware community and proposes directions to go for reconfigurable hardware and research on self-adaptive computing; it tries to identify some of the most promising future technologies for reconfiguration, while pointing out the main foreseen Challenges for reconfigurable hardware. This chapter presents the main solutions the ÆTHER project proposed for some of the major concerns in trying to engineer a self-adaptive computing system. The text exposes the ÆTHER vision of self-adaptation and its requirements. It describes and discusses the proposed solutions for tackling self-adaptivity at the various levels of abstractions. It exposes how the developed technologies could be put together in a real methodology and how self-adaptation could then be used in potential applications. Finally and based on lessons learned from ÆTHER, we discuss open issues and research opportunities and put those in perspective along other investigations and roadmaps.

7.1 Project Partners

1. CEA, LIST, France (coordinator)
2. University of Amsterdam, The Netherlands
3. University of Hertfordshire, UK
4. University of Karlsruhe – Karlsruhe Institute of Technology, Germany

C. Gamrat (✉)
CEA, LIST, Centre de Saclay - Point Courrier 94,
F-91191 Gif sur Yvette Cedex, France
e-mail: christian.gamrat@cea.fr

5. Università della Svizzera italiana, Switzerland
6. Imperial College, UK
7. VTT, Finland
8. Universitat Politècnica de Catalunya, Spain
9. UTIA AV CR, Czech Republic
10. CNRS, France
11. INTRACOM, Greece
12. THALES, France
13. ACIES, France

- Project Coordinator: Christian Gamrat, CEA, LIST, France
- Start Date: 2006-01-01
- Expected End Date: 2009-06-30
- EU Program: 6th Framework Programme, FP6-2004-IST-4, FET-ACA Project No. 027611
- Global Budget: 6 M €
- Global Funding by EU: 4 M €
- Contact Author: Jean-Marc Philippe, jean-marc.philippe@cea.fr

7.2 Introduction

A few decades ago, a computer was a very big, expensive and intriguing machine for much of the people. It was thought by many as something that would probably never cross their daily life. It was a time when computers were rare and enormous machines surrounded by millions of intrigued people. Nowadays the picture has completely changed and we find people surrounded by dozens of computers, many of them sitting hidden in the most awkward places. At the same time, we have witnessed a silent change in the way most computing resources are being used: instead of having one mainframe computer used by hundreds of persons, one can see now each person using at the same time multiple computing resources. They can be found everywhere running practical applications in our mobile phones, our TV sets, our cars, our houses, in the streets of our towns, etc. All those embedded computing resources are increasingly heterogeneous and increasingly interconnected and build up very complex intricate computing mesh. Indeed, many modern applications do not rely on a single computing resource but rather on a group of various computing elements. And the situation is bound to be even more complex in the future as new applications needs and new technology emerge.

In the context of applications running on vast amount of heterogeneous and potentially volatile computing resources, the tasks of designing, programming, optimizing and managing systems are key issues. When seeking solutions for those problems, one very straightforward observation comes to mind: it would be far easier for the application designer if each part of the system could embed enough intelligence and independence so that they could locally and dynamically optimize their resources and the way they perform tasks according to the function they have to perform. In such a system, the application designer could focus on the topic he knows best: the application. There is certainly more in the idea of self-adaptive computing than the fact of helping the designer, even though this is an important reason for its use in implementing complex and heterogeneous systems. The computing system would therefore act like an army of tiny assistants (or agents) not only executing a given program but also assisting the designer to dynamically fine-tune the program according to implementation specific details. This intuitive observation has been the root idea envisioned when preparing the ÆTHER project.

The fact that current computing techniques will not be able to cope with the rising complexity of future applications and architectures has long been identified by research groups. One of the most notable efforts has been put forward by the IBM Autonomic Computing Research initiative [1]. Even before this, authors who laid down the foundation of modern computing [2] were aware that building the ideal computer system should involve a degree of autonomy in order to deal with the computing complexity of high-level tasks. High level tasks were then considered as those trying to mimic the human intelligence and cognitive processes. The rationale was then straightforward: an ideal electronic computer (electronic brain) should behave like a biological brain exhibiting a level of autonomy, self-healing and self-control. In the meantime, the fantastic evolution of microelectronics boosted by Moore's law [3] has led to multi-billion transistors chips that can implement hundreds of microprocessors in a typical computing system. At the same time, the complexity of the software layers has dramatically increased, leading to systems that become very difficult to design and operate.

Being very aware of the intricacy of the various levels of a computing system, the ÆTHER consortium embarked on the challenging path of pluri-disciplinary co-engineering three of the main aspects of self-adaptive computing: software programming, run-time system and hardware implementation. This document presents the solutions we came about to address some of the major concerns of building a self-adaptive computing system. After exposing our vision of self-adaptation and its requirements, we describe and discuss the solutions we put forward for tackling self-adaptivity at the various levels of abstraction. We then present how the proposed solutions could be put together in a real methodology and how self-adaptation could then be used in real applications. Finally and based on lessons learned from ÆTHER, we discuss open issues and research opportunities and put those in perspective along other investigations and roadmaps.

7.3 Self-Adaptation

Self-adaptation can be defined as the ability of a system to react to external changes by modifying its behavior based on a local policy. This definition introduces two important concepts:

- Obviously, such a system first needs to exhibit adaptability. Adaptability is the result of a **reactive process** modeled by a feedback loop. **Loops** (for short) are the basic objects of any adaptation process and as such shall be the basic objects to deal with. And **loops are everywhere around us**!
- Secondly there is the notion of **Self**. In the above simplistic definition, "Self" is tightly related to the "local policy". In its simplest form the "Self" of a system is entirely described by its local policy. "Self" is the local behavior of the system, it is the knowledge that needs to be maintained locally and that provides the autonomous (or local) behavior.

Therefore, Self-Adaptation is not only about adaptive loops. Self-adaptation means loops plus knowledge and this was well described by Kephart and Chess [4] of IBM Autonomic Computing research group. A lot can be said on the relevance of this simple definition on computing systems as it implies various notions:

- What does the system need to observe in order to trigger adaptation? What is an observer?
- What is the mechanism allowing the adaptation process? Is it parametric or structural?
- What kind of rule or policy should be applied to the system so that it can adapt?
- What level of supervision is needed? Can the process be completely autonomous?

In the simplest adaptive system, a closed loop allows for controlling a process based on the observation of how the process performs against a given performance target. The target or objective is generally set or computed by an external mean. Example of such adaptive systems includes the dynamic regulators that can be designed using analogue or digital techniques for specific tasks: temperature control, speed regulation, etc. Such systems are generally designed in a specific fashion for the task to be realized. Sensors and actuators play a key role in their implementation. In a self-adaptive system, the target objective can be adjusted locally based on the context and/or the observation of various parameters. Let us look at a simple example. A fairly basic application is one of a system that dynamically adapts the temperature of a device. In its simplest form, the circuit depicted in Fig. 7.1 will continuously control the temperature of the object by applying a corrective action on the actuator directly computed from the difference between the measured and target temperatures. Such a simple setup is alright if the system is in a known context, the components do not change much and finally the target can be expressed in a very accurate way like a numerical temperature value. Now what happens if something changes in the system environment so that the statically designed control loop can no longer achieve its goal?

In the simple adaptive loop, the best that can be done is probably to detect that the system is entering a non-optimal scenario by monitoring the closed loop error and identifying that the programmed behavior of the controller cannot cope with the situation. In this case the best that can be done is to raise an error signal and inform that the system is diverging from an optimal scenario. It is important to note that the optimal scenario has been defined statically at design time and is embedded into the controller behavior.

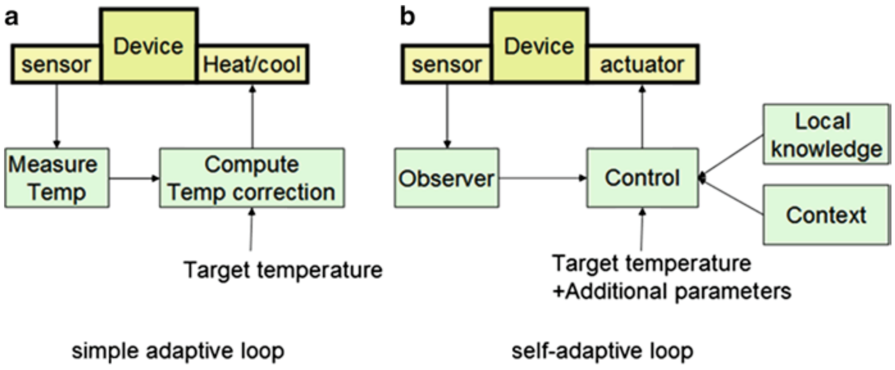

Fig. 7.1 Representation of a simple adaptive loop (**a**) compared to a self-adaptive loop. (**b**) Self-adaptation is more than an adaptation loop; it is control loop + knowledge

In the self-adaptive system, the optimal scenario (probably the most frequent one) would be part of the initial knowledge embedded into the system at design time. A knowledge management structure would constantly monitor the behavior of the loop and keep track of patterns of activity. This unit would also keep track of the related context and store initial parameters. In such an arrangement, a local knowledge base contains a set of optimal scenarios in the form of various parameters (e.g. range of values, bitstreams, algorithms, etc.) applicable for different known situations or contexts. In case a non-optimal scenario occurs, a self-adaptive system will be able to start a process for selecting a scenario (i.e. part of the current knowledge base) adapted to the current context. In a more sophisticated implementation, the system will be able to synthesize a new scenario adapted to an unknown situation and update the local knowledge base.

That's where we enter the realm of self-adaptivity. The first ability of the system should be to detect that it is diverging from a normal scenario. The second ability it must have is to take a decision on what is the best action to take when the nominal control scenario is out of bounds.

Using a programmable computer, adaptive and self-adaptive loops can be easily programmed in software using known software techniques but there is more than the mere programming of loops in the design of a self-adaptive system.

Software can be programmed in a self-adaptive way without necessarily relying on specific support mechanisms provided by the system (self-adaptation embedded in the source code); this self-adaptation mechanism can be entirely described at the software level. When software is considered, a number of possibilities for self-adaptation are available: run-time and dynamic change of the application goals (i.e., the application changes its high-level requirements for the system), adaptation based on selection of different behaviors (i.e., a different implementation of the same algorithm is selected), and intra-algorithm adaptation (i.e., some of the parameters of the considered algorithm implementations are changed at run-time). All of these self-adaptation mechanisms could be directly implemented in the software application, even though the first method requires support from the system to be effective.

This is fine as long as the adaptation process does not require any data from or any action to the underlying system.

Nevertheless, management of self-adaptivity should not be mixed with the functionalities in applications as this would put a burden on application developers and it would be a source of programming errors. *Separation of concerns*, which is a principle in which this separation is defined, should be therefore applied at all system levels.

