Paper accepted for Third International Workshop on Software Development and Integration in Robotics SDIR 2008 (SDIR-III), Pasadena, CA, May 20th, 2008

PROTOTYPING PLANT CONTROL SOFTWARE WITH MICROSOFT ROBOTICS STUDIO

Alwin Hoffmann, Florian Nafz, Frank Ortmeier, Andreas Schierl and Wolfgang Reif
Department of Software Engineering and Programming Languages
University of Augsburg, Augsburg, Germany
Email: {*alwin.hoffmann, nafz, ortmeier, reif*}@informatik.uni-augsburg.de, andreas.schierl@student.uni-augsburg.de

Abstract— **The development of industrial robotics applications is a complex and often a very expensive task. One of the core problems is that a lot of implementation and adaptation effort can only be done *after* the robotic hardware has been installed.**

This paper shows how Microsoft Robotics Studio (MSRS) can facilitate the fast prototyping of novel industrial applications and thus lower the overall development costs. Microsoft Robotics Studio is a development tool for creating software for robotic applications. It includes an asynchronous, service-oriented runtime and a realistic physics-based simulation environment. This allows for testing and improving software prototypes *before* any hardware is installed.

As an example control software for a vision of tomorrow‘s production automation systems has been implemented and evaluated in the simulation environment of MSRS.

I. Introduction

During the last two decades the complexity of automated production processes has steadily increased. This led on the one hand to a dramatic increase in productivity but on the other hand to a steep rise in development costs for the production system. A major part of these costs are costs for development of control software. This is mainly (but not only) because software components have to be either developed in isolation and subsequently need to be integrated after the whole plant has been installed (this approach is typically very expensive as it often requires many changes and error corrections) or the software is only developed *after* the plant has been installed (this approach means that the installation time for the whole plant is increased). In most industrial scenarios a mixture of both approaches is taken.

A new third approach can be to develop software in a simulated environment of the plant. This allows for early start of the software development process as well as early testing and validation. Such an approach requires two ingredients: first of all a simulation environment in which the plant can be easily modeled and second the control software - for the simulated plant – must be easy to adapt from controlling a simulated world to controlling the physical plant and its robots. A framework which promises to fulfill both requirements is Microsoft Robotics Studio [1]. It is a service-oriented framework for software development of robotic applications. Instead of using real hardware, software can be developed within a simulated environment using a realistic physics engine which has been developed for computer games. Furthermore, the service-oriented architecture allows the simulated hardware to be easily substituted with real hardware and enables the development of modular applications which are orchestrated from low-level services.

This paper shows that it is possible to use Microsoft Robotics Studio for prototyping control software for industrial robotic applications. It also describes benefits, potentials, risk and limitations of this approach. The results are demonstrated on an illustrative example of a vision of tomorrows automated production systems.

In Sect. II a brief introduction to Microsoft Robotics Studio and its main technologies is given. Sect. III describes an example scenario, its design and implementation The last Sect. IV summarizes the results and gives an overview of possible next steps.

II. Microsoft Robotics Studio

In this section, a short summary of the Microsoft Robotics Studio is given. For more detailed information see the online documentation at [1]. The Microsoft Robotics Studio, released in December 2006, is a Windows-based development environment for developing robotics applications for a variety of hardware platforms. The target groups are academic, hobbyist and commercial software developers. It includes a lightweight asynchronous, service-oriented runtime, a visual programming environment as well as a realistic 3D physics-based virtual environment for the simulation of robotics software. The Microsoft Robotics Studio runtime consists of two main components: the Concurrency and Coordination Runtime (CCR) and the Decentralized Software Services (DSS).

The Concurrency and Coordination Runtime [2] is a code library accessible from any language targeting the .NET 2.0 Common Language Runtime. It provides a concurrent, message-based programming model for service-oriented applications. The CCR manages asynchronous operations, deals with failure scenarios and enables concurrency without the use of manual threading and synchronization. Software components are loosely coupled within the runtime, as they only interact through asynchronous messages.

