Towards an Intelligent Environment for Programming Multi-core Computing Systems

Sabri Pllana[1], Siegfried Benkner[1], Eduard Mehofer[1], Lasse Natvig[2], and Fatos Xhafa[3]

[1] University of Vienna, Department of Scientific Computing, Nordbergstrasse 15, 1090 Vienna, Austria
{pllana,sigi,mehofer}@par.univie.ac.at
[2] NTNU, Department of Computer and Information Science, Sem Saelands vei 9, NO-7491 Trondheim, Norway
Lasse.Natvig@idi.ntnu.no
[3] UPC, Department of Languages and Informatics Systems, C/Jordi Girona 1-3, 08034 Barcelona, Spain
fatos@lsi.upc.edu

Abstract. In this position paper we argue that an intelligent program development environment that proactively supports the user helps a mainstream programmer to overcome the difficulties of programming multi-core computing systems. We propose a programming environment based on intelligent software agents that enables users to work at a high level of abstraction while automating low-level implementation activities. The programming environment supports program composition in a model-driven development fashion using parallel building blocks and proactively assists the user during major phases of program development and performance tuning. We highlight the potential benefits of using such a programming environment with usage-scenarios. An experiment with a parallel building block on a Sun UltraSPARC T2 Plus processor shows how the system may assist the programmer in achieving performance improvements.

1 Introduction

While multi-core processors alleviate several problems that are related to single-core processors - known as *memory wall*, *power wall*, or *instruction-level parallelism wall* - they raise the issue of the *programmability wall*. On the one hand, program development for multi-core processors, especially for heterogeneous multi-core processors, is significantly more complex than for single-core processors. On the other hand, programmers have been traditionally trained for the development of sequential programs, and only a small percentage of them have experience with parallel programming.

Additionally, there is a portability problem. In the past programmers could trust that compilers succeeded to pass the increased computing power of next processor generations without high porting effort. This was due to relatively homogeneous processor designs even from different hardware vendors with instruction level parallelism (ILP) supported at hardware level. The architectural

E. Luque et al. (Eds.): Euro-Par 2008 Workshops, LNCS 5415, pp. 137–147, 2009.

change to multi-core processors, however, affects the programmer in several ways. On the one hand, thread level parallelism (TLP) must be exploited effectively and efficiently. In general, this cannot be done automatically by a compilation system, but requires assistance by the programmer. On the other hand, multi-core architectures differ significantly requiring that applications must be adapted to the various platforms.

While in the past only a relatively small group of programmers interested in HPC was concerned with the parallel programming issues, the situation has changed dramatically with the appearance of multi-core processors on commonly used computing systems. Traditionally parallel programs in HPC community have been developed by *heroic programmers*[1] using a simple text editor as programming environment, programming at a low-level of abstraction, and doing manual performance optimization. It is expected that with the pervasiveness of multi-core processors parallel programming will become mainstream, but it can not be expected that a mainstream programmer will like to become a HPC hero.

In this paper we argue that the programming productivity of multi-core[2] systems is increased if an intelligent programming environment would be available that (1) enables the programmer to work during the process of program development at a higher level of abstraction using domain-specific modeling languages in a model-driven development fashion; and (2) provides context-specific knowledge and performs iterative time-consuming tasks involved in program development in a semi automatic/autonomic manner (for instance, performance tuning). We propose a parallel programming methodology that combines model-driven and agent-supported program development with the use of high-level parallel building blocks. The goal is to increase programming productivity without restricting flexibility and creativity, allowing the programmer to fully use his/her intellectual capacity for software design at model-level. Although software development is considered to be an art, we anticipate that there are many implementation activities that can be performed more automatically/autonomically.

The rest of this paper is organized as follows. Section 2 describes our vision for programming of multi-core computing systems. We illustrate our approach experimentally in Section 3. Section 4 reviews the state-of-the-art in programming multi-core computing systems. We conclude the paper with a summary and future work in Section 5.

2 Intelligent Programming of Multi-core Systems

In this section we outline our methodology and the corresponding environment for programming multi-core systems.

[1] Andrea: "Unhappy is the land that breeds no hero." Galileo: "No, Andrea: Unhappy is the land that needs a hero." – Bertolt Brecht in *Life of Galileo*.

