OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

Developing self-organizing robotic cells using organic computing principles

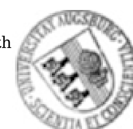
Alwin Hoffmann, Florian Nafz, Andreas Schierl, Hella Seebach, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Hoffmann, Alwin, Florian Nafz, Andreas Schierl, Hella Seebach, and Wolfgang Reif. 2011. "Developing self-organizing robotic cells using organic computing principles." In Bio-Inspired Self-Organizing Robotic Systems, edited by Yan Meng and Yaochu Jin, 253–73. Berlin: Springer.
https://doi.org/10.1007/978-3-642-20760-0_11.

Nutzungsbedingungen / Terms of use: licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under th conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/



Chapter accepted for Bio-Inspired Self-Organizing Robotic Systems © 2011 Springer-Verlag Berlin Heidelberg
The final authenticated version is available online at https://doi.org/10.1007/978-3-642-20760-0_11

Chapter 1
Developing Self-Organizing Robotic Cells using Organic Computing Principles

Alwin Hoffmann, Florian Nafz, Andreas Schierl, Hella Seebach, and Wolfgang Reif

Abstract Nowadays industrial robotics applications, which are often designed and planned with a huge amount of effort, have a fixed behavior during runtime and cannot react to changes in their environment. Failures can hardly be compensated and often can only be repaired by human involvement. The idea of Organic Computing is to enable systems to possess life-like properties, such as self-organizing or self-healing. In this chapter we present a layered architecture to bring these two worlds together. Further it is discussed what are the requirements of the respective layers to allow to engineer self-x properties into such systems. The presented approach allows for developing self-organizing robotic applications that are able to take advantage of Organic Computing principles and therefore are more robust and flexible during runtime.

1.1 Introduction

With respect to their structure, traditional automation systems are very static. The material flow is fixed and every component is optimized according to the planned system structure to reach maximum throughput. This approach is very suitable for mass production as in the automotive industries, where one product is manufactured for a considerable time. Even the use of industrial robots does not change this situation. In fact, industrial robots are very flexible and, given an appropriate tool, are able to perform a large variety of tasks [10]. However, the complex and tedious programming of today's industrial robots, the fixed wiring and difficult integration of additional devices, as well as the very static layout of shop floors do not exploit the possible flexibility of robotic solutions. As a consequence, high effort is needed to customize and adjust automation systems, making them hardly applicable

Alwin Hoffmann · Florian Nafz · Andreas Schierl · Hella Seebach · Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg, 86135 Augsburg, Germany

for small-series production with varying products where regular adaptation is required. Moreover, failure tolerance and flexible optimization is hard to achieve with traditional automation systems.

On the other hand, the idea of Organic Computing [3, 20, 39] and Autonomic Computing [6, 15] is to develop systems which possess self-x properties, like self-healing (i.e. the compensation of failures), self-optimization (i.e. the autonomous optimization according to a given fitness function), or self-adaptation (i.e. the adaptation to new or changing tasks or a different system structure). In the context of production automation, the goal is to provide architectures and techniques to build *organic* automation systems, where organic means life-like behavior or more concise that the systems are capable of autonomously adapting to changes in their environment. This behavior is often realized by the use of bio-inspired paradigms and algorithms, e.g. genetic algorithms or pheromone-based approaches.

Hence, the well-designed combination of Organic Computing principles and robot technology can lead to hyper flexible and robust automation systems. While robots – mobile platforms as well as industrial manipulators – provide mechanical flexibility and the ability to perform a large variety of different tasks, Organic Computing introduces self-organization to the systems which enables them for self-healing, self-optimization or self-adaptation. For example, an Organic automation system can compensate failures due to its self-healing capabilities and continues to operate in *graceful degradation* – a requirement for robotic systems which is getting more and more important [11]. Especially in small and medium enterprises, where there are phantom shifts at night, systems of this kind are welcome. Using self-optimization allows for continuously and autonomously optimizing an automation system during its runtime and without external interaction. Finally, due to self-organization, organic systems are reconfigurable by design and are easily able to adapt to new tasks or products – a requirement in today's globalized economy with its turbulent markets and fast changing demands [17]. However, evolutionary approaches and bio-inspired principle which solely rely on the idea of emergence cannot directly be applied to production systems. Emergence rather has to be controlled and directed to accomplish a defined goal, i.e. manufacturing a product.