The central component of the CCR is a *Port*, which represents a typed FIFO queue of items and is used as a point of interaction between any two software components. A port can only accept items of its designated type. Several independent port instances can be grouped into a *Portset* allowing for a component to accept different item types. Basically, a

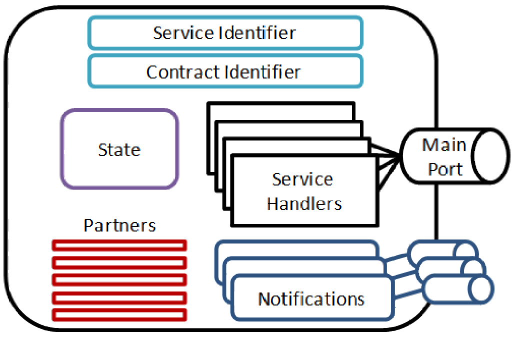

Fig. 1. Main components of a service in the DSS application model [1]

portset defines the interface of a software component within the CCR.

Usually, items posted on a port are not directly processed. Instead, an arbiter can be registered for each port to determine how items are handled. An arbiter can, for example, simply receive these items and pass them to a method. However, arbiters can even be composed to express complex coordination patterns (e.g. joins, choices and interleavings) and as a result facilitate the coordination between multiple operations within a component. After the arbiters for a port are determined, they have to be activated. As items are posted on a port, the activated arbiters decides how to process the item and schedules it for execution. Note that this is the source of asynchronous behavior within the CCR.

The Decentralized Software Services (DSS) runtime provides a distributed service-oriented application model on top of the CCR. An application is a composition of services where each service has a state and a set of operations over that state including support for event notification and structured data manipulation. By orchestrating these services either programmatically or via the Visual Programming Language, complex applications can be created and composed anew to create even more complex applications. Moreover, the DSS runtime provides a hosting environment for services (DSS node) and a set of infrastructure services for service creation, discovery, logging, debugging, monitoring and security.

Microsoft Robotics Studio also introduces the SOAP-based Decentralized Software Services Protocol (DSSP) [3]. Its operations are intended to be a superset of the methods defined by HTTP (e.g. `Delete`, `Get`, `Post` and `Put`) providing support of structured data manipulation and event notification. Due to a different protocol characteristics, DSSP is designed to complement HTTP. Hence, DSS can use both DSSP and HTTP for communicating with services. DSSP defines operations for service state retrieval (e.g. `Get`) as well as for modifying the state (e.g. `Insert` and `Update`). Furthermore, a service can receive event notification by subscribing to another service using the `Subscribe` operation.

A DSS service instance is always created and executed within the context of its hosting environment, a DSS node. Figure 1 shows the main components of a service in the DSS application model. The *Service Identifier* is a URI and refers to a particular instance of a service running on a particular node. The *Contract Identifier* is also an URI and refers to the contract of the service. A contract is a short description of the service behavior facilitating the composition and reuse of services. The *State* is the representation of a service at a given point in time and describes its current contents. To compose more and more complex applications, services must be able to interact with each other in an efficient and deterministic way. Hence, the DSS application model offers the concept of partner services. Specifying a set of *Partners* implies that a service interacts with and possibly depends on these services in order to operate properly. Partner services are specified by service identifiers and can either run on the same DSS node or across the network. The *Main Port* is a CCR portset where messages from other services arrive. The accepted messages are determined by the type of the main port and must be either operations defined by HTTP or DSSP. For every valid operation on the main port, a *Service Handler* must be registered to handle the incoming message. In the DSS application model a service implementation interacts with another service by sending messages to its main port. Additionally, a service can subscribe to other services and will be notified when a particular event has occurred. For each subscription, a service will receive *Notifications* on a separate CCR port.

Furthermore, Microsoft Robotics Studio includes a visual 3D physics-based simulation environment which supports the creation of advanced robotics scenarios without the need of expensive hardware. The simulation environment includes the AGEIA PhysX Technology [4] for enabling real-world physics. So the simulation environment allows programmers to easily prototype robotics applications with real-world physics and test software before deploying it on hardware.