[2] Although some authors have introduced the term *many-core* to denote multi-core systems with many cores (i.e. 100 or more), we will stick to the more established term multi-core. We do not see a need to make a distinction between multi- and many.

2.1 Methodology

Our parallel programming methodology combines model-driven agent-supported program development with the use of high-level *parallel building blocks* (PBB). We propose to address the complexity of programming multi-core systems as follows:

- Raise the level of abstraction at which the programmer performs most of the activities during the process of software development, by using a model-driven development approach combined with PBBs;
- Support the programmer during the software development, by using intelligent software agents for providing context-specific knowledge and automation of iterative activities involved in software development and optimization.

Model-Driven Development (MDD) [13]. MDD is a software development method that advocates to *first model* a program and *then build* the program code. It is inspired by mature engineering disciplines such as *civil engineering*, where before an artifact (for instance a *bridge*) is built first the corresponding model is developed. In software engineering the models are usually described graphically using the Unified Modeling Language (UML). The model should preferably describe the program at an abstraction level that is independent from a specific platform. Models may be used to study the functionality and the performance of the program before the program code for a specific platform is developed. MDD has the potential to reduce software development time and complexity, by using tools for automatic model-to-code transformation and thereby reducing the programmer's effort for manual coding. Since multi-core architectures differ significantly from each other, a significant effort is required to adapt (that is *port*) programs to the various platforms. Since MDD captures the program logic as a platform-independent model, then program models remain largely unaffected from the changes in processor architectures. In our previous work we have developed an extension of UML for the domain of performance-oriented parallel/distributed programs [16] and the corresponding tool-support *Teuta* [10]. Teuta allows to build models of parallel programs, enrich them with performance-related information, and generate various textual representations (such as XML or C++).

Parallel Building Blocks. The PBBs are inspired from research in programming concepts such as *skeletons* [1,2,8] or *dwarfs* [3]. Basically, PBBs may be thought of as program-independent generic programming units that support software re-usability. A set of parameters is used to specify the functionality of a PBB in the context of a certain program. For instance, as parameter may serve the program-specific code (that is the code that PBB requires to perform the expected functionality in the context of a certain program). PBBs may be implemented for instance using *C++ Templates* or *Java Generics*. Parallelism is described within the PBB, and therefore the programmer is not exposed directly to the parallel programming complexity (such as dealing explicitly with the *communication and synchronization* among processing units or *deadlock avoidance*).

Commonly various combinations of PBBs may be used for solving a certain problem. In the context of programming environments PBBs lend themselves to an increased level of automation of various activities such as program transformation, code generation, performance optimization, and resource usage optimization. In our previous work, in the context of MALLBA project [1], we have developed a library of parallel skeletons (such as *branch and bound*, *metropolis*, *simulated annealing*, *genetic algorithms*, or *tabu search*) for solving various optimization problems.

Intelligent Software Agents. Software agents are programs that are *reactive*, *proactive*, *autonomic*, and *social* [21]. Software agents that have *learning* and *adapting* abilities are known as *intelligent software agents*. *Reactiveness* indicates the ability to respond adequately to changes in the context in which it operates. A *proactive* program performs activities to achieve a specific goal based on its initiative (it does not wait passively for a request of another entity to perform a certain activity). *Autonomy* indicates the ability to perform activities independently of user intervention in order to achieve a specific goal. *Social* programs are able to communicate and coordinate activities with other programs (that is agents). A program is considered intelligent if it is able to learn from the previous experience (for instance, via trial-and-error or generalization) and is able to adapt accordingly to the perceived changes in the environment. We have a vision about several intelligent software agents cooperating with each other and the programmer during the process of program development. Our vision is based on the idea that the programming environment should be better at helping the programmer as a more active partner. In our previous work, in the context of the AURORA project [4], we have used intelligent software agents to automate systematic performance analysis for parallel and distributed programs. Although software development is considered to be an art, we anticipate that there are many implementation activities that can be performed more automatically/autonomically using intelligent software agents.

In the following sub-section we propose a programming environment for multi-core computing systems that uses MDD, PBBs, and intelligent software agents.