In this chapter, we want to outline how self-x properties and Organic Computing principles can be applied to industrial robotic manufacturing cells [2]. In previous work, we have introduced an approach for modeling and designing self-organizing resource-flow systems based on Organic Computing principles [31] and showed how to specify a behavioral corridor for this system class [8]. Moreover, we have developed a guideline for systematically engineering self-organizing resource-flow systems [30] and verified their functional correctness using formal methods [23].

Because robotic manufacturing cells usually constitute a resource-flow system, where a product (i.e. the resource) is manufactured step-by-step, our approach can be applied to the domain of industrial robotics. However, we identified three robotic-specific challenges one is facing in order to build self-organizing robotic cells. These challenges are ranging from uncontrolled emergence over the problems in the layout of robotic cells to limitations in current software architectures and will be described in detail in Section 1.2. Based on these challenges, we present a multi-layer architec-

ture (Section 1.3) which allows for developing self-organizing robotic manufacturing cells using Organic Computing. Every layer addresses the system at a different level of abstraction and has distinct responsibilities in order to comply with the architectural requirements. In Section 1.4, we illustrate our approach with a simple but evident case study. Finally, in Section 1.5, a conclusion is drawn and future research steps are highlighted.

1.2 Challenges

From our point of view, the development of self-organizing robotic cells using Organic Computing principles poses three major challenges:

1. To render organic systems acceptable for industry, *emergence must be controlled* to accomplish a defined goal.
2. To apply self-x properties, the layout of robotic cells must provide *additional degrees of freedom*.
3. To utilize these additional degrees of freedom, robotic software architectures must provide *flexibility with regard to programming techniques*, coping with geometric uncertainty and device integration.

These challenges and their influence on the development of organic automation systems are described in detail in the following sections.

1.2.1 Controlling Emergence

A main concept of self-organizing systems is emergence. Emergence describes the appearance of complex system behavior caused by relatively simple and local interactions of individuals without the control of a central instance. Hence, the system behavior is not explicitly programmed, but a result of these local interactions. An example of emergent behavior is an ant colony where no central control is present. Instead, each ant is an autonomous unit that reacts depending on local information, i.e. pheromones, and genetically encoded rules.

Thus, the behavior of the individual components cannot be exactly predicted. Müller-Schloer [18] calls this kind of behavior *bottom-up constraint propagation* which stands in contrast to the classical top-down design of technical systems. In the latter approach, the developer tries to model and implement all possible system states. This usually starts with a high-level specification, until after a number of transformations and refinements, executable code is generated.

However, having an exhaustive model of a complex system is often not feasible and even contradictory to the idea of emergence. In order to solve this contradiction, we suggest defining a corridor of *good* expected behavior [8] for every organic production system. Inside this corridor, emergent behavior is approved and

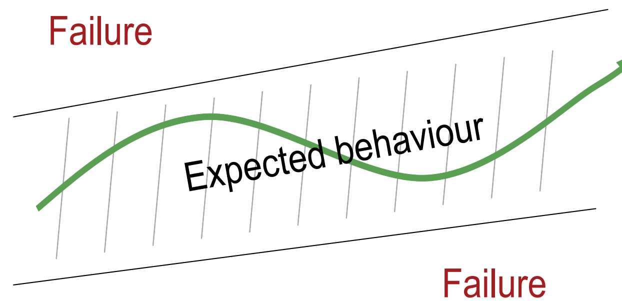


Fig. 1.1 Organic production systems require a corridor of expected behavior. Inside this corridor, emergent behavior is approved and even desired.

even desired, whereas the system is in an exceptional state when this corridor is left (cf. Fig. 1.1).