The simulation environment is also implemented as a service and must be partnered with in order to use the simulation. If an application adds the simulation engine service as a partner, a simulation window will automatically open when the application is started. Objects representing hardware and physical objects in the simulated world are called entities and consist of both a graphical 3D representation and a physical model. The first is a complex mesh setting up the detailed appearance, whereas the latter only approximates the mesh with simple shapes and is used for physical calculations and collision detection. Entities can be linked with services in order to be controlled from outside the simulation environment. Usually, entities like robot arms, carts, or sensors are linked with a service and therefore are controlled by the services.

III. Prototyping Plant Control Software

In this section, we show how Microsoft Robotics Studio can help for prototyping industrial robotic applications. In the first part of this section a vision of tomorrows production systems is presented. The second part shows challenges in developing such a system and how they can be solved. The last part of this section details how MSRS helped in rapid prototyping the control software, how it supported evaluation of quality of the control system and elaborates some experiences we made.

A. *Case study*

Traditionally production automation systems are very static in their nature. Process flow is fixed during design/installation time and optimized for maximum efficiency. While this approach is very useful for mass production it still has several drawbacks. The two most important ones are lack of failure tolerance and a high effort for adaptation to new production processes. This in general makes production automation on rarely available for small series.

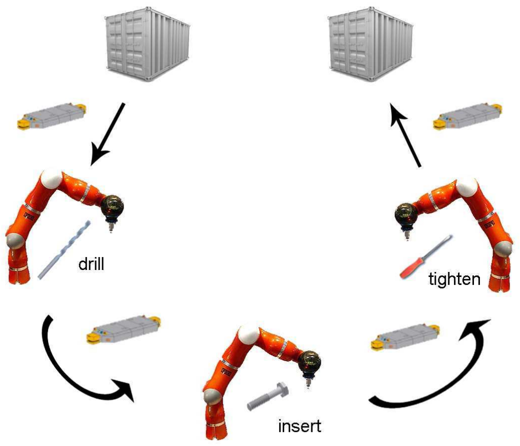

Fig. 2. Standard role allocation of the production cell

A new idea is to build "organic" production systems. The idea of organic computing [5] and autonomic computing[6] is to build system which show "organic" behavior. Organic means in this context, that the system automatically adapt to changes in their environment. These capabilities are called self-healing (adapting to failures), self-organizing (working jointly to solve a task) or self-adapting (changing and adapting to new jobs and tasks). In the context of production automation an (organic) adaptive production cell which for example autonomously reconfigures itself after component failures or adapts dynamically to new tasks.

In this paper an adaptive production cell consisting of three KUKA Lightweight Robots (LWR), four autonomous carts and two storages is considered (see Fig. 2). The goal is to process workpieces according to a given workplan. In the example, the workplan is to drill a hole, to insert a screw into this hole and then to tighten the screw (short: DIT). Each of the LWRs is capable of using any of the three required tools (Driller, Inserter and Screwdriver). The workpieces are transported through the cell with autonomous carts. As changing the tool of a robot requires a lot of time, the standard configuration is to let the robots specialize and transport the workpiece from robot to robot. This situation is depicted in figure 2.

If a failure occurs (e.g the drill of the drilling robot breaks) then a traditional production system would come to a standstill. It is obvious that this behavior is not really

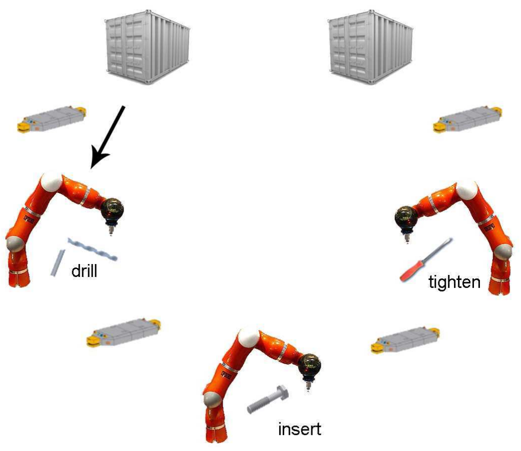

Fig. 3. A failure occurs

necessary as the affected robot could simply specialize for some other job. An adaptive production cell would now try to automatically find such a solution, reconfigure itself and start operation again (these situation are shown in Fig. 3 and in Fig. 4).