Self-adaptivity can be both in the software and in the underlying hardware; the self-adaptivity control mechanism must provide a way to manage adaptations in hardware and in software effectively, thus allowing reaching application goals. Adaptation at different levels must be coordinated properly.

At hardware level there are two possible kinds of self-adaptation: structural (i.e., change in the functional units or in the interconnections) and on parameters (i.e., hardware parameters – such as frequency – are changed during run-time). Self-adaptive hardware can either manage adaptations internally or it can delegate (partly or entirely) this management to the software layer. For providing internal self-adaptation, the hardware needs to be able to change its configuration in a transparent way with respect to the software layer. Whenever software support is required for self-adaptivity, the hardware must notify to the software its reconfiguration capabilities. In both cases, the hardware may provide some information on the application execution and on the parameters that can be monitored and/or directly controlled by the software layer. Different kinds of hardware architectures may be utilized in a self-adaptive system. Software developers should be enabled to write applications without necessarily knowing the structure of the underlying hardware and the mechanisms used for self-adaptation. In fact, the management of all of these details would make the job of the programmer too complex, it would break portability of applications, and it would remove any convenience in using self-adaptive systems.

The task is then to find ways to engineering loops within the complete computing system: loops in the software, loops in the run-time system and loops at the processor hardware level. Moreover, the adaptation loops at the various level of the computing system shall be implemented in a way that interactions across them are not only possible but inevitable.

7.4 Self-Adaptation from the Software Engineering Point of View

What is *self-* at the software engineering level? Here we are not talking about platforms that adapt to the environment, we are talking about a program that is ***not*** engineered to respond to some changing environment (that would be merely part of the ***control*** structure of the code), but rather is ***presented*** in a way that makes it possible to achieve self-adaptation at the ***system*** level. Hence self-adaptivity in software engineering is always an indirect phenomenon, its manifestation is more efficacy than effectiveness, and in that it is an enabling factor rather than the effect itself.

How is software to be engineered to promote ***self***-adaptivity? Obviously, by trying not to ***prevent*** it by the obfuscation of the algorithmic properties with the machinery of **imperative code**. Whatever layers of adaptation lie below the top level of software engineering, these layers would need to have room for manoeuvre if they are to deploy their adaptation capabilities. But what specifically do we mean by this?

There are two main forms in which an application can be presented. First of all, it can be phrased in terms of state-transitions of an application-specific virtual machine. Indeed, that is what typically happens in software development processes in industry nowadays. A programmer creates a virtualization level by preparing program code for basic computations of an application domain, such as FFT, filters, correlators, etc., in the form of library function, and then a program is written which follows a transition diagram of the "business logic" (a video encoder would be one good example) using library functions with each transition. The problem with this form of presentation is that the notional state machine that makes those transitions is too rigid a metaphor, and so it encourages the implementer to realize it literally as a sequential state-transition process. The notional transition diagram, however, is rich in symmetry, which enables a good programmer to produce parallel code, whereby several avenues across the diagram are taken at the same time while preserving its sequential semantics. Nevertheless, doing so requires a deeper understanding of the business logic than the original design required and/or passed on. Such understanding is not readily available given the prevailing training and recruitment paradigms in industry, but even where it is, the process of designing genuinely parallel code is very complicated and error-prone.

There is another form in which programs can be presented. That form is based on the view of an application as a kind of dynamic circuit, in which information flows along streams that connect component nodes. It is **a *stream processing* view**, in which a program is conceptualized as a graph and the nodes of that graph define the *mapping* of inputs onto outputs. For example a node that multiplies input data by two before outputting it is seen as an order-preserving mapping of any x in the input stream onto the output stream with the value of the corresponding element being $2x$. The radical difference from the state transition view is that the "library function" here, i.e. multiply-by-2, is not *called* by a transition machine of any kind; instead it is activated by the arrival of the data itself. The stream processing view is inherently non-prescriptive as far as the order of actions, it is asynchronous and naturally parallel. Moreover, it is highly available to programmers of all levels of proficiency since the desirable properties (concurrency, asynchrony, decentralization) are not ***engineered*** by the developer, but are discernibly present simply in the way the application is presented. They aid self-adaptivity by allowing the ***system*** to choose its way of executing the program, by activating those nodes that have data and are not overloaded, while reducing the pressure on the nodes that are either overloaded or have insufficient resources. The programmer, the system designer and the application domain specialist are not involved in making those adaptations; rather the system itself finds its way through the configuration space at run time.

In ÆTHER we have seen it as our purpose to employ the stream-processing abstraction as a vehicle of software specification and implementation. In order to expose as much potential self-adaptivity to the lower levels as possible we came up with a simple design principle, which we call ***aggressive decomposition***. Unlike the state transition view, where splitting a state machine into several communicating state machines is potentially error-prone, the stream processing view is essentially safe: for example, splitting a node into two connected in a pipeline is as well understood as the mathematical notion of function composition. Communication overheads are not an issue either: if the graph structure becomes too fine, one can fuse a graph segment back into a single node down at the implementation level without jeopardizing the semantic correctness.

The principle of aggressive decomposition is stated as follows:

Decompose the application into a graph using the smallest meaningful nodes possible

By a *meaningful node* we assume a node whose function can be understood in terms of the application domain concepts from its name and interface definition. For example, an FFT node is meaningful, and so is matrix dot product, but multiplication by 2 may be not.

The way we propose to achieve the required decomposition is a top-down strategy. A designer in collaboration with an application domain expert produce a stream processing graph annotated with nodal functions at the nodes and data types at the arcs. Then each nodal function is considered in turn. For a node that is complex enough to be decomposable, a subgraph is produced which replaces the node, until all nodes are smallest meaningful nodes. Then an application programmer is engaged to produce nodal code for all the nodes according to the relevant application domain algorithms, while an independent *concurrency engineer* writes some coordination code (using the language proposed by ÆTHER, S-Net) which defines the graph structure and the packaging and synchronization of any data as it moves about the graph. In reality, the strategy could be partly bottom-up, partly top-down: some components may already be available and some networks can be designed without the knowledge of the bigger picture. Finally the coordination program is presented to a self-adaptive SANE platform, which dynamically merges sections of the graph into single nodes for adaptation purposes and from time to time splits merged nodes back into networks when the conditions change. We have partly implemented this vision and have shown that this style of application design is possible and natural to a range of applications. It is up to the future research to demonstrate that lower levels of the system hierarchy as envisaged by ÆTHER can use the exposed ***adaptability*** to achieve ***self-adaptation***. ÆTHER has certainly made a start in this direction.

7.5 Separation of Concerns

Dijkstra was probably the first who saw software engineering as a systematic activity based on separation of concerns [5]. Indeed, it is as true now as it was in his life time, only the concerns have become much more involved. To an extent, the software

Table 7.1 Separation of concerns between concurrency engineering and application programming

Concurrency engineer	Applications programmer
Basic understanding of component logistics	Expert knowledge of component logic
Expert knowledge of concurrency issues	Basic understanding of component composition for a given application
Focus on: coordination of components in multi/many-core systems	Focus on: algorithms, correctness, abstract complexity of components
Additional focus on: system self-adaptivity and flexibility	Additional focus on: component compatibility and generality

engineering strategy outlined above is about separation of various concerns: those of adaptivity from those of program specification, those of coordination from those of computation, etc. However, we believe that the same level of disruptiveness is required in the mechanisms of *abstraction* (another name for separation of concerns) as it is in more immediate implementation-related aspects of application development. Here the principle that ÆTHER has put forward is one of ***universal componentization***. The components in our stream processing view are, naturally, graph nodes. What is new here is that we do not allow them to have a variety of connection types and a variety of state-transition behaviors.[1] We have found that for the many types of applications components do not need to have a *persistent* state. This means that whatever internal stages the function may go through, whatever decisions may be made that cause it to change the course of computation, as soon as the result is produced and sent to the output, the state can safely be destroyed. This means that the processing node is a pure function that inputs one message and outputs zero, one or more messages and then re-initializes itself.

Moreover, we institute a SISO (single-input, single-output) principle [6, 7], whereby the component has only one input stream and produces only one output stream. On the surface this strategy seems very restrictive, but in fact it is not. The reason for it is our use of non-determinism as a phenomenon that almost replaces the multiplicity of incoming stream connection by allowing data to be arbitrarily ordered when streams are merged into one. It is yet another principle that promotes uniformity and separation of concerns. Suffice it to say that careful use of data types and some structural second-order components that we call *combinators* make non-determinism an enabling mechanism for multiply connected networks of singly connected components, as well as an additional mechanism of self-adaptivity.

Our approach draws a clear dividing line between the coordination infrastructure, fully defined by the concurrency engineer, and the computational infrastructure for which the applications programmer is responsible, and that line also divides up the variety of concerns (see Table 7.1). The concurrency engineer sees components as black boxes equipped with a certain meaningful interface. It is his concern to supply the

[1]It should be said that despite the fact that the whole application is a graph rather than a state-transition machine, the nodal functions are pieces of conventional code written by an ordinary applications programmer, hence they *do* have *a* state-transition behavior.

required data by bringing in, mixing and synchronizing various streams present in the graph in arc form so that a unit of work could be produced from a unit of message.

It is also the concurrency engineer's concern to ensure that the unit of work which produces output messages through the component interface is supported by the outgoing arcs that will deliver these messages to the nodes that need them for their units of work. The concurrency engineer has no great need to understand the intricacies of the processing scheme; if the designer has done his job properly, the same engineer can be used to process data in areas as different as multimedia compression and high-energy physics. What will change is the graph and its labelling, but not the nature of the stream coordination. The quality indicator of the concurrency engineer is his ability to appreciate the system's requirement for self-adaptation and to introduce monitoring, feedback and reconfiguration facilities on top of the basic data-processing scheme. He is empowered to do so by the high level of abstraction provided by our paradigm, which guarantees that encapsulated data cannot mix and be delivered to the wrong place in a type-correct program.

At the other end of the software production line is an applications programmer, now liberated from *all and every* communication, concurrency or synchronization concern. The programmer now assumes the availability of bite-size work, prepackaged for execution with all the necessary data and state information that the node needs, and for which the computation must result in the production of zero or more messages. Even the fate of these messages is not a concern for the programmer, only the values contained in them. Never before in the history of distributed parallel computing was the application programmer so much liberated. There is no miracle here either: the functions formerly foisted on that programmer are now passed on to a generalist (as far as the application domain) concurrency engineering expert. That is the main achievement of the ÆTHER software engineering philosophy – and the reader will notice that its feasibility is inexorably linked with the adaptivity of the platform. Without that adaptivity, the concerns of parallel execution cannot be approached by the concurrency engineer *generally*, and as a consequence, some of them would be passed up to the applications programmer thus destroying the separation of concerns.