This corridor is defined through constraints by the system developer and allows *controlled emergence*. Usually, these constraints can be observed locally by each autonomous component of the system. If one or more constraints are violated, the component tries to restore the constraints locally. If this is not possible, it starts to involve surrounding components until a valid solution is found that satisfies the constraints. In Organic Computing this kind of architecture is called an Observer/Controller architecture [18, 31]. By doing so, organic automation systems are self-organizing and can be directed to accomplish a defined goal, e.g. to manufacture products. Besides, behavioral guarantees in terms of functional correctness can be given [21].

1.2.2 Adding Degrees of Freedom

Usually, automation systems are designed and tuned to accomplish pre-defined tasks for a long period. In single-station automated cells, a production machine is typically equipped with a material handling system (e.g. a robot for loading and unloading the machine) and a storage system. Due to this setting, the cell is able to operate unattended but the system fails if any of the components breaks. An automated production line consists of multiple workstations that are automated and linked by a transport system which transfers parts from one station to the next. Again, if one components breaks, the whole system fails. According to [40], flexible manufacturing systems still have limited capabilities regarding customized products and failure compensation. They even state that the need for flexible products and adaptive systems cannot be supplied with traditional approaches.

In order to become self-organizing, automation systems require additional degrees of freedom and redundancy in the available hardware. Without these prerequisites, the system is not able to adapt to new environmental conditions or to compensate failures:

- For *self-healing*, an organic automation system needs redundant hardware components. Otherwise, it cannot compensate for the failure of one component and continue operation in graceful degradation.
- Regarding *self-adaptation*, an organic production system needs degrees of freedom, i.e. flexible tools or transport systems, in order to adapt to changing or new tasks as well as to a modified system structure.
- Finally, *self-optimization* is only possible if there are several degrees of freedom which can be optimized with respect to a given fitness function.

Due to these reasons, we believe that robotic cells are well-suited for self-organization by using Organic Computing. In robotic-based systems, additional degrees of freedom can be achieved by adding robots, redundant tools, or tool-changing systems. Concerning transportation, robots can be connected using carousels, two-way conveyors, or even mobile platforms. Further details are given in Sect. 1.3, but here it is worth mentioning that the concrete choice of how redundancy is added can impact the system's robustness and its mean time to failure, as the example in Sect. 1.4 shows. Giving one component all redundancy is in general a bad choice, as a component failure will lead to a complete loss of the available redundancy. To find good distribution strategies for redundancy, the ADCCA[1] technique [9] can be used, which calculates minimal combinations of failures which lead to a standstill of the whole production system. Also similar safety analysis techniques like Fault Tree Analysis [37] can be used to identify single-point or n-point failures and optimize the redundancy distribution accordingly.

1.2.3 Requiring Software Flexibility

By adding degrees of freedom and redundancy to the available devices and to the shop floor layout, self-organization becomes feasible. However, to completely utilize self-x properties, additional requirements to the architectures of robotic systems with regard to software flexibility are necessary.

Flexible and reconfigurable automation systems require the introduction of *smart products* carrying information about how to be processed by the system. This can be e.g. realized by using RFID [40]. As a consequence, a *product-centric* approach of configuring and commanding industrial robots and their tools is required. Predefined motion sequences have to be replaced by more dynamic motion planning considering the environment and avoiding obstacles. Due to the dynamic system behavior, the use of previously taught motions cannot be sufficient anymore. Instead, the use of sensor feedback (e.g. vision) or compliant devices should be considered. With sensor-based or compliant motions [16], an error-tolerant execution of complex robot tasks in uncertain and unknown environments is possible [34].

In contrast to these demanding requirements, industrial robots are still programmed with special robot programming languages which are derived from early

[1] Adaptive Deductive Cause-Consequence Analysis

imperative languages and have not evolved much since then. Due to these low-level programming techniques, developing software for an industrial robot is a complex and tedious task requiring considerable technical expertise [26]. Hence, industrial robots are usually equipped and programmed to perform only a set of pre-defined tasks. This contradiction between low-level programming and high-demanding requirements must be solved in the future to realize self-organizing robotic systems.