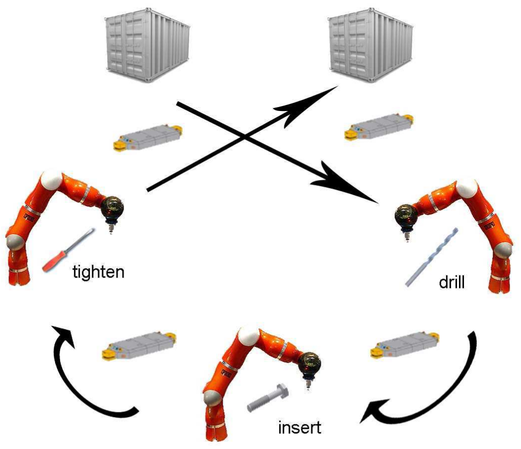

Fig. 4. Reconfigured production cell

Similar reconfigurations are easy to imagine, which can cope with new robots/carts or changes in workpieces. It is worth mentioning that also failure analysis and verification was applied at this model. A more detailed report on the application, especially the safety and self-healing related questions may be found in [7][8].

B. *Modelling and Design*

The last part described a vision. It did not outline how such a system can be built. It is clear, that designing a system with all the mentioned capabilities will be more difficult than design a traditional static control system. In [9] an approach for design and construction of organic computing systems is described. For lack of space we will not go into the details of this approach in this paper. the only important point for now is, that many organic computing systems can be split into two parts: one part which provides the basic functionalities (functional part) and one part which comprises the “organic” part of the system. In the example the functional part consists of the actual control software of robots, carts and storages: i.e. point-to-point movements, opening/closing grippers, changing tools etc.

The organic part of the system is a planning component, which constantly monitors the systems components and its environment. Whenever it detects anomalies or problems, it searches for possible solutions to re-achieve the goals of the system. This is basically done by reconfiguring components (i.e. changing specialization of robots, reassign transportation routes etc.). For more details on design, construction and analysis of such organic control algorithms, we refer to [10], [7].

Assume now, that a process for construction of such an organic algorithm is available and assume further that this algorithm has been implemented. The open question is now. “Will this proposed algorithm really work for the production cell?”. The answer of this question is not easy by any means. The problem is, that besides the actual computations of reconfigurations a lot of meta knowledge about the system has to be considered. For example if transportation routes are reallocated, then it must be assured that carts won‘t collide on these new routes. It must also be assured, that if carts approach the robots from different directions, the robots then recognize this and are still able to locate and take workpieces for processing. These and other problems can only be considered if either real hardware is used or if a simulated environment is available. MSRS allows for very realistic physical simulation of the hardware.

C. Implementation

The adaptive production cell as in Sect. III-A (together with a reconfiguration algorithm) has been modelled in Microsoft Robotics Studio. The main application is implemented as a DSS service, which sets up the simulation environment by partnering with the simulation engine service. This includes the initialization of the 3D scene by populating the simulation world with entities as well as the creation of required services. The required entities and its positions are read from an XML file facilitating different spatial configurations. According to the service-oriented application model, the main application uses further services to represent the production cell. These services are implemented as modular and reusable DSS services for Microsoft Robotics Studio.