7.6 Self-Adaptation from the System Engineering Point of View

First we make the observation that executing sequential code on a conventional core does not really expose much opportunity for adaptation of any kind. Some tradeoffs may be possible here in terms of performance vs. power requirements but it is not until we introduce the concerns of concurrency and reconfiguration that this picture becomes anything other than one dimensional. Thus we assert that the fundamental issues in adaptation from the systems engineering point of view are largely concerned with defining concurrent units of work, mapping those units onto available processing resources and providing a schedule for execution, where multiple units of work are mapped to the same resource. The latter is critical in providing an efficient utilization of a given resource when embedded in an asynchronous networked environment. The scheduling of virtual concurrency or parallel slackness [8] is the only generic mechanism available to enable an application to tolerate high latency operations, while maintaining throughput at the processing resource. Moreover, for efficiency, that scheduling should be data driven rather than based on any polling mechanism. This is a fundamental tenant of the dataflow principle, which executes a unit of work based on data availability rather than through some pre-programmed sequence [9].

Having established what we mean by adaptation, again we have to ask what the self at the system engineering level is. And again, we are **not** talking about control structures in the program that are engineered to respond to some changing environment. This would need to be predicated on the program having knowledge of its environment or mapping. Self-adaptivity in systems engineering comes about through the dynamic mapping and scheduling of defined units of work onto available resources in order to meet the computational demands of the application, while respecting any system constraints such as resource limitation, power consumption and physical location. Thus self implies a layer, component or components in the system, which given some knowledge of its environment and given some knowledge of its instantaneous or predicted computational load, reconfigures itself by mapping and scheduling the computation to best meet any constraints imposed upon it.

Clearly, one of the key separations of concern in systems engineering for self-adaptation is the **separation of the definition of units of work from their mapping and scheduling on a set of resources**. This implies providing the means to decompose programs into concurrent units of work in an abstract manner that does not make any assumptions about mapping and which moreover retains some characteristics important for such mapping. The main issue here is to be able to capture locality (i.e. tasks or units of work communicating with each other) in an abstract manner.

7.6.1 Granularity of Units of Work

One of the key issues in any form of parallel computing, which determines the efficiency of computation is the granularity of the unit of work which is mapped (distributed to another node) and/or scheduled.

From the point of view of distribution, if the cost of communication (time and/or energy) is large compared to the cost of computation at the remote place (time and or energy saved), then there is little to be gained from distribution unless the amount of computation performed there can be increased. In other words for a given cost in distribution, there is an amount of computation or grain size below which it does not make any sense to distribute the computation.

Similarly, from the point of view of managing concurrency locally, there is a cost for creating, synchronizing and scheduling a unit of work. Again, if the cost of managing concurrency (time and/or energy) is large compared to the cost of computation at the remote place (time and or energy saved), then there is little to be gained from executing the unit of work concurrently unless the amount of computation performed there can be increased.

These constraints provide quite a dilemma in being able to define a virtual machine model for self-adaptation. On the one hand we need to separate the decomposition of a program into concurrent units of work from the processing resources to which those units of work will be mapped. Yet on the other hand, as we see above, there are constraints on the minimum size of a computation. The only solution to this dilemma that maintains generality is to statically decompose the application into the finest level of granularity possible and then to dynamically aggregate units of work (sequentialize the concurrency) where necessary. Unless a fine grain approach is adopted, fine-grain architectures cannot fully exploit the concurrency available, which may only be at the instruction level in some algorithms. However, a well-defined concurrent composition can always be executed with a sequential schedule to meet the granularity constraints imposed by a particular processing resource.

This in turn raises a number of other issues, such as how to represent the code for the abstract machine and how to dynamically transform this code so as to perform this aggregation (*granularization)*. These issues are best dealt with from a more concrete perspective and will be picked up once the Self-Adaptive Virtual Processor has been defined.

7.6.2 SVP – An Abstract Model of Concurrency

In the discussion above two major issues have been identified. Self-adaptation requires the abstract decomposition of a computation into a maximally concurrent representation. That computation must then be dynamically mapped to available resources in order to meet the constraints imposed by the computation itself, i.e. non-functional constraints or by its execution environment.

To achieve this we have defined the Self-adaptive Virtual processor, (SVP), which captures concurrency as hierarchical families of threads, where dependencies between threads are captured by defining synchronizing variables in the code (i-structures or dataflow synchronizers). SVP [10] is an abstract model of concurrency developed from prior work on Dynamically-scheduled RISC (DRISC) architectures [11] and refined into a general computational model in the NWO Microgrids (2005–2009, [12]) and EU ÆTHER (2006–2009, [13]) projects.

In order to justify the choices made in defining SVP, against the more general model of fine-grain dataflow, a number of patterns of concurrent execution need to be explored. The first of these is *replicated concurrency.* Here a unit of work is often a loop body, which can be executed independently a given number of times. That number may be statically known, dynamically know prior to concurrency creation or may be dynamically determined during concurrent execution. With the exception of dynamically terminated replicated concurrency, this is the low hanging fruit, exploited in most programming and execution models. These independent units of work can be distributed or aggregated quite trivially. In SVP, this is captured by the SVP create action, which creates a family of identical threads, where threads are differentiated only by their index, which is initialized on creation.

A second form of concurrent computation often exploited in concurrency models is *asynchronous functional concurrency,* where the unit of computation is a function application. Here, concurrency is obtained by executing one or more functions asynchronously with respect to the "calling" function. There are two approaches to this form of concurrency: *non-blocking,* where all parameters to the function need to be defined prior to its concurrent invocation and *blocking,* where a function is invoked regardless of whether a parameter is yet defined and where the function blocks when it requires that parameter. The latter exposes more fine-grain concurrency, as whereas the unit of scheduling in non-blocking execution is the function itself, in blocking execution it becomes a partition on the function defined by the synchronization points determined by the functions use of its parameters. This means that, in the limit, the units of work that are scheduled may be individual instructions. Dataflow computation is asynchronous functional concurrency, where all functions are defined as machine instructions, i.e. where control flow is replaced entirely with the dataflow firing rule, which required all operands of an instruction to be defined before an instruction is scheduled.

The SVP model combines control flow and dataflow for efficiency in implementation. It implements asynchronous functional concurrency in a blocking manner, where a function of arbitrary granularity is dynamically instantiated using the create action and where the function definition defines certain of its parameters as synchronizing.

In dataflow, a program is represented by a directed acyclic graph, where the nodes represent the instructions and the arcs the dependencies between those instructions. In dynamic dataflow, cycles, as induced for example by loops, are removed by the expedient of coloring or tagging the graph's edges. There are no other limitations or the form of the graph, which exposes all possible concurrency in a computation, which is both its advantage and achiles heel. The main barrier to its implementation is in only being able to expose a fraction of this dependency graph at any time due to the constraints of only being able to implement a finite set of synchronizers. The SVP approach is preferred over this approach in order to limit the amount of resources required for the capture of a program. It is much more efficient to use sequence as a synchronizer where the delay in instruction execution is known and can be statically scheduled.

The final form of concurrency supported in SVP is that of a classic pipeline, which is implemented as replicated concurrency with a linear dependency (or dependencies) between the threads created in index order. This is really an optimization on asynchronous functional concurrency, which reduces the cost of concurrency creation.

The SVP model is a hierarchical one. Any thread function in SVP may create subordinate threads to an arbitrary level. The only constraint on the depth of such recursive concurrency comes about in implementation, where a finite set of synchronizers (i-structures) must span all dependencies exposed. As dependencies in replicated concurrency are constrained to be linear in index order, it is trivial to sequence the creation of dependent threads in a single family and restrict the synchronizing resources required to those required by a single thread. The same applies to recursive creates, where synchronizers must be provided for all functions in the create tree. Providing both concurrent and sequential versions of the function are available, the run-time system may switch to sequential execution in a deep create tree at the point where the last of the synchronizers is allocated.

7.6.3 Implementations of SVP

SVP is defined in terms of its concurrency control actions and captured in an extension to the C language where the concurrency controls and synchronizing variables extend the syntax and semantics of the C language in μTC [14]. Using the create action and its corresponding synchronization point, which determines when the created function has terminated (and updated global memory), both asynchronous functional concurrency and replicated concurrency can be captured in the model. This provides an abstract representation of a deterministic concurrent program. To achieve an implementation SVP must provide a number of further actions and concepts. *Places* are introduced as identifiers for processing resources and in an implementation, a run-time system will attach a place at any appropriate point in the concurrency tree. The model also provides an asynchronous mechanism to terminate the execution of a concurrent unit of work and a mechanism to reflect on the result of such actions, so that fault tolerance can be built into the systems developed using SVP. Finally, the model is made fully generic (and non-deterministic) by the use of an exclusive place used as a mutex, which sequentializes any request to execute a function at that place.

In the ÆTHER project we have implemented SVP at a coarse level of granularity using compilers from μTC that translate this language into C++ and bind the resulting code with a library that implements the SVP actions. This implementation has been used to demonstrate a complete tool chain from S-Net programs down to the execution of functions on FPGAs, which are dynamically selected at run time for executing S-Net components.

One of the successes of the SVP approach outside ÆTHER project has been its use in the EU Apple-CORE [15] project, which is investigating the implementation of SVP at the finest level of granularity, namely in the ISA of a DRISC processor. In this implementation binary code can be used as the representation for a self-adaptive program and the various units of work can be mapped and scheduled to cluster of cores of varying size with any changes to the binary code at all. This project is also developing high-level compilers to and from this model. This work includes dedicated SVP support for the functional array processing language SAC [16, 17], as well as an automatically parallelizing compiler for legacy C code [18].

7.7 At the Hardware Level: Self-Adaptive Networked Entities

At the processor level, the key concept that supports self-adaptive properties at the hardware level is the Self-Adaptive Networked Element (**SANE**). By designing each computing element along the SANE design pattern, we guarantee its seamless integration within the ÆTHER framework. The prerequisite is that each SANE processor implements the SVP protocol for concurrency and resource management. Furthermore, SANE implements a compute-monitor-control loop, making computing elements aware of what they are currently executing. This is the basic mechanism which allows a SANE element to manage as much things as possible on a local basis. This way each and every bit of computing resource has a level of autonomy that makes it suitable to accept jobs delegated by the Run-Time system and returns reports indicating the actual cost of execution. Each SANE is responsible for meeting the conditions of any contract that it may agree to with the SVP program. The SANE design patterns can be applied to a variety of hardware targets. In the course of the ÆTHER project we have explored implementations based on standard processors and hardware reconfigurable technologies but we believe that it could also be applied to future architectures and technologies such as bio-inspired and nano-fabricated hardware.