Furthermore, the integration of external devices must be facilitated. Today, tools are usually connected by a fixed wiring to a robot controller and communicates over digital and analog I/O ports. But when using tool changing systems, no human interaction should be required. The software of the robot controller must be able to independently cope with different tools mounted to the robot. Moreover, it must be possible to integrate arbitrary sensors for intelligent perception and sophisticated tools that allow e.g. complex grasping strategies and dexterous manipulation [24]. The introduction of plug-and-play mechanisms as proposed in [11] would cover this requirement for flexible device integration.

1.3 Architecture

Our approach uses a layered software architecture which addresses the system at different levels of abstraction. The proposed architecture is depicted in Fig. 1.2.

On top of the hardware layer, two software layers are located for controlling the robot. The lower one, the *Robot Control Layer*, is responsible for the real-time critical, low-level hardware control, whereas the upper layer, the *Robot Programming Layer*, is used for defining the control flow and specifying required motions and tools actions. For traditional production systems, these three layers are sufficient, as they allow the robot to execute arbitrary, pre-programmed tasks in a reliable, repeatable fashion. However, to extend the system towards self-organization, additional communication and control software is required.

Therefore, our approach adds two more layers on top, which control the robotic system according to Organic Computing principles. The first layer, the *Organic Control Layer*, wraps the components of the robotic cell and turns them into software agents coordinating with other agents through communication. Furthermore, it is responsible for the execution of *capabilities* which are to be applied to the workpiece. When this layer detects a locally unrecoverable error, the *Organic Planning Layer* takes control and searches for a new configuration to achieve the task. Once a solution has been found, control is returned to the Organic Control Layer for further execution. The layers of the architecture are explained below from bottom up.

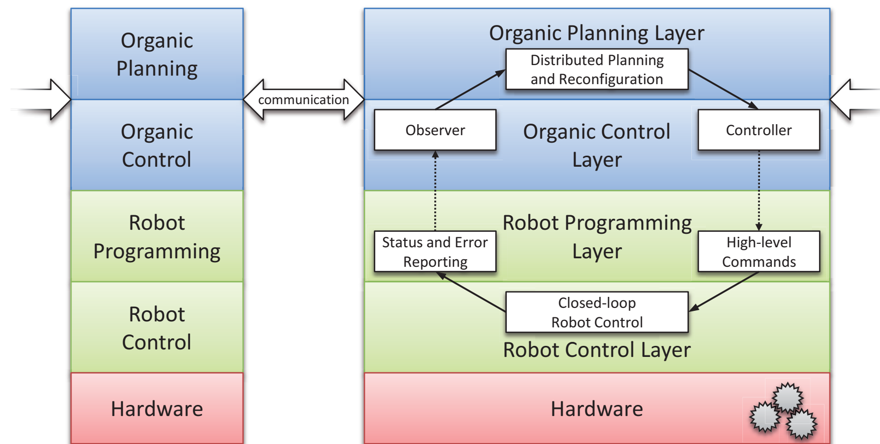


Fig. 1.2 The proposed architecture for self-organizing robotic cells showing two individual components.

1.3.1 Hardware

The foundation of each robotic cell is a set of robots with tools that are interlinked with a transportation system. In order to become a self-organizing production system, additional degrees of freedom are required as stated in Sect. 1.2.2. This means that a robot cannot only be equipped with one static tool corresponding to its preassigned task. For simple cases, it might e. g. be enough to equip the robot with a set of equal drills so that it can replace them when they fail during production, but for exploiting all advantages of self-organizing systems, different tools are needed that can perform a variety of diverse tasks, and a way to interchange them without human interaction. This can be achieved by the use of external tools, by an automatic tool exchange system, or by using advanced tools like anthropomorphic hands which allow dexterous manipulation.