The robots are based on and reusing the KUKA LWR3 arm and gripper implementation from the KUKA Educational Framework [11], a set of services implementing KUKA robots for Microsoft Robotics Studio. These low-level services (open and close gripper, change arm angles) are controlled by the `CellRobot` service that uses transformation and motion planning services from the KUKA Educational Framework to provide high-level functions for the simulated robots on its main service port. Apart from the default DSSP operations `Get` and `Update` that allow to retrieve and replace the current service state thereby enabling serializing and saving the service state as well as stopping and resuming service operations, the `CellRobot` service contains a `Configure` method to set the current task and the currently supported capabilities, allowing the user to cause errors in the simulation environment. Furthermore, there are `Take` and `Drop` to make the robot take a workpiece from a cart standing below it or drop an already taken workpiece to another cart. The `Process` operation is used to tell the robot to perform one of its capabilities on the taken workpiece, and `Reset` asks the robot to return to a controlled initial state (e.g. to restore a safe state after an error). `Take` and `Drop` are implemented much like conventional robot programs as a sequence of linear and point-to-point movements and gripper commands, whereas the `Process` action is abstracted to only changing the color of the workpiece for better visualization.

The cart implementation in `CellCart` uses a differential drive with two wheels. It is based on the service `SimulatedDifferentialDrive` from the KUKA Educational Framework with tiny changes in shape and color and reduced height to increase stability. Its main operation `Drive` is used to command the cart to a certain position. Therefore, the cart first rotates toward its destination by applying opposite forces to both wheels. As the carts are symmetric and do not distinguish between front and rear, this can always be achieved by rotating less than 90 degrees. Afterwards, they drive forward or backward until they have reached the expected position. The drive operation is finished after rotating to the target direction.

The workpieces are simply boxes with notches to allow the robot gripper to take them firmly. For the simulation, they are created and destroyed at the storages, and their color tells what jobs have already been done on them. A video demonstration of the cell including reconfiguration can be found at [12].

Applications outside Microsoft Robotics Studio can control the simulated services using either an HTTP interface or a connector class based on it. The functionality includes the creation and deletion of workpieces, the movement of the carts as well as controlling the robots. Moreover, it allows to control the environment to for e.g. enable or disable capabilities of the robots (more on this topic in the next paragraph).

So far, simulation services described previously – as well as their operations – could be part of a traditional production cell. Their cooperation could be achieved by implementing a simple, inflexible controlling service. However it is now possible to simply wrap the organic reconfiguration algorithm into an DSS and use it to dynamically (re-)configure the robots.

In order to be controlled by an organic reconfiguration service, all simulation services implement wrapper operations to provide a common interface. The operation `OrganicConfigure` is called to inform services that they were assigned a new task (handled by robots to display a new current task using the mentioned `Configure` operation). Together with implementations of `OrganicStop` and `OrganicReset` – based on the `Reset` operations if required