Basically, the SANE concept has two main properties which are described in [19]. The first one is that it can react autonomously by changing its working parameters or its structure to improve its behavior (i.e. to meet the requirements of the contract it has accepted). It is represented by the closed "compute, observe, control" loop in the Fig. 7.2 that provides the SANE with embedded self-adaptation. The second property of the SANE is that it is a collaborative entity which is able to publish its capabilities and also listen to other SANEs for their capabilities. It is also able to delegate part of its work to other SANEs. This collaborative interface is based on the SVP protocol that is described in the previous section.

As shown in Fig. 7.2, the SANE has been decomposed into four main blocks. Each of the blocks represents one of the four main properties of the SANE. The Computing Engine block is meant to be a computing resource for processing data in the most flexible way. The computation process is monitored trough an observer that computes metrics about properties related to the parameters to be monitored. The goal of the observation feature is to capture self-adaptation triggering events in order to feed the adaptation controller. Based on the information about the self-adaptation triggering events, the adaptation controller takes all the required decisions to face potential problems or to optimize the overall computation and perform parametric or structural adaptation if needed. This observation-control loop enables the SANE to manage its own resources.

The SANE has the ability to collaborate with other SANEs by means of a collaboration interface. This collaboration interface enables the SANE to publish its abilities to the environment. It is also able to delegate and to receive applications or parts of them to/from other SANEs. It is based on the SVP/SEP protocol (SEP, for System Environment Process, being a resource negotiation protocol on top of SVP [20]).

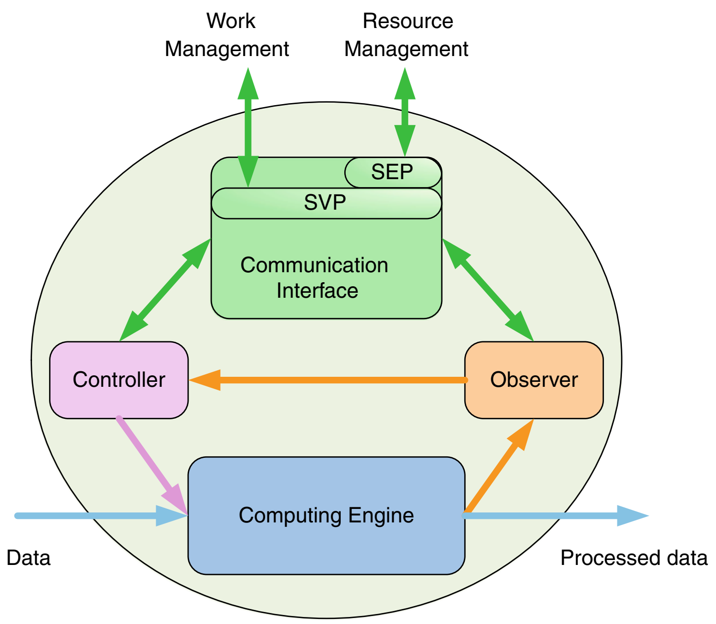

Fig. 7.2 General view of the SANE. The compute, observe, control loop represents the embedded self-adaptive behavior of the SANE whereas the communication interface represents its ability to collaborate with other SANEs

When defining a self-adaptive architecture, the designer needs to list the different events the architecture will have to cope with (e.g. failures or optimization to new applications). This step enables to provide the architecture with the right sensors/ observation features to capture them. The observation feature of the SANE enables it to track self-adaptation triggering events (i.e. modifications of the environment or of the computation that will imply an adaption). From the hardware point of view, these self-adaptation triggering events can be classified in different categories, listed in Tables 7.2–7.4.

A designer should implement a SANE regarding these events in order to both implement the required sensors (to observe the events) and to provide the implementation with the required flexibility (including the inner controlling algorithms) so as to be able to react to these events.

A self-adaptive system needs observations to know about its state (self-awareness) and the state of the environment (context awareness). These observations are done through the use of sensors that take raw information and send them to the observer. The observer transforms this information into variables and metrics (i.e. complex variables or events) that can be exploited by the control to:

1. Trigger an adaptation process
2. Choose which part of the system to adapt and the type of required adaptation.

Table 7.2 Part I. Events, related measures, and possible adaptations at the hardware level (delegation and observations of the computing environment are more related to system level but may also have an influence on the local hardware resources)

Event	Related measures/observations	Possible adaptations
Deadline miss	Execution time by online profiling or workload prediction	1. Increase the processing speed by increasing the clock rate (or processing rate) 2. Change the implementation or the routing of the task to gain speed if possible (possibly switch to an interpolation implementation if exact implementation is too slow) 3. Take advantage of possible application parallelism by requesting additional local resources if available 4. Take advantage of the pervasiveness of the system by requesting computing resources in the environment to delegate the computation
Lack of battery power	Battery power given by sensors	1. Decrease clock rate or processing rate (if possible) 2. Change the implementation for a less power consuming one (including change of the physical routing links, clock or power supply gating, etc.) 3. Delegate the computation to another resource in the environment

Based on the taxonomy given in Tables 7.2–7.4, a number of sensors that can be used to take knowledge from the environment and from the system itself can be listed.

Table 7.5 shows the variety of the different variables that may be monitored and exploited by the observer. One can notice that the types of the observations are very different from each other. Thus, the interfaces between the sensors and the observer are to be considered carefully for such a system. In the end, the observer is a data transformation process that gathers the raw observations from the sensors and computes a report composed of a set of metrics. These metrics help the controller to monitor if the selected self-adaptation triggering events occur. The different adaptation scenarios described in Tables 7.2–7.4 are not possible with all the implementations. For example, changing the hardware implementation of a task is only possible with reconfigurable architectures such as FPGAs.

Table 7.3 Part II. Events, related measures, and possible adaptations at the hardware level (delegation and observations of the computing environment are more related to system level but may also have an influence on the local hardware resources)

Event	Related measures/observations	Possible adaptations
Change of the data to be processed	Observation of data type through tags Observations of the data values	1. Dynamic modification of the interfaces (clock rate, synchronization mechanisms, protocols, bus bitwidth, etc.) 2. Modification of the implementation of the function
Change of the mission (application or constraints)	No real associated measure: it depends on a control command given from the user who wants to execute a new program or another resource that delegate a task	1. Change the current application context 2. Tune the parameters of the already loaded task (soft and fast adaptation) 3. Change the task or its implementation (hard and slow adaptation)

In traditional microprocessors, there is no possibility of structural adaptation, whereas they are quite efficient for parametric adaptation (example of DVFS – Dynamic Voltage and Frequency Scaling – algorithms for frequency and voltage scaling). The adaptation of the implementation must be done at the software level. This implies that the adaptation possibilities depend on the level of flexibility provided to the computing entity by the chip designers. A completely flexible design will have the potential to deal with a larger number of situations in comparison with a design that only have one or two degrees of freedom. But the potential advantages of a completely flexible design also imply drawbacks leading the system designers to consider trade-offs.

These drawbacks are linked with both the degree of flexibility of the system and the adaptation mechanism (adaptation rules and learning abilities). The more flexible the architecture is, the more complex the adaptation engine will be (since the design and space exploration for selecting the good target configuration is larger). The adaptation mechanism is also of great importance with respect to the control complexity. This mechanism can vary from simple control mechanisms such as "if then else" statements that allow self-adaptation only between well-known situations to learning and evolvable algorithms that may be very complex and take time to converge, even if low convergence time is a practical criterion for self-adaptive systems (see Table 7.6). In order to be useful, the decision taking mechanism and the actual adaptation needs to be done quickly at a rate depending on the application processing rate.

It is obvious that the observer cannot capture all the variables to know about the complete state of the object and/or of the environment (due to the overhead of the observers and sensors). The sensors and the observer help the controller to construct and evolve (through adaptation rules) a model of the environment and the

Table 7.4 Part III. Events, related measures, and possible adaptations at the hardware level (delegation and observations of the computing environment are more related to system level but may also have an influence on the local hardware resources)

Event	Related measures/observations	Possible adaptations
Failure of a computing node or of the network (partial or global: only one part of the chip is concerned or not, transient or definitive)	Results from a set of available self-tests or remote tests (another entity does not respond to requests, etc.)	1. In case of partial failure (chip level): avoiding the faulty unit (re-routing process by self-organization, switching to a degraded mode, try to repair 2. In case of global failure: try to repair (chip level), re-routing/isolation of the faulty chip (system level) 3. In case of transient failure: error detecting/correcting code (depends on the error frequency and the deadline), etc. 4. In case of permanent failure: error-correcting code or re-routing
Change in the Environment (new resources coming/leaving, value of parameter, etc.)	Depends on the Collaboration protocol between the entities. For a publish/discovery mechanism, an entity can observe the available services offered by new resources. The system must also be aware of leaving resources.	1. Take into account new needs based on experience: reconfigure to accept new applications or tasks seamlessly (prediction) 2. Take advantage of new services offered by the environment to optimize internal process (slower the internal computations and delegate some work to a more efficient and maybe specialized entity, etc.) 3. Recover from an interrupted computation (due to a leaving resource), re-distribute the computing tasks.

Table 7.5 Examples of variables to be observed related to self-adaptation triggering events and examples of the related measurement sensors

Observed variables	Examples of related sensors
Task speed	Timers
Battery power	Power supply current sensors, in-battery sensors etc.
Data type	Sensors related to meta-data extraction and identification
Mission change	Identifier of pre-defined scenarios, critical parameter triggering pre-defined mission switch (the mission semantics is generally statically defined at a higher abstraction level and not sensed as such)
Failure	BIST sensors, on-chip noise measurement sensors [21], etc.
Environment	Sensors related to the physical environment (such as temperature, light, etc.)
	Sensors related to the computing environment (sensing new resources and their capabilities, timers for timeout measurements, etc.)
State of the chip	Temperature sensors, current sensors, etc. [22–24]

Table 7.6 Different types of self-adaptation control algorithms and related characteristics.

Control structure	Time to converge	Potential complexity	Ability to deal with unknown situations by self-adaptation
Control "if then else"	Fast	Low	Low
Evolutionary and learning algorithms	Potentially slow	High	Potentially high

chip. This model can be explicit or implicit. For example, the model can be explicit if the chip and its environment are formally described as sets of variables and equations. The model is implicitly described in the "if-then-else" mechanism since this statement contains the necessary knowledge to take an adaptation decision based on the gathered observations.

One can notice that one of the major problems of self-adaptation is the overhead of the observation-control loop compared to the computation process. Taking into account one variable describing the system, its environment or a self-adaptation triggering event implies to include the related sensors, interfaces and logic in the observer but also the related control logic in the controller. If the number of variables is high, it can slow down the decision taking mechanism. One solution could be to distribute the management of the parameters to different control loops, which may be difficult since the different chip parameters are not independent from each other. ÆTHER considered and studied different implementations of the SANE concept and its specific parts especially the Computing Engine.