If different tasks have to be executed, or the different task steps should be assigned to different robots, the transportation system also has to become flexible. Instead of a single conveyor connecting the robots in a given order, this set-up requires a way to change the order a workpiece passes the different robots. Similar to existing systems, robots can be connected using a carousel or two conveyors, one moving forward and one moving backwards. Thus, each robot can forward the workpiece to any other robot by placing it onto the right conveyor. Corresponding to the idea of hyper flexible manufacturing systems [11], another solution is to replace the conveyor by a set of mobile platforms navigating between the robots, transporting partly processed workpieces.

This allows a system to show a dynamic behavior. However, as the hardware devices are expected to perform different tasks over time, all of them have to be

controlled by a computer-based system. Its software must provide real-time guarantees to reliably control the hardware devices.

1.3.2 Robot Control Layer

Low-level hardware control is performed in the Robot Control Layer. It is responsible for applying open or closed loop control laws on actuators and sensors in order to make the hardware execute the requested actions. Therefore it has to be implemented in a real-time capable environment, e. g. running on a micro controller for simple actions or under a real-time operating system (such as VxWorks, QNX or real-time extensions for Linux). For commercial KUKA robot systems, this layer – the so-called *kernel system* – is implemented in VxWorks. It can execute motion commands and send data to attached tools using fieldbus communication. However, it is quite limited with respect to sensor integration or compliant motion – fields where research robot controllers like OROCOS [32] are more advanced. Furthermore the control layer has to monitor the attached hardware for errors, and report them to the above layer to allow reasonable failure strategies.

As a typical robot action consists of more than the application of one single control law with given parameters (e. g. one motion to a point), the robot control layer has to provide an interface allowing to specify multiple control laws or commands that are to be applied sequentially or in parallel. This interface can be used by the programming layer. Examples for action task descriptions specified over such an interface are manipulation primitives [5], constraint-based task specifications [4] or realtime primitive nets [38].

Realtime primitive nets describe actions executed by one or multiple cooperating robots. These actions are composed of calculation primitives (blocks) and data flows between them, which are evaluated in a real-time loop and form the corresponding control loop for the hardware devices. All actions that have to be executed with exactly given timing constrains or depend on each other's progress can thus be combined into one realtime primitive net that will be executed atomically. This allows to specify complex or composed tasks with realtime requirements as a single transaction and execute them in the control layer, removing the need for real-time capability in the higher layers.

A limited example for a realtime primitive net is given in Figure 1.3. It shows a motion of a robot along a given trajectory, followed by a control action enabling the gas flow of a welding torch. The dotted boxes represent the high-level constructs used in the programming layer to define the task, whereas the solid boxes show realtime primitives and their dataflow links. In this example, position values from the trajectory generator are sent to the robot block as set points (a typical example of open loop control), and the digital output representing the welding torch is enabled immediately once the trajectory has finished. Of course, a real world welding task consists of more actions to be included in the realtime primitive net, e. g. ignition of the welding torch, going along the welding seam and disabling the torch once the

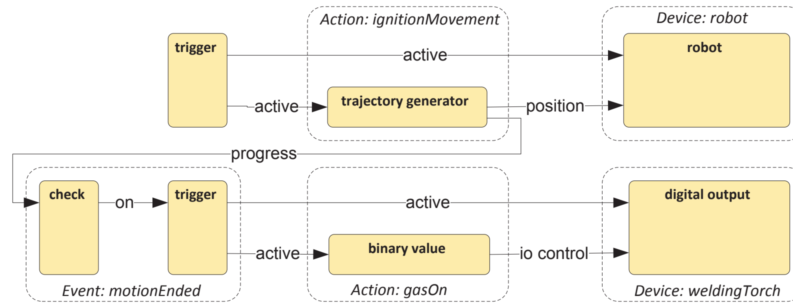


Fig. 1.3 An example realtime primitive net describing the motion of a robot followed by the execution of a tool action.

destination has been reached. Further details about the realtime primitives interface can be found in [38].

Commercial robot controllers usually omit a clear separation between control and programming layer and execute complete robot programs on the control layer. However, by separating these layers and thereby encapsulating the realtime requirements on the control layer, a standard programming language can be used for the programming layer. This allows making the programming layer extensible and simplifies the integration of robot programs into the surrounding software system.