2.2 Programming Environment

The proposed programming environment comprises a set of intelligent software agents that may help to automate the programming process at several levels. Some agents will advice the composition of programs using PBBs, while others will guide the exploration of different possible parallel strategies, load balancing and performance optimization (see Figure 1).

The programming environment provides the programmer with information feedback useful in the process of developing a program for a multi-core system. This information is collected at several levels, from program composition to information about resource usage (such as the cache behavior) obtained by execution or simulated execution. Also, information is exchanged between the agents at the system level in an automated manner continuously looking for ways of obtaining and improving knowledge about the performance of the program

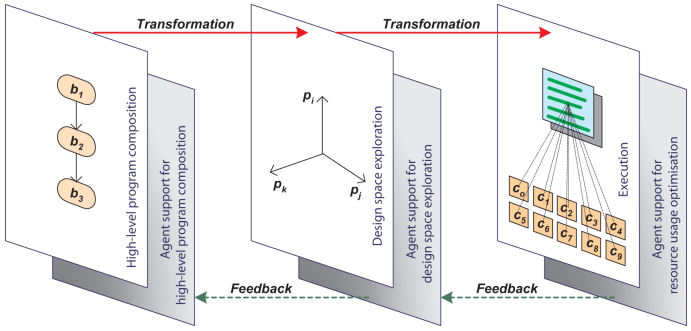


Fig. 1. Agent supported program development. The programming environment comprises multiple intelligent software agents that support program composition, design space exploration and resource usage optimization.

being developed. In this way, a parallel program with good performance can be developed with high programmer productivity.

In what follows in this section we highlight the major program development and tuning phases: (i) high-level program composition, (2) design space exploration, (3) resource usage optimization.

High-level Program Composition. This phase deals with the composition and coordination of PBBs. The granularity of PBBs may range from frequently used programming idioms, to larger patterns or dwarfs [3]. High-level descriptors are used to capture the main parallelization aspects of PBBs and serve as interface to agents in the design space exploration phase. The user composes the program graphically using a UML extension for multi-core systems.

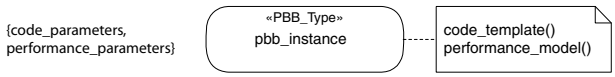


Fig. 2. UML representation of a PBB

The UML may be extended by defining new modeling elements, *stereotypes*, based on existing elements (also known as *base classes* or *metaclasses*). Stereotypes are notated by the stereotype name enclosed in guillemets `<<Stereotype Name>>`. Figure 2 depicts the graphical representation of a PBB. `<<PBB_Type>>` indicates the kind of PBB. With a PBB is associated the corresponding *parametrised code and performance model*. Parameters determine the behavior of the PBB instance in the context of a specific program.

The programming environment assists the user proactively during the program composition. For instance while the user is loading some old BLAS code for some dense linear algebra operations – the *composer agent* interrupts and suggests using the PBB for dense linear algebra tailored for efficient execution on

multi-core systems. Additionally, it may offer a list of other PBBs that often are used together with this one, as well as presenting typical compositional patterns in a graphical way.

Design Space Exploration. High level discrete-event simulation is used for rapid model-based performance evaluation of programs, using a hybrid method that combines mathematical modeling with high level discrete-event simulation [15].

For instance, after the completion of the program composition phase the programming environment may suggest to the user doing some high level rapid design space exploration. The estimated performance of various possible program implementations is presented by a *visualization agent*. While the user is studying the graphs, and gets some ideas for improvement, the programming environment is also analyzing the results and suggests changing some of the parameters in one of the PBBs (such as the parallelization granularity), and to perform some more detailed simulations for getting better knowledge of the performance that can be obtained with different task allocation and scheduling policies.

Resource Usage Optimization. Instruction-level simulation is used for more detailed studies of the utilization of shared resources such as shared on-chip memory and off-chip bandwidth. For instance, in [9] an efficient utilization of the shared cache resources has been found to have great affect on multi-core performance. This is integrated with the use of performance counters. A performance monitoring agent provides information about the state of the system (resource characteristics and usage). Instruction-level simulation is time consuming (may take several hours or days), and therefore should run in background. When finished, the findings will be propagated upwards back to the higher level performance models, as a model calibration process. It is a systematic way of bringing performance information from the execution (or simulated execution) environment back to the development environment. Please note that this kind of optimization is architecture-dependent.