7.7.1 SANE Implementation, the Case of Reconfigurable Hardware

At first, FPGA technology seems a rather natural candidate when it comes to implementing an architecture that changes over time. The capacity of new FPGAs to be dynamically and partially reconfigured (even internally for certain Xilinx FPGA families with the ICAP interface) is seen as a nice self-adaptation enabler for modifying the internal structure of the architecture that is placed on the device [25].

The fact that FPGAs are in essence fine grained reconfigurable architecture is both a very interesting property and a real problem when designing self-adaptive architecture. This paradox is linked to the above-presented trade-off between the degrees of freedom of an architecture and the required control complexity to derive the new configuration. The high degrees of freedom provided by FPGAs makes them good candidates to implement SANEs since almost every modification of the placed architecture can be envisioned. Therefore, the optimal architecture (i.e. the one that best fit to the current environment) could be obtained in principle. The problem resides in the fact that for approaching this optimal architecture, a lot of different knobs must be controlled (i.e. a new configuration composed of millions of bits should be computed). With a standard approach to hardware design, the steps necessary to compute a new configuration (synthesis and place/route) are very time consuming; they require tenths of minutes of computation and gigabytes of memory on the latest workstations. As a result, and as of today, this cannot be reasonably done dynamically at run-time.

The problem is alleviated on state of the art dynamic and partial reconfigurable FPGA designs by performing the generation of pre-defined partial configurations. Thus, the system just needs to load the pre-generated bitstreams at runtime. It means that the system simply choose between available configurations using for example a simple "if-then-else" control structure. These configurations may be hardware tasks (e.g. bitstreams) or software tasks (e.g. different implementations of an application or firmwares). This possibility offers basic self-adaptation properties such as loading the right configuration depending on demands from applications or choosing the best configuration by comparing the results and behaviors of multiple implementations. Additionally, having hardware and software flexibility enables to choose between fast adaptation (usually provided by software reconfiguration) and slower but deeper structural hardware reconfiguration.

A possibility to make progress toward the "on the fly" generation of bitstreams consists in considering coarser-grain architectures. Since a SANE needs to be able to rapidly adapt to the potential modifications of its environment, it must be quickly reconfigured. Therefore, the generation and mapping of the bitstreams required by the application needs to be automatic and very fast. These properties can be provided by coarse grain reconfigurable architectures. Such architectures have larger, more complex basic hardware cells requiring less memory bits for their configuration. One drawback is that these kinds of architecture are less flexible than fine grain reconfigurable architectures.

Another potential problem related to the use of FPGAs is that a partial configuration is only generated for a particular place on the device. The same VHDL or Verilog code (even the same netlist) will have to be placed and routed for every location on the chip. For example, with a system composed of a Xilinx MicroBlaze with four dynamic and partial reconfigurable areas as presented in Fig. 7.3, there are four places for hardware accelerators [26]. Even if the places have the same shape, four different bitstreams need to be generated per accelerator (i.e. per VHDL or Verilog description) to be able to place one hardware accelerator in

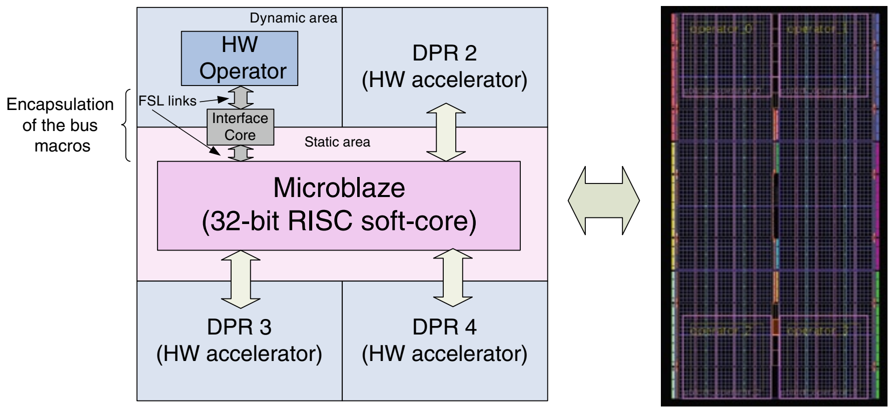

Fig. 7.3 Schematic of a partially reconfigurable chip composed of four independent reconfigurable areas (DPR) under the control of a local RISC controller (Microblaze). This is one example of a SANE hardware implementation using state of the art fine-grained reconfigurable chips. Floor plan view on the right

Table 7.7 Different types of relocation depending on the targeted hardware

Type of relocation	Targeted hardware	Required transformations
Static synthesis, placement and routing	Exactly the same as the old place	Rerouting of the I/O of the module, all the internal structure of the relocated module remains the same
Static synthesis, dynamic placement and routing	Same resources but placed differently	The placement and routing of the module must be computed again (including the I/O of the module).
Dynamic synthesis, placement and routing	Different resources (i.e. optimized or spare resources)	The synthesis phase must be done again (and consequently, all the other phases)

each area. This implies a waste of memory resources for storing all the required bitstreams. Even if this problem can be alleviated using the collaboration interface of the SANE (i.e. the ability to ask for configurations to the network if the needed configuration is not found in local memory), **re-locatable hardware** remains one of the great challenges to be tackled to design self-reconfigurable hardware architectures.

7.7.2 SANE and Future Technologies

The relocation ability can have different levels depending on the structure of the targeted hardware place (see Table 7.7).

In order to efficiently perform the relocation of hardware implemented tasks, two prerequisites are needed at least:

- First, the task should be described in a sufficient abstract language (abstract when compared to traditional hardware netlists) so that the required relocation be possible. In particular it should not contain any absolute locations for the routing information ("parameterizable" IPs).
- Secondly, a built-in distributed self-placement and self-routing mechanisms should be available. The hardware substrate can then take the responsibility to place and route the different tasks in a distributed way, without the need of a centralized place and route supervisor.

So far, the steps and algorithms implied by those two prerequisites are very computationally intensive. They would require embedding powerful processors within the hardware substrate, a very costly solution that is probably not the way to go. It is therefore a very promising but challenging opportunity to investigate solutions that could make relocatable hardware possible in future generations of FPGA by possibly using 3D-stacking and routing techniques.

7.8 The ÆTHER Computing Framework: Implementation and Applications

The technologies described and discussed above are just ways (the ÆTHER ways) to try and give solutions to the generic problem of implementing adaptation loops at the various level of a computing system (see Fig. 7.4). If defining and carefully crafting technologies adapted to the various levels is a good start, it is far from enough. The big challenge is thus to make those implementations of the common design pattern for self-adaptation work together. That is exactly what the ÆTHER computing framework is about.

7.8.1 Design Workflow

The ÆTHER framework aims at implementing a design flow and supporting tools for future dynamic, self-adaptive applications down to their implementation on distributed processor architectures (standard multicores or μGrids [27]) or dynamically reconfigurable hardware [28] that implement the Self Adaptive Network Entity (SANE) concept. The framework is based on two so-called "box" languages, Single Assignment C (SAC) and Microthreaded C (μTC), one coordination language, S-Net, that supports adaptability, and on the definition of the SANE Virtual Processor (SVP) interface between architecture-naive application capture and the implementation on specific architectures (see Fig. 7.5).

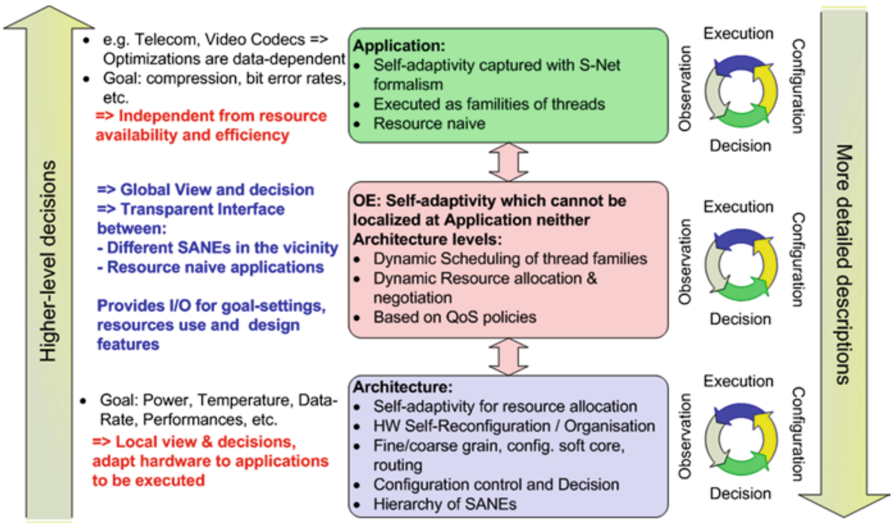
Higher-level decisions
• e.g. Telecom, Video Codecs => Optimizations are data-dependent
• Goal: compression, bit error rates, etc.
=> Independent from resource availability and efficiency
=> Global View and decision
=> Transparent Interface between:
- Different SANEs in the vicinity
- Resource naive applications
Provides I/O for goal-settings, resources use and design features
• Goal: Power, Temperature, Data-Rate, Performances, etc.
=> Local view & decisions, adapt hardware to applications to be executed
Application:
• Self-adaptivity captured with S-Net formalism
• Executed as famillies of threads
• Resource naive
OE: Self-adaptivity which cannot be localized at Application neither Architecture levels:
• Dynamic Scheduling of thread families
• Dynamic Resource allocation & negotiation
• Based on QoS policies
Architecture:
• Self-adaptivity for resource allocation
• HW Self-Reconfiguration / Organisation
• Fine/coarse grain, config. soft core, routing
• Configuration control and Decision
• Hierarchy of SANEs
Execution
Observation
Configuration
Decision
More detailed descriptions

Fig. 7.4 Self-adaptation and its potential applications at different levels of the ÆTHER computing framework. OE stands for Operating Environment: it is the distributed runtime manager of the system. It allocates and schedules families of threads based both on QoS policies and resources availability

Different systems already use self-adaptivity to achieve different specific non functional goals such as self-repairing and self-organization. Self-adaptation has been used for a long time in different fields, such as, for example, networking and distributed computing. In networking, self-adaptivity is used at router level to provide quality of service (QoS) to the communications. Self-adaptivity can be used, in such a context, to recover from QoS violations. Other uses of self-adaptivity are in the context of distributed and grid systems; in this case self-adaptivity is used to provide QoS to the applications, to improve stability and fairness, and to manage resources efficiently. In complex networking environments self-adaptivity is presently being proposed for dealing with the heterogeneity in the behavior of network apparatus and to enable configuration-less insertion of new network elements.

Self-adaptivity may be also used to provide fault-tolerance capabilities as it naturally provides a way to deal with this issue by means of system reconfigurations. Applications designers might be provided with mechanisms to specify fault-tolerance requirements of applications; systems might be designed to automatically provide these capabilities by means of self-adaptation. In this context, self-adaptive systems might automatically put in place predefined self-adaptation patterns to provide fault tolerance at different levels.