1.3.3 Robot Programming Layer

The Robot Programming Layer offers an interface which accepts high-level commands to be executed by the robot. It is responsible for transforming them into control laws or task descriptions that can be executed with real-time guarantees in the robot control layer. Furthermore, it transfers them to the robot control layer and monitors execution progress, errors and sensor events. For KUKA robots, this layer can be seen in the robot programming language KRL which allows writing robot programs including extended control flow (e.g. conditional statements and loops), motions and tool commands. Similar features are available in the languages RAPID for ABB and Karel for FANUC robots.

However, the self-organizing robot cell – opposed to traditional production cells – does not have a fixed processing or material flow order, thus it is not possible to write one program for each robot that can be executed repeatedly to perform the unchanging robot task. Each robot needs a set of robot programs (one for each robot capability) that can be started and controlled from a higher architecture layer.

As the dynamic nature of a flexible production system makes it hard to guarantee exact positioning of the workpieces during transportation, these systems also have to cope with greater uncertainty about object locations. Thus the integration of sensor feedback for object localization becomes more important here, as well as the

possibility to program tolerant or compliant manipulators or tools. Also dynamic motion planning with obstacle avoidance for both robots and mobile platforms must be possible using this layer.

When trying to control a flexible production cell through a set of individual robot programs (one for each robot capability), these programs as well have to be flexible and highly configurable as described in Sect. 1.2.3. However, passing detailed environment information to traditional robot programs is often quite complex, involving fieldbus communication, thus limiting the range of possibilities [13]. These problems can be solved by using a robot control architecture that allows programming robots in standard, high-level programming languages, such as the one described in [1].

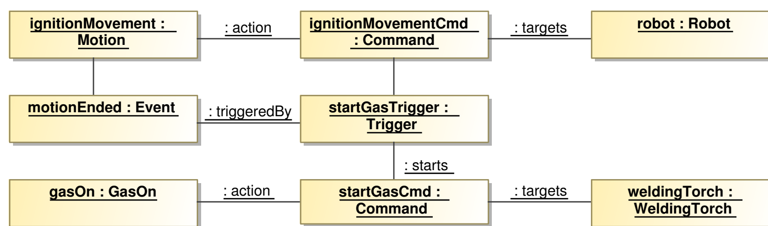


Fig. 1.4 The robot task from Fig. 1.3 represented as an object structure with actions, devices and events.

It provides a high level, object-oriented API for programming robots which can be directly used from the higher layers or encapsulated into a service that can be e.g. accessed via standard service-oriented methods.

Figure 1.4 shows an example of a robot task created using the object-oriented API. It contains two robot commands, one targeted at a robot and containing a motion, and the second enabling the gas flow of a welding torch. These commands are connected using a trigger that starts the second command once the motion of the first command has ended. This (and many more, when dealing with a real welding scenario) has to be executed with real-time guarantees to ensure that the welding seam is created with repeatable quality, so it is converted into a realtime primitive net (like the one shown in Fig. 1.3) and executed using the robot contorl layer.

As an overview, the two robot software layers are shown in figure 1.5. The upper part (programming environment and Robotics API) of the robot software architecture represents the programming layer, and the lower part (realtime primitives interface and robot control core) is an example for a robot control layer. Using this architecture, all layers above can communicate with the robot system using the standard means of object-oriented software development. This simplifies the development of the two organic layers located on top of the robotic layers in the proposed software architecture.