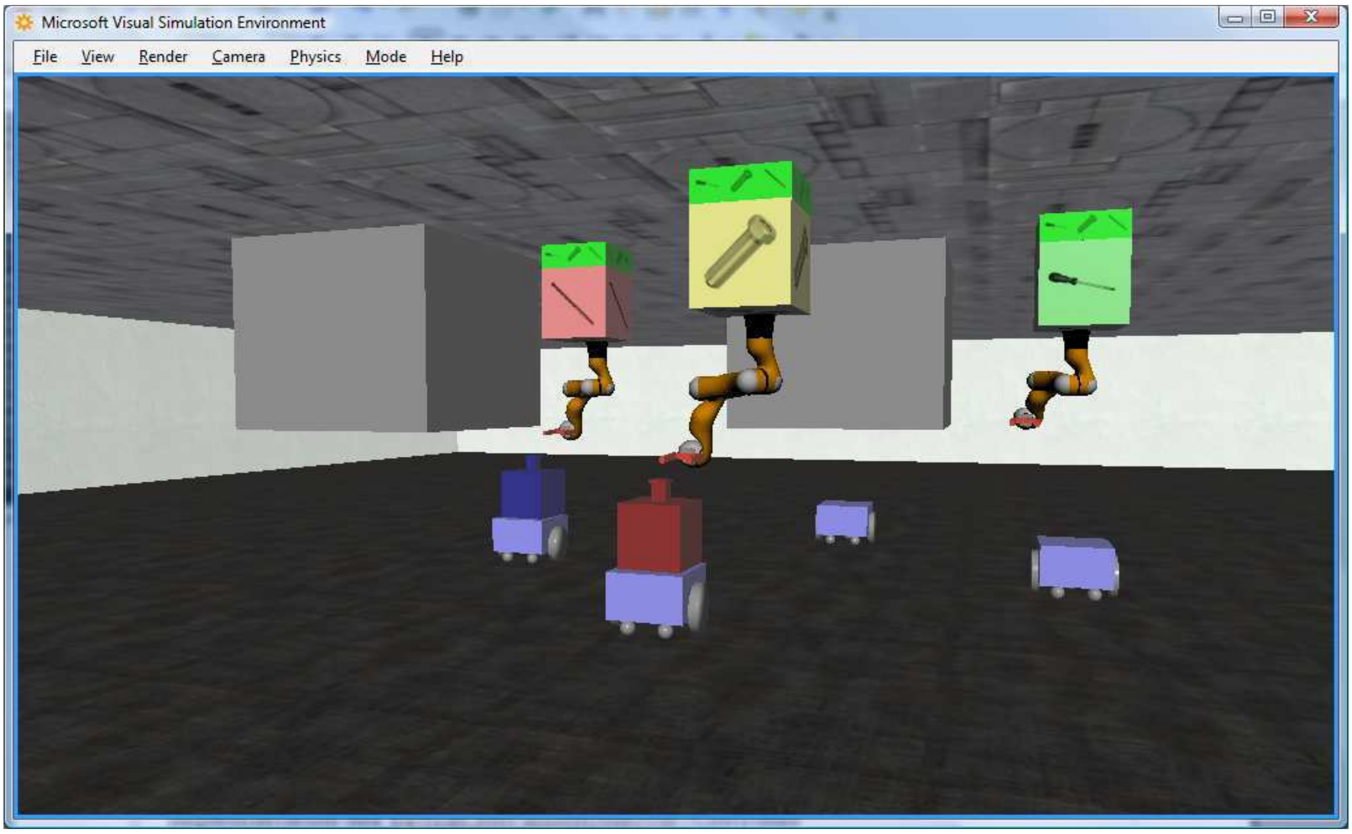
Microsoft Visual Simulation Environment
File View Render Camera Physics Mode Help

Fig. 5. Simulation of the adaptive production cell with MSRS

– these allow the services to be reconfigured in a safe and controlled manner when a problem with the current configuration occurs. Implementations of `OrganicCommand` provide a common interface for robot or cart specific operations, e.g. `Drive` or `Take`, described by the the name of the operation and the partner to cooperate with, e.g. the robot or storage to drive to for `Drive`). [1]

Building a working simulation and programming the mentioned wrappers took less than one man-week. Developing a suitable reconfiguration algorithm is easy at the first look. But only the simulation can show how good the high level algorithm work for the actual example and show where physical limitations have to be taken into account.

D. Simulation

For simulation the robots, storages and carts are placed in a production hall with robots and storages mounted at the ceiling (see Fig 5). The robots contain a display showing the the current task and available capabilities and can process workpieces that are placed on carts below them. The placement of the actors in the production cell is configured using an XML file, so different numbers or locations of actors can easily simulated by changing the configuration file.

The simulation is started by running the application `RobotCellSimulation` which itself is a DSS service running inside a DSS node. It initializes the further required services for 3D and physics simulation as well as for the simulated robot cell. Afterwards it creates the adaptive controller services to start processing in the production cell.

To test and demonstrate the self-configuration abilities of the adaptive controlling agents, a control GUI using the HTTP interface is implemented that allows users to trigger failure of a robot's tool. This allows for testing the reconfiguration algorithm and its interplay with real (simulated) hardware.

We made the experience, that in this scenario reconfiguration works fairly good. However, the rapid prototyping approach led to some additional points of synchronization between robots, storages and carts, which were not immediately obvious when the reconfiguration algorithm has been designed. It also turns out, that for more complex geometries carts have to be enhanced, such that collision free transportation can be guaranteed.

IV. Conclusion

We already started studying the presented example of an adaptive productive cell in 2005 in the context of research in the domain of organic computing. In this context the adaptive production cell was only one of the case studies we used for evaluation of our method.