The applications of self-adaptive computing are very diverse ranging from mobile and pervasive computing to any field that involves huge assemblies of computing resources on or off chip. In all of those applications fields, the computing

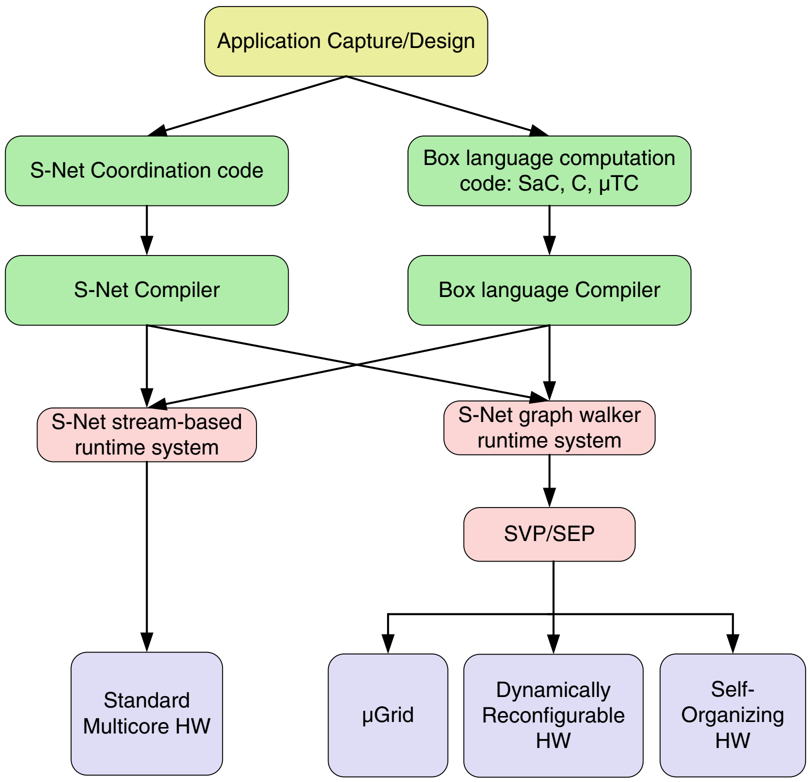
Application Capture/Design
S-Net Coordination code
Box language computation code: SaC, C, μTC
S-Net Compiler
Box language Compiler
S-Net stream-based runtime system
S-Net graph walker runtime system
SVP/SEP
Standard Multicore HW
μGrid
Dynamically Reconfigurable HW
Self-Organizing HW

Fig. 7.5 The AETHER framework

system is exposed to potential changes in its environment that may occur as a result of external or internal events such as incoming new devices, missing resources, power shortage, reduced connectivity, component failure, etc. Self-adaptivity, coupled with proper adaptation policies, helps dealing with such events, regardless if they can be foreseen at design time (e.g., the migration of the system in different environments) or not (e.g., a part of the system that becomes faulty). The ÆTHER platform already provides all the mechanisms that are necessary to handle most of these situations: it allows the system to change its behavior by mapping different functionalities (e.g., different network protocols) on hardware blocks or by using their software implementations; furthermore, the platform provides the capability to change system parameters (e.g., the clock of hardware blocks), to control performances of the different functionalities (e.g., by moving its implementation from software to a specific hardware block and/or by parallelizing some of its parts). Even the concept of self-adaptive security at application level relies on the aforementioned mechanisms to keep the security level of applications constant, regardless the changes in the environment and in the system. Proper self-adaptation policies need to be designed for different systems: a small embedded system will have different self-adaptation policies than a big distributed computational system. Though, the description of these policies is much easier than designing the system to cope with all known possible events.

Furthermore, the ÆTHER platform provides a first meaningful step in the direction of supporting automatic reconfiguration of systems to satisfy high-level non-functional goals of applications. By building on this platform, it will be possible to design systems in which these goals will be automatically satisfied by the system. Non-functional goals express some requirements that are not related to functional requirements (i.e., they are not related to the main functionalities of the applications); they express additional requirements such as the need of a certain level of security, the need of a certain level of reliability, and the need of a certain level of performances. For example, the user or the application programmer may want to specify that a certain application requires a certain level of reliability; the system should be able to reconfigure itself to try achieving this goal. For example, faulty components may be automatically replaced with non-faulty ones, even with a decrease of performance when graceful degradation is accepted. In other cases, the system may require more complex reconfigurations (e.g., by instantiating parallel slow components to replace a fast faulty one). Similar measures can be taken for supporting security requirements. In the same way, the programmer may want specify some performance requirements for the application; the system will self-reconfigure to provide the desired performances (e.g., by mapping certain functionalities to hardware or by parallelizing hardware components). The ÆTHER platform provides some mechanisms that can be used to support the utilization of high-level non-functional requirements: by using the mechanisms provided by the platform, adaptations to cope with reliability and performance problems are available. A mechanism to translate these high-level goals into proper control algorithms for self-adaptivity has still to be developed and remains one of the key areas where further research needs to be done.

One promising field of application of the ÆTHER technologies is to introduce self-healing or self-repairing behavior in computing resources in order to provide graceful degradation of performances under a variety of threats (e.g. power shortage, missing resources, sensor or general hardware failure).

7.8.2 Implementation and Applications

To illustrate the ÆTHER framework and its supporting concepts and technologies (see Fig. 7.5) which were described in previous pages, a coordinated set of demonstrations were implemented during the project.

Among the different demonstrations, two will be described in the following paragraphs. The first one shows how an industrial-level radar Moving Target Indication (MTI) application can be captured and expressed in S-Net with boxes implemented in the functional array programming language SAC (Single Assignment C).

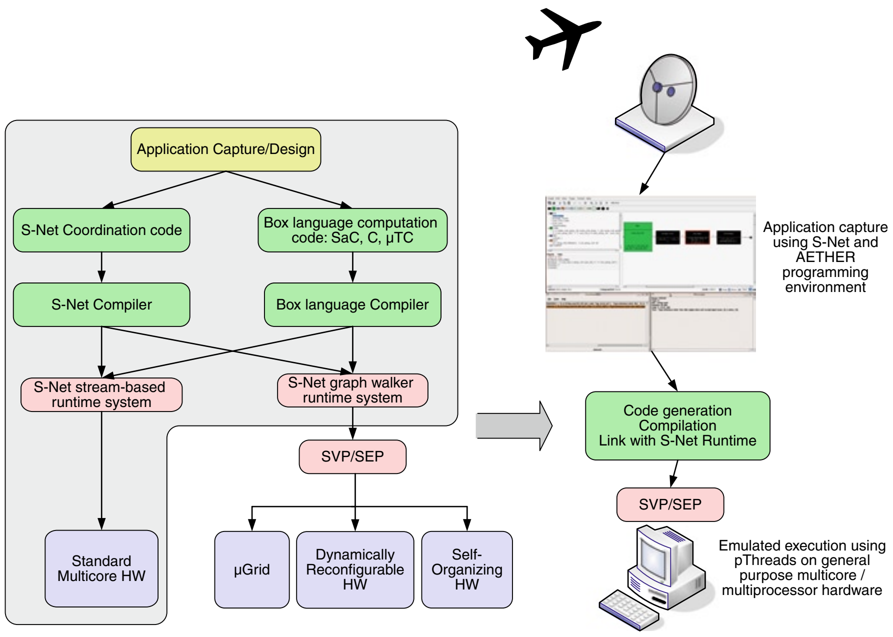
Application Capture/Design
S-Net Coordination code
Box language computation code: SaC, C, μTC
S-Net Compiler
Box language Compiler
S-Net stream-based runtime system
S-Net graph walker runtime system
SVP/SEP
Standard Multicore HW
μGrid
Dynamically Reconfigurable HW
Self-Organizing HW
Application capture using S-Net and AETHER programming environment
Code generation Compilation Link with S-Net Runtime
SVP/SEP
Emulated execution using pThreads on general purpose multicore / multiprocessor hardware

Fig. 7.6 Demonstration of a radar application, from S-Net to standard multicore hardware (emphasis on SNet and its programming environment)

The design of coordination code is supported by the S-Net graphical development environment, which is based on the MetaEdit + toolkit (see Fig. 7.6).

This demonstration shows how the graphical representation of an S-Net graph can automatically be converted into a textual representation. The S-Net compiler started from within the IDE validates the soundness of the S-Net description via type inference and type checking. The resulting code is then automatically linked with the stream-based S-Net runtime system for parallel execution on standard general-purpose multicore and multiprocessor hardware.

The goal of this first demonstration was to show the suitability of the S-Net coordination language to capture non-trivial, industrial-level applications. Its aim was also to demonstrate the entire S-Net tool chain:

- a graphical development environment for network construction
- a compiler for validation of soundness based on a type system with record subtyping
- a runtime system for automatic management of parallel execution on standard general-purpose multicore and multiprocessor hardware.

The goal of the second demonstration is to illustrate by a simple example the entire ÆTHER design flow from application capture to execution on a dynamically reconfigurable platform (see Fig. 7.7). A single application (Optical Character Recognition) is captured as an S-Net network of C-coded processing boxes.

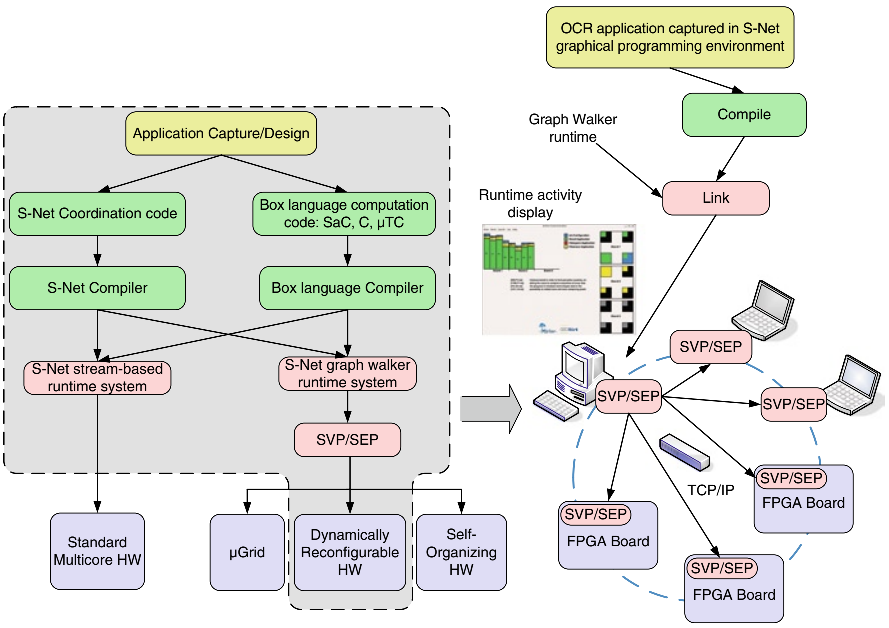

Fig. 7.7 Demonstration of the OCR application, from S-Net to reconfigurable hardware (emphasis on the execution of an S-Net-coded application on dynamically reconfigurable hardware, including the use of hardware accelerators)

Coordination design is also supported by the S-Net graphical development environment, which is based on MetaEdit + based S-Net environment. The box code and the S-Net network will be respectively compiled by standard C and S-Net compilers, and linked with the S-Net graph walker runtime which will interpret the S-Net network and interface with SVP/SEP capabilities on both a Linux platform (PC) and an embedded Linux for the Xilinx MicroBlaze processor within the FPGAs.