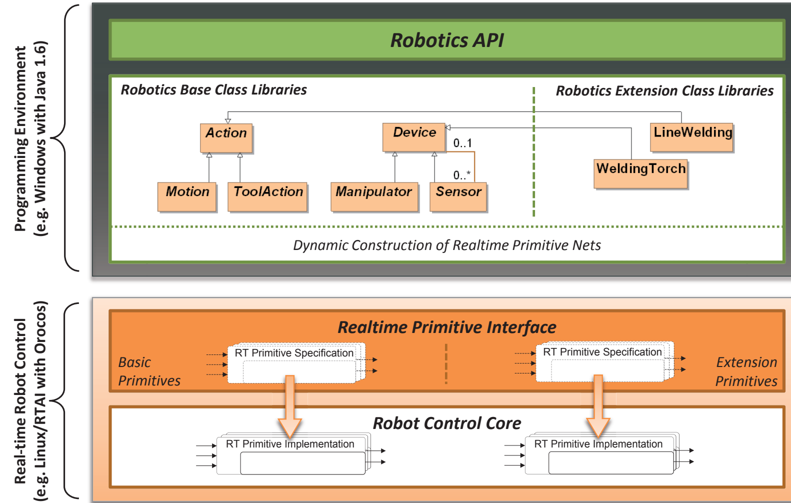


Fig. 1.5 Overview of the robot software layers.

1.3.4 Organic Control Layer

The presented architecture for the top two layers is similar to observer/controller architectures often used in the field of Organic Computing to realize the self-x features of a system [19, 28]. The main task of these layers is to maintain the behavioral corridor of the system (see Sect. 1.2.1). The corridor is specified by OCL Constraints [25, 31], which are annotated to the particular models during the design process and describing "good" system configurations leading to functionally correct behavior. By not explicitly forcing the system into a fixed set of configurations an additional degree of freedom is gained, in which the system can pick the configuration it assesses as good. Further the constraints ensure that only configurations are chosen that lead to a functional correct system. These constraints define a kind of invariant over the system state and distinguish good from erroneous states. They specify how correct configurations of the robots must look like. The system then tries to preserve these constraints as long as possible. In case of a violation information is forwarded to the planning layer which tries to restore them, by calculating a new reconfiguration for the system. This approach is called the *restore invariant approach* and described in detail in [8].

The Organic Control Layer therefore consists of two main components. An *observer* component which constantly evaluates the constraints, based on the status information of the system it receives from the lower layers. Here interfaces which allow to receive feedback from the Robot Programming Layer (e.g. example error-messages or sensor data) are needed. Whenever the observer detects a violation, it activates the planning layer and forwards all gathered information. The main challenge here is to formulate the constraints in such a way and granularity that the

robot is able to locally decide whether a constraint is violated or not. As constraints usually are only violated if a system failure occurs, the observer must be able to reason about the impact of a system failure on the constraints. Here a failure analysis [37, 9] can detect the possible failures of the system, which can impact the validity of a constraint. For example, considering the constraint that a robot must have the tool needed to perform the roles assigned to it. A tool failure will than lead to a constraint violation in case the robot has a role assigned where it needs this tool.

The second component in this layer is the *Controller*. It performs the capabilities assigned by the planning layer and commands robot actions required to apply the capabilities and exchange resources. It makes use of the interface provided by the Robot Programming Layer and controls the robot to ensure that the right capability is applied. It further reacts to new configurations sent by the Organic Planning Layer, for instance to change the performed actions of the robot.

1.3.5 Organic Planning Layer

On top of the control layer is the Organic Planning Layer. It is triggered by the observer of the control layer and responsible for calculating new configurations if an error occurs. It analyzes the current situation and, as most of the failures cannot be compensated by one robot alone, it has to communicate with the planning layers of other robots to gather information about available robots and their capabilities. Then, the planning component tries to find a common solution to reach the objectives. After a consensus is found, the planning layer forwards the new configuration for its responsible robot to the Organic Control Layer, which then commands the robot accordingly.

The advantage of moving all the self-organization into this high-level layer is to be able use the full bandwidth of planning approaches, like bio-inspired or genetic algorithms as well as simple planners. Therefore this layer provides a plug-in interface to allow the use of several methods and algorithms for coordination and planning, implemented as centralized or decentralized variants. System architects can choose what is best suited for their kind of system and problem to solve. Another reason is that on this layer real time must not be considered, as all real time critical commands are dealt with on a lower layer.