For instance, the user may get hints from the programming environment for changes that will improve performance of the program. The programming environment may offer some detailed simulations at the instruction level, and helps the user to select those simulation experiments that are likely to be the most relevant. For instance, if higher-level simulations show that some of the processor cores were waiting for data for long periods, a more detailed study of the on-chip shared memory resources should be done.

3 Example

In this section we illustrate how best practices from HPC combined with agent based program development offer new opportunities to obtain efficient solutions.

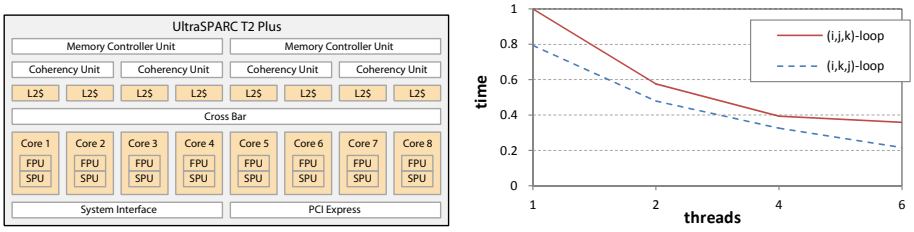
PBBs allow a programmer to specify various parallelization strategies together with the code and a first guess for individual parameters which are subject to the tuning process. This follows our assumption that only semi-automatic

parallelization is reasonable. The programmer specifies the main strategies for parallelizing the code and the system explores this restricted optimization space to generate efficient code. Two factors back up this approach. First, rich analysis work has been done in the past by the HPC community, including the authors institutions (Vienna Fortran Compilation System [6]), which can be reused. Second, in the past the strong emphasis on the target-code performance and manual performance tuning resulted in low programming productivity. The increasing importance of development of economically viable software nowadays reveals opportunities for semi-automatic parallelization, even at the price of achieving lower performance compared to a hand-tuned version.

In our example we use as hardware platform the Sun UltraSPARC T2 Plus, co-denamed Niagara-2, multi-core processor (shown in Figure 3(a)) which is an SMP extended version of the T2 allowing multiple Chip-level MultiThreading (CMT) processors to be used within a single system. The T2 Plus was presented in April 2008 and has up to 8 cores per processor with 8 hardware threads per core resulting in a maximum number of 64 threads per processor or logical CPUs as reported by the operating system. T2 Plus offers only poor support for instruction-level parallelism emphasizing thread-level parallelism. Two integer units are provided per core with four threads sharing one unit, and one FPU is provided per core with all eight threads sharing it. The L1 data cache has 8 KB per core and the on-chip L2 cache offers 4 MB which are shared between the cores.

In what follows in this section we present an example scenario to illustrate the agent-supported software development cycle. Different forms of PBBs are possible, but in the simplest case a PBB can be some loop nest together with data layout and work distribution annotations. Consider e.g. an application written in C consisting of a series of PBBs with one of them denoting a floating point matrix-matrix multiplication, i.e. `C[i,j] = C[i,j] + A[i,k] * B[k,j]` with loop nest (i,j,k). As parallelization strategy the programmer specifies that the elements of result matrix C should be assigned to processor cores in a row-wise manner and calculated by them. Since the target architecture is a Sun T2 Plus with 8 cores and 8 FPUs, the programmer specifies that the rows shall be assigned to 8 threads.

When submitted to the *design space exploration agent* and its analysis framework (cf. [6,7]), the framework detects poor spatial cache locality and performs



(a) Sun UltraSPARC T2 Plus. (b) Performance improvements.