The application is then run on an hardware platform composed of a PC connected to three reconfigurable FPGA boards via an Ethernet network. On each of those boards, a SANE implementation contains an FPGA capable of running a character recognition task in software (on the MicroBlaze CPU) and/or in hardware, in one or several of its four dynamically reconfigurable areas. For the purpose of demonstration, the activities of MicroBlaze embedded microprocessors, PC and hardware SANEs are displayed on a monitoring PC. Reconfigurations are shown, first by having all computations done by the PC alone and then dynamically moved to hardware SANEs when they join the system, or redistributed when they leave: this will also show the capacity of the SANE to adapt themselves to execute the delegated task the best they can (by on-the-fly partial reconfiguration and dynamic retrieving of partial bitstreams over the network).

This demonstration gave an overview of the ÆTHER framework, showing application capture with S-Net, the compile chains, and SVP/SEP operations on reconfigurable SANEs. It illustrated how a programmer can build an application independently from any specific target. Furthermore and thanks to the implicit parallelism offered within S-Net, the demonstration illustrates how the Graph Walker exploiting the SVP/SEP protocol, can dynamically map tasks to reconfigurable resources and run them according to the specific properties of the target.

7.9 Open Issues and Research Opportunities

A number of documents and roadmaps have been edited that strives to identify some of the research opportunities for autonomous computing. They give us another complementary perspective on the ÆTHER research.

7.9.1 The AgentLink Roadmap

One such example is the AgentLink coordinate action that produced a roadmap report [29] on **agent based computing technologies**. Many of the topics addressed by the agent computing community are very relevant to the idea of self-adaptive computing. As the authors highlights in their conclusions, "for agent technologies, the objectives are to create systems situated in dynamic and open environments, able to adapt to these environments and capable of incorporating autonomous and self-interested components." Agent based technologies deals with computational entities designed in such a way that interaction among them is central. As a consequence agents need to "socialize" with their counterparts by exchanging information and have their own autonomous way of handling events. In this respect the AgentLink community has identified three categories of technologies that are of particular relevance for the agent system design:

- The **organisation-level** category deals with assemblies of agent entities, their structure and how they self-organize and handle a particular mission in a collective way. The ÆTHER research did not really propose any new concepts in this category and it is certainly an open topic.
- The **interaction-level** category deals with communication between agents at the language and protocol level. In a way the S-NET, and run-time protocols of ÆTHER fall into this category although with a computing approach mainly focused on managing concurrency and computing resources.
- The **Agent-level** category includes ways to implement autonomous behaviors and self-x features at the entity level. If the SANE concept naturally falls into this category it exhibits one major difference: the ÆTHER entity focused on a hardware based approach to provide autonomy at the SANE level when the agent community refers to techniques such as machine learning and artificial intelligence which are typically rooted in software engineering.

Several consequences can be drawn from this analysis. The first one is that future research in computer architecture oriented toward self-x should tackle the organization-level category described above. By studying the behavior of autonomic entities and their interactions within the context of the full ecosystem, apprehend them like collective assembly right from the start rather than aggregations of individual entities.

The second observation is in exploring more disruptive ways to implement the interactions and negotiations between entities. Finally, a mutual cross-feed between the AgentLink approach to autonomic computing, derived from software engineering, with the computer architecture engineering followed in ÆTHER should be beneficial in future research on self-x.

7.9.2 Self-Adaptive Computing and the ITRS Roadmap

The International Technology Roadmap for Semiconductors (ITRS) is an industry driven document that publishes technological visions for the developments of the semiconductor industry. The roadmap was historically focused toward silicon based technologies (essentially CMOS), but in the last few updates has made a move toward technologies beyond-CMOS in an effort to forecast their potentials as replacement technologies. This is the role of the Emerging Research Device (ERD) chapter of the ITRS. In the 2008 Document update [30], the ITRS exposes the trends and challenges in hardware and system design for the period 2008–2022:

- On the technology trends, the cell logic gate size is expected to increase almost linearly from the current values (2009) of 1 μm^2 (for MPU) and 0.01 μm^2 (for DRAM) down to 0.07 μm^2 (for MPU) and 0.001 μm^2 (for DRAM) in 2022. That's about an order of magnitude in the next 10 years which is likely to yield the same increase in processor core number for example. It is foreseen that the number of processing engines for portable consumer devices (cell phones, PDA, GPS) will jump from about 60 in 2009 up to about 1,400 in 2022. **Scaling according to Moore's law is thus continuing** for the next 10–15 years.
- On system chip trends the ITRS notes that the "More-than-Moore" approach will get a boost in the next decade. This design philosophy states that an increase of functional density can be obtained at the chip level by **allowing non-digital functionalities onboard or above the chips** (e.g. analogue circuits, power control, sensors, actuators, passive components). This is clearly a trend toward more heterogeneous Microsystems on chip. The updated ITRS roadmap also mentions the requirement for System on Chip reconfigurability to provide flexible, reconfigurable communication structures.
- On design trends, the ITRS update highlights several important challenges that will be crucial to tackle in order to design future chips. Among those: **Design productivity** needs to be enhanced in order to keep design and verification costs, **Power management** is getting critical and dynamic power and thermal control will be needed, **Reliability** is getting problematic below 45 nm because of atomic-scale effects and single-event upsets (soft errors). It is clear that runtime control, reconfiguration and healing techniques will be required.

- On the Emerging Research Device chapter, the ITRS 2008 updates notes that the most likely beyond CMOS technology would be **Carbon Nanoelectronics** (e.g. Carbon nanotubes, Graphene). If there are indications that such technologies would be able to yield adaptive components in the form of Carbon Nanotube transistors [31] most probably hybridized with CMOS processes [32], it is certainly very hypothetical at the moment of this writing.

From the ITRS roadmap we can see that the trend toward more complex computing systems is really continuing for at least the next decade. Heterogeneity of future systems will be growing, with several non-digital functions embedded within the chips and most likely included within the adaptation loops. For example power/thermal controllers embedded within microprocessors will be built with various digital and analogue components and will be based on adaptive loops that should be programmed in a way transparent to the application designer.

7.9.3 ÆTHER Research and Grand Challenges

In 2009, the European Commission ISTAG FET Working group produced a document that gathers a number of particularly important research grand challenges for the society and the knowledge in the future [33]. Based on the work gathered in several proceeding reports in the period 2003–2008, the report has identified five candidates for future research. Although all challenges are very multi-disciplinary in the topics addressed we can see among them that a good share is very relevant to the idea of self-adaptive and autonomic computing:

- **Understanding Life (Life Science)** – In this challenge we see opportunities in the Neuro-ICT challenges, Personalized ICT for health aims at providing systems for real-time, autonomous and personalized health-care. The **Neuro-ICT** challenge by aiming at understanding the functioning of the brain through emulation or simulation would greatly enhance the understanding of learning processes and in consequence **help implementing machine learning algorithms and adaptation behaviors** within self-x and autonomous computing engines.
- **Managing Complex Systems (Modeling & Simulation)** – In this challenge, the report explicitly mentions that advances in simulation techniques and most probably multi-scale simulation will benefit the field of "autonomous systems that perceive the environment, understand the situation, draw conclusions, act in an appropriate manner and cooperate with other systems". It is also indicated that such research will benefit the field of investigating **non-von Neumann principles.** Indeed the von-Neumann paradigm seems to show its limits with a strong binding with imperative programming models. Alternatives such as neuro-inspired principles have been proposed but so far have been used at very small scale. Going to the next level with those principles will requires **modeling and simulation of realistically complex systems** most probably in a mixed-technology and heterogeneous context. The same is true for investigating the collective dynamics

behaviors of large assemblies of future interacting self-x systems and their relation with their environment as emphasized in this research challenge: "**concepts for autonomous systems modeling**: Research in this area could also be directed to autonomous systems that perceive the environment, understand the situation, draw conclusions, act in an appropriate manner and cooperate with other systems."

- **Future Information Processing Technologies** – This challenge addresses future computing technologies such as quantum devices, novel processing devices, nano-computing and future data storage. If any research in those areas will definitely be key enablers for the development of self-adaptive computing, the report explicitly mentions **Self-repairing and Self-evolving computational devices** as a major research challenge. Indeed and as the report accurately says: "The research efforts in this area only scratched the surface of this problem, which will become increasingly more important as transistors shrink and computational devices are embedded in everyday objects or sent to outer space or inaccessible areas of this planet, not to speak within the human body." We could not agree more as we effectively learned while performing the ÆTHER research, that the problem of self-x and autonomous computing is tremendously complex and requires much more than what has been done so far.
- **Future problem solving technologies** – This challenge aims at studying completely new programming paradigms able to cope with the complexity of both future computing systems and future applications. The ÆTHER research very quickly identified that self-adaptive systems would require programming paradigms that need to be **non-imperative and includes ways to express semantics and non-functional features**: this is clearly a critical research topic that needs to be addressed. Conversely the report highlights: "It would be much more effective, if computers problems recognized problems to be solved from their observation of nature and the environment directly. This is a system automation problem, with sensory inputs and actuator outputs to the environment, monitoring it and determining that a new problem may arise, expressing/transforming it into a solvable formulation, solving it and acting back on environment with the solution." This way, the self-adaptive machine system becomes a central actor of the programming paradigm instead of being just a listener of a human centric language. The problem solving challenge also emphasized the importance of research on **trust, reliability and security challenges** which are indeed critical when deploying autonomous computing elements.
- **Robot Companions for Citizens** – It was probably one of the first applications in history in which the concept of a machine with autonomous behavior had been so clearly expressed. So it is today, where robotics, whether human-like or for the industry offer a reservoir of opportunities and challenges for self-adaptive computation. The research challenge in robotic is therefore of prime importance for the field of self-adaptive computing. In particular, the report indicates that research on robots brains, human-robot interaction and robot adaptive bodies will be critical. Included in this last challenge we read: "It is necessary to develop

electronic architectures that can support the requirements for adaptation, distribution, and rich sensory perception, yet at the same time, be compatible with novel body structures." Again we note that the concepts of self-adaptation and self-x are presents everywhere in this challenge. Finally the problem of ethics is evoked in the case of robots, but it goes far beyond robotics and can become a general issue when designing systems with autonomous behaviors be them robots or not.