The topic of the work presented in this paper was now: "How would one make use of organic computing principles and methods for building an production automation system?"

[1]Note, that instead of implementing the wrapper operations in the simulation services, another service layer could be introduced. Due to simplification, we have abstained from doing so.

For this task MSRS was a great help. The main reason is, that only a realistic, physical simulation is precise enough for checking the usability of control software for production automation. This is because there exist a lot of implicit dependencies between components (like the acceleration of a cart and the mass of the transported objects), which must be taken into account. MSRS here really helps a lot. We also made the experience, that building a model of the hardware and its environment is really easy. The control software can then be implemented as a single (or also multiple) service(s). This is very beneficial for software re-use and composition.

On the other hand it also turned out, that there exists a variety of limitations. Computing power and only a limited number of robots is one of them. However, more important (and more difficult to estimate) is the accuracy of the simulation. As a matter of fact MSRS use different algorithms for planning robot motions than real KUKA robots. It is also not possible to directly use the developed control software as real robots are not (yet) controllable by DSS services. So this will be part of our future research.

Summarizing, we think the MSRS is a very good tool for developing and rapid prototyping robotic applications. The presented example is only one of the possible application domains. But it shows all challenges which a traditional control software would also face (and some additional ones). It turned out, that using MSRS‘s simulated environment allows for faster and cost effective development of software, but can not replace evaluation and testing on real hardware.

REFERENCES

[1] Microsoft Corporation. (2007) Microsoft Robotics Studio Development Center. [Online]. Available: http://msdn.microsoft.com/robotics/

[2] G. Chrysanthakopoulos and S. Singh, “An asynchronous messaging library for C#,” in *Proceedings of OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

[3] H. F. Nielsen and G. Chrysanthakopoulos, *Decentralized Software Services Protocol - DSSP/1.0*, 2007. [Online]. Available: http://purl.org/msrs/dssp.pdf

[4] AGEIA Technologies Inc. (2007) AGEIA PhysX. [Online]. Available: http://www.ageia.com/physx/index.html

[5] C. Müller-Schloer, C. von der Malsburg, and R. P. Würtz, “Organic computing,” *Informatik Spektrum*, vol. 27, no. 4, pp. 332–336, Aug. 2004.

[6] “IBM autonomic computing home page,” http://www.research.ibm.com/autonomic/.

[7] M. Güdemann, F. Ortmeier, and W. Reif, “Formal modeling and verification of systems with self-x properties,” in *Proceedings of the Third International Conference on Autonomic and Trusted Computing (ATC-06)*, ser. Lecture Notes in Computer Science, L. T. Yang, H. Jin, J. Ma, and T. Ungerer, Eds., vol. 4158. Berlin/Heidelberg: Springer, Sept. 2006, pp. 38–47.

[8] M. Güdemann, F. Ortmeier, and W. Reif, “Safety and dependability analysis of self-adaptive systems,” in *Proceedings of ISoLA 2006, 2nd Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. IEEE CS Press, 2006.

[9] H. Seebach, F. Ortmeier, and W. Reif, “Design and Construction of Organic Computing Systems,” in *Proceedings of the IEEE Congress on Evolutionary Computation 2007*. IEEE Computer Society Press, 2007, accepted for publication.

[10] M. Güdemann, F. Nafz, W. Reif, and H. Seebach, “Towards safe and secure organic computing applications,” in *INFORMATIK 2006 – Informatik für Menschen*, ser. GI-Edition – Lecture Notes in Informatics, C. Hochberger and R. Liskowsky, Eds., vol. P-93. Bonn, Germany: Köllen Verlag, Sept. 2006, pp. 153–160.

[11] KUKA. (2007) KUKA Educational Framework. [Online]. Available: http://www.kuka.com/en/products/software/educational_framework/

[12] University of Augsburg. (2007) Advanced production cell demonstration. [Online]. Available: http://www.informatik.uni-augsburg.de/de/lehrstuehle/swt/se/projects/organic_computing/