Fig. 3. Processor block diagram and optimization results

loop interchange resulting in loop nest (i,k,j). Then the code is split up in 8 threads as suggested by the programmer and assigned to the 8 cores of T2 Plus and executed. The monitoring component of the *resource usage agent* reveals low memory bandwidth utilization and low FPU utilization for this PBB and reports this feedback information to the agent. The *resource usage agent* is aware of the hardware characteristics of T2 Plus and knows about the hyper-threading (HT) technology provided by this kind of architecture with up to 8 hardware threads. Therefore the agent suggests to use HT technology to increase FPU utilization and reports to the *design space agent* to explore possibilities to increase the number of threads. Consequently, the *design space agent* proposes to assign the rows of result matrix C to 2, 4, 6 hardware threads per core resulting in a total number of 16, 32, 48 threads, respectively. Three versions are generated and submitted for execution. Moreover, feedback information is used by the compilation system to perform further optimizations (cf. [11]).

The key point is that this time-consuming tuning task is done automatically by the system and not by the programmer. The different versions are automatically generated and run on T2 Plus and the monitoring results are reported back to the agents and the programmer. Figure 3(b) shows the normalized execution times (longest execution time denoted by time unit 1.0) for the different versions with 1, 2, 4, 6 threads per core and the improvements achieved by the optimizations taking programmer annotations and hardware characteristics into account. The performance improvement of loop interchange is considerable and amounts to 26% for 1 thread per core, approx. 20% for 2 and 4 threads per core, and 66% for 6 threads per core. The performance improvement for increasing the number of threads per core to deal with memory latency is even more significant. The performance improvement assigning 2 and 4 threads to one core was for both loop nest versions approx. a factor of 1.7 and 2.5, respectively. For 6 threads per core we got for (i,j,k) loop nest a factor of 2.8 and for (i,k,j) loop nest up to 3.7. Based on this experience, the *resource usage agent* classifies increasing the number of threads to deal with memory latency as valuable optimization which has proven beneficial for this processor. The programming environment may suggest this kind of optimization for similar processor architectures as well.

4 A Review of the State-of-the-Art

An increasing number of research projects is addressing the challenge of programming multi-core computing systems. The *Habanero project* [12], which started in Fall 2007 at Rice University, aims to develop languages and compilers for the development of portable software for multi-core systems. The *SALSA project* [19] at Indiana University is investigating the use of services as building blocks for composing parallel data-mining applications based on the workflow paradigm. *Linked Sequential Activities* in SALSA, which are conceptually based on Communicating Sequential Processes of Hoare, are used to build services. The *Berkeley View* [3] project investigates the influence of multi-core processors in applications, hardware, programming models, and systems software for parallel computing. The

Berkeley View proposes to use a set of *dwarfs* (a dwarf defines a specific computation and communication pattern) for evaluation of parallel programming models. The recently established *Pervasive Parallelism Laboratory (PPL)* [17] at Stanford University is investigating future parallel computing platforms. PPL is supported by six computer and chip makers that are convinced that their product sales may decline if software is not able to use effectively the new multi-core-based hardware. *SWARM* [5], developed at Georgia Institute of Technology, is a parallel programming framework that provides a collection of primitives for programming multi-core processors. The *Programming Environments Laboratory (PELAB)* [18] at Linköping University is investigating the applicability of round-trip engineering techniques to parallelization of sequential programs. The *Cell Superscalar (CellSs)* [14] project at Barcelona Supercomputing Center focuses on parallelization of sequential programs for Cell BE processor. The CellSs parallelization involves the functional decomposition, code annotation and the use of a source-to-source compiler. The *IT Research Division of the NEC Laboratories Europe* [20] is investigating the use of work stealing concept to achieve load balancing.

In contrast to the related work we propose an intelligent programming environment that proactively supports the user during major phases of program development and performance tuning by providing context-specific knowledge and performing iterative time-consuming tasks involved in program development in a semi automatic/autonomic manner.

5 Conclusions

We have outlined an intelligent programming environment, which proactively supports the user during high-level program composition, design space exploration, and resource usage optimization. We have highlighted the potential benefits of using such a programming environment with usage-scenarios.

We have observed that even for a rather simple parallel building block such as matrix multiplication the exploration of the parameter space may be time prohibitive on one hand, but on the other hand there is a big potential for performance improvement. The example scenario described a first and manageable step towards an intelligent program environment for multi-core architectures. Several projects at the authors' home institutions are currently pursued towards the realization of such an intelligent programming environment for multi-core computing systems.