Research in self-adaptive computing are key to almost all of the five research grand challenges identified in the context of the Future and Emerging Technologies flagship projects. Additionally it can be seen that those grand challenges are very inter-related with one providing enabling technologies for the others. The case of self–x computing systems is interesting. We note that some challenges will bring the necessary understanding tools and technologies to ultimately design and use them: understanding life, managing complex systems, future information processing technologies, future problem solving. While for others challenges they constitute enabling technologies: robotics, problem solving.

The ubiquity of the self-adaptive computing research fields in all of the grand challenges identified stresses its importance in the future of ICT regardless of the research direction that will be actually taken. The research on self-adaptive computing could have been a grand challenge of its own since it embeds all of the characteristics of the above-mentioned topics: novelty, ambition and interdisciplinary research.

7.9.4 ÆTHER Lessons and Open Issues

The ÆTHER technologies described in this chapter represent a first attempt at trying to implement the required elements of a self-adaptive computing environment. From designing adaptive loops in hardware architectures, specifying runtime system and their protocols up to defining a language construct that tries to represent the interaction of adaptive loops ÆTHER spans the whole chain of techniques and disciplines. The holistic and interdisciplinary approach of the program which has been one of the strong points of the research had the disadvantage of leaving the exploration of some of the most advanced topics to a somewhat shallow level. For example, most of the ÆTHER technologies are just about loops, concurrency and resource management taken from their standard meaning. However as we highlighted at the beginning of this text, self-adaptation is more than just loops, it is more than just the aggregation of known concepts. Self-adaptation is loops plus knowledge, the very expression of self-awareness. This is one of the main issues that remain to be tackled in the future: **embedded knowledge management**.

On the run-time part, we have seen the importance of concurrency management for managing self-adaptation at runtime. The distributing of tasks either sequentially and/or concurrently is key in allowing tasks adaptation between processors. We have shown that it was possible to devise a good enough system for managing concurrency with the proposed SVP. We have shown that it is possible to implement the required virtual processor and specific instructions even with quite different processor architectures (RISC, FPGA). However, with the growing variety and number of heterogeneous processing unit on a chip as highlighted by the ITRS roadmap, it will become increasingly difficult to devise a concurrency control system in all possible cases. And that is probably one of the main issues that will remain: **managing runtime concurrency with very heterogeneous processor architecture**.

Software ideally should provide application designers with sufficient tools and constructs to express their problems within their own field of expertise and their semantics without worrying about the specific details of the underlying computing architecture. This is separation of concerns seen at large. Through the S-NET language we have laid down the basic mechanisms to do this but there is still some work to do in order to **include non-functional features**.

Reconfigurable hardware (FPGA) has long been considered as the ideal enabling technology in order to implement dynamically adaptive hardware circuits. Indeed, the example implementations that we have developed using state of the art runtime reconfigurable devices show that it is technically possible. However this hides all of the engineering effort that need to be done specifically for each application design. And this design effort is way too important when compared with comparable software design times. Again, the ITRS roadmap highlighting this design productivity shows that although current reconfigurable technology could be used to implement run-time adaptive hardware in specific cases. It is nowhere near for general use, unless a dramatic change and breakthrough in Reconfigurable circuits's concepts is made. One clear path for such an advance would be the availability of fully **relocatable hardware circuits within a reconfigurable logic mesh**.

7.10 Conclusion

In its current state the ÆTHER project has laid the foundation of a complete framework for designing and programming computing resources that live in changing environments and need to re-configure their objectives in a dynamic way. Key concepts and technologies have been developed: S-NET, SVP, SANE but a lot of investigation and opportunities for further research remains. In many ways ÆTHER has just scratched the surface of a whole new universe. The research on self-adaptive computing has been mostly focused on investigating solutions that derives from the mainstream computing ecosystem. In many ways the solutions we have investigated, although workable in their present states, show the limits of the "standard" computing technologies when trying to develop self-x/autonomic systems.

A more disruptive approach to self-adaptivity spanning from the language level down to the processing machine implementation should be explored. In order to achieve true self-x capabilities we believe it is necessary to explore and adopt bio-inspired mechanisms, such as neural or genetic models, mechanical and architectural paradigms, such as in bionics or swarm architectures, and to move towards a comprehensive approach that captures the very principles behind the success of biological systems: biology shows everyday that it is a very successful adaptive paradigm derived from ecological, evolutional and, more generally, informational considerations.

References

1. "IBM Research Autonomic Computing." http://www.research.ibm.com/autonomic/
2. J. Von Neumann, *Theory of Self-Reproducing Automata*, University of Illinois Press, 1966.
3. G.E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, 1965.
4. J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, 2003, pp. 41–50.
5. Edsger W. Dijkstra, "On the role of scientific thought", in Dijkstra, Edsger W., Selected writings on Computing: A Personal Perspective, New York, NY, USA: Springer-Verlag New York, Inc., pp. 60–66, 1982
6. Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, "Asynchronous Stream Processing with S-Net", International Journal of Parallel Programming 38(1), pp. 38-67, Springer-Verlag, Dordrecht, Netherlands, 2010.
7. S-Net website, http://www.snet-home.org/
8. David B. Skillicorn, *Foundations of parallel programming*, ISBN-13: 9780521455114, Cambridge University Press, Cambridge, England, 1994.
9. John A. Sharp (Ed.), *Data flow computing*, ISBN:0-89391-654-4, Ablex Publishing Corp., Norwood, NJ, USA, 1992.
10. C.R. Jesshope. A model for the design and programming of multi-cores, in L. Grandinetti, editor, *High Performance Computing and Grids in Action*, **volume 16** of Advances in Parallel Computing, pages 37–55. IOS Press, 2008.
11. A. Bolychevsky, C.R. Jesshope, and V.B. Muchnick. Dynamic scheduling in RISC architectures. IEE Trans. E, Computers and Digital Techniques (**143**):309–317, 1996.
12. Chris R. Jesshope. MICROGRIDS: Foundations for massively parallel on-chip architectures using microthreading. http://www.nwo.nl/nwohome.nsf/pages/NWOP_6DSBSV.
13. ÆTHER: Self-adaptive embedded technologies for pervasive computing architectures. http://www.aether-ist.org.
14. C. R. Jesshope, "μTC an intermediate language for programming chip multiprocessors," in Proceedings of the Pacific Computer Systems Architecture Conference (ACSAC'06), LNCS 4186, 2006, pp. 147–160.
15. Apple-CORE: Architecture paradigms and programming languages for efficient programming of multiple cores. http://www.apple-core.info/.
16. Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, **34**(4):383–427, 2006.
17. Clemens Grelck and Sven-Bodo Scholz. SAC: off-the-shelf support for data-parallelism on multicores. In *DAMP'07*: Proceedings of the 2007 workshop on Declarative aspects of multicore programming, pages 25–33, New York, NY, USA, 2007. ACM.
18. Dimitris Saougkos, Despina Evgenidou, and George Manis. Specifying loop transformations for C2μTC source-to-source compiler. In 14th *Workshop on Compilers for Parallel Computing (CPC'09)*, Zurich, Switzerland. IBM Research Center, 2009.
19. ÆTHER Deliverable D1.1.1, First research report on SANE hardware architecture, issued 31/12/2006.
20. Chris Jesshope, Jean-Marc Philippe, and Michiel Tol, "An Architecture and Protocol for the Management of Resources in Ubiquitous and Heterogeneous Systems Based on the SVP

Model of Concurrency", In Proceedings of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS '08), Mladen Berekovic, Nikitas Dimopoulos, and Stephan Wong (Eds.). Springer-Verlag, Berlin, Heidelberg, 218–228.

21. L. Li, V. Narayanan, M. Kandemir, and M. J. Irwin, "Adaptive Error Protection for Energy Efficiency", In the Proceedings of the International Conference on Computer Aided Design (ICCAD'03), November, 2003.
22. S. Lopez-Buedo, J. Garrido, and E. I. Boemo, "Dynamically Inserting, Operating, and Eliminating Thermal Sensors of FPGA-Based Systems", *IEEE Transactions on Components and Packaging Technologies*, Vol. 25, No. 4, December 2002
23. S. Mondal, R. Mukherjee, and S.O. Memik, "Fine-grain thermal profiling and sensor insertion for FPGAs", Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'06), pp. 4387-4390, 2006
24. Xilinx, "Virtex-5 FPGA System Monitor", November 7, 2008 http://www.xilinx.com/support/documentation/user_guides/ug192.pdf
25. K. Paulsson, M. Hübner, J. Becker, J.-M. Philippe, C. Gamrat, "On-Line Routing of Reconfigurable Functions for Future Self-Adaptive Systems – Investigations within the ÆTHER Project," International Conference on Field Programmable Logic and Applications (FPL 2007), pp.415-422, 27-29 Aug. 2007.
26. Jean-Marc Philippe, Benoit Tain, and Christian Gamrat, "A self-reconfigurable FPGA-based platform for prototyping future pervasive systems", In Proceedings of the 9th international conference on Evolvable systems: from biology to hardware (ICES'10), Gianluca Tempesti, Andy M. Tyrrell, and Julian F. Miller (Eds.). Springer-Verlag, Berlin, Heidelberg, 262–273.
27. L. Zhang and C. Jesshope, "On-Chip COMA Cache-coherence Protocol for Microgrids of Microthreaded Cores", Eds. Bouge et. al., Proc Euro Par 2007 Workshops, LNCS Volume 4854, Springer, pp 38-48, 2007.
28. Martin Danek, Jean-Marc Philippe, Petr Honzik, Christian Gamrat and Roman Bartosinski, "Self-Adaptive Networked Entities for Building Pervasive Computing Architectures", Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, 2008, Volume 5216/2008, 94–105
29. M. Luck, P. McBurney, O. Shehory, S. Willmott and The AgentLink Community, "Agent Technology Roadmap, a roadmap for agent based computing", September 2005
30. The International Technology Roadmap for Semiconductor, ITRS Update 2008, http://www.itrs.net/, 2009
31. G. Agnus et al., "Two-Terminal Carbon Nanotube Programmable Devices for Adaptive Architectures," Advanced Materials, vol. 22, no. 6, pp. 702-706, 2010.
32. Akinwande, D. et al. "Monolithic Integration of CMOS VLSI and Carbon Nanotubes for Hybrid Nanotechnology Applications." Nanotechnology, IEEE Transactions on 7, 636–639 (2008).
33. The Information Society Technologies Advisory Group, "European Challenges and Flagships 2020 and beyond", July 2009.