References

1. Alba, E., Almeida, F., Blesa, M., Cabeza, J., Cotta, C., Diaz, M., Dorta, I., Gabarro, J., Leon, C., Luna, J., Moreno, L., Pablos, C., Petit, J., Rojas, A., Xhafa, F.: MALLBA: A Library of Skeletons for Combinatorial Optimisation (Research Note). In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, p. 927. Springer, Heidelberg (2002)

2. Alind, M., Eriksson, M., Kessler, C.: BlockLib: A Skeleton Library for Cell Broadband Engine. In: International Workshop on Multicore Software Engineering (IWMSE 2008) at ICSE 2008, Leipzig, Germany, May 2008. ACM, New York (2008)
3. Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., Yelick, K.: The Landscape of Parallel Computing Research: A View from Berkeley. EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, December 18 (2006)
4. AURORA: A Priority Research Program on Advanced Models, Applications and Software Systems for High Performance Computing (1997–2007), http://www.vcpc.univie.ac.at/aurora/
5. Bader, D., Kanade, V., Madduri, K.: SWARM: A Parallel Programming Framework for Multi-Core Processors. In: First Workshop on Multithreaded Architectures and Applications (MTAAP) at IPDPS 2007, Long Beach, CA, USA, March 2007. IEEE, Los Alamitos (2007)
6. Benkner, S., Andel, S., Blasko, R., Brezany, P., Celic, A., Chapman, B., Egg, M., Fahringer, T., Hulman, J., Kelc, E., Mehofer, E., Moritsch, H., Paul, M., Sanjari, K., Sipkova, V., Velkov, B., Wender, B., Zima, H.: Vienna Fortran Compilation System - Version 1.2 - User's Guide. Technical report, Institute for Software Technology and Parallel Systems, University of Vienna (February 1996)
7. Benkner, S.: VFC: The Vienna Fortran Compiler. Scientific Programming 7(1), 67–81 (1999)
8. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing 30(3), 389–406 (2004)
9. Dybdahl, H., Stenström, P., Natvig, L.: A cache-partitioning aware replacement policy for chip multiprocessors. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297, pp. 22–34. Springer, Heidelberg (2006)
10. Fahringer, T., Pllana, S., Testori, J.: Teuta: Tool Support for Performance Modeling of Distributed and Parallel Applications. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3038, pp. 456–463. Springer, Heidelberg (2004)
11. Gupta, R., Mehofer, E., Zhang, Y.: Profile Guided Code Optimizations. In: Srikant, Y.N., Shankar, P. (eds.) The Compiler Design Handbook: Optimizations & Machine Code Generation. CRC Press, Boca Raton (2002)
12. Habanero Multicore Software Project, http://www.cs.rice.edu/~vs3/habanero/
13. Model Driven Architecture, http://www.omg.org/mda/
14. Perez, J., Bellens, P., Badia, R., Labarta, J.: CellSs: Making it easier to program the Cell Broadband Engine processor. IBM Journal of Research and Development 51(5), 593–604 (2007)
15. Pllana, S., Benkner, S., Xhafa, F., Barolli, L.: Hybrid Performance Modeling and Prediction of Large-Scale Computing Systems. In: 2008 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS 2008), Barcelona, Spain, March 2008. IEEE CS, Los Alamitos (2008)
16. Pllana, S., Fahringer, T.: On Customizing the UML for Modeling Performance-Oriented Applications. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, p. 259. Springer, Heidelberg (2002)
17. Pervasive Parallelism Laboratory, http://ppl.stanford.edu/wiki/index.php/Pervasive_Parallelism_Laboratory

18. Programming Environments Laboratory (PELAB), http://www.ida.liu.se/labs/pelab/
19. Service Aggregated Linked Sequential Activities (SALSA), http://www.infomall.org/multicore/
20. Wagner, J., Jahanpanah, A., Träff, J.: User-Land Work Stealing Schedulers: Towards a Standard. In: 2008 International Workshop on Multi-Core Computing Systems (MuCoCoS 2008) at CISIS 2008, Barcelona, Spain, March 2008. IEEE CS, Los Alamitos (2008)
21. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley & Sons Ltd., Chichester (2002)