The planning task is basically a constraint satisfaction problem on the systems configurations [36, 22]. Depending on the application and the parameters, this can be rather complex, especially as the robots do not have global knowledge. Here the challenge is to find proper communication protocols and algorithms which can deal with specific requirements of the application. One may think of a simple gathering of the global knowledge at one robot and then calculating a new configuration on this robot. The solution is then spread to the other robots. While this may applicable for smaller manufacturing cells it is not for larger systems. Further one does not want to always stop the complete cell, instead a local reconfiguration is preferred, where only a few robots are participating in the reconfiguration process.

Usually not just any solution for the problem is wanted, but an optimal solution for the actual situation. Here the planner's task is extended to find a best or nearly optimal configuration for the system according to the given optimization criteria, load balancing criteria or minimum number of reconfigured robots, for example.

1.4 An adaptive production cell example

In this section we want to illustrate the presented approach on a vision of a future adaptive production cell. It shows the benefits of applying organic principles to traditional robot systems. Traditional engineering would handle and design such a production cell in a rather static way, consisting of individual machines that process workpieces with their tools and linked to each other in a strict sequential order using conveyors or similar mechanisms. The layout of the cell is therefore predefined, very inflexible, and rigid. Additionally, and maybe more important, such a system is extremely prone to system errors as the failure of one component will stop the whole system. However, the adaptive production cell is self-organizing which means that it is adaptive according to user-defined tasks (work plans) and compensates for component failures. Furthermore, it tries to optimize the throughput by finding a configuration which is best suited for the actual work plan.

1.4.1 System description

The adaptive production cell consists of KUKA Light-weight Robots (LWR), which are capable of using different tools. The traditional conveyor belt has been replaced by flexible and autonomous transportation units, which can carry workpieces. Some interesting concepts and ideas for flexible transportation units or conveyor belts are described by Bussmann in [29]. The goal of the example cell is to process workpieces in a user-defined sequence of tool applications (task).

Sect. 1.2 expounds that redundancy and software flexibility are needed to enhance traditional systems with self-organization. To achieve the maximum benefit from redundancy, it is important how the redundancy is distributed within the system. For example, it would be possible that a robot has the same tool three times and is the only one with this tool. Then, the robot is capable of reacting two times on tool breaks, but the breakdown of the whole robot stays a single-point of failure. Therefore a more failure tolerant distribution is to give one tool of each type to each robot, here a system breakdown needs at least a three-point of failure.

According to these results, the case study is arranged as follows (Fig. 1.6). We have three LWRs for processing, four carts for transportation and two storages, which provide unprocessed workpieces and store finished ones. Each LWR is able to perform all three capabilities: drill a hole into a workpiece, insert a screw into the drilled hole and tighten the inserted screw. The user-defined standard work plan

is to process all workpieces with all three tools. The given order is first to drill the hole, then insert the screw and at last to tighten it. In principle, an easy but not very high-performance solution is to let each robot perform all capabilities and change tools after each step. As switching tools is very time consuming compared to the time for applying the capability, the standard configuration is to let every robot perform a different task. The distribution of processing steps among different robots requires flexible routing of carts so that the correct order is maintained. One such configuration is sketched in Fig. 1.6.

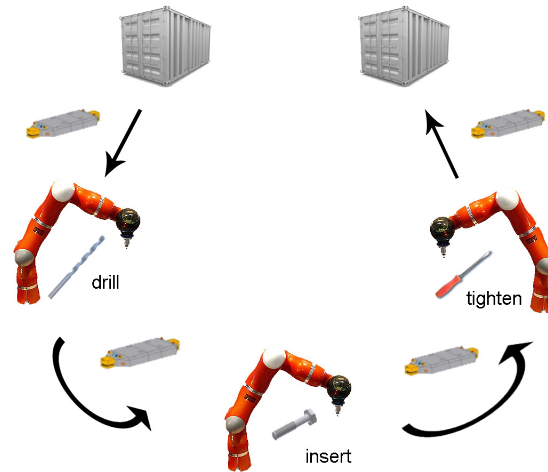


Fig. 1.6 Adaptive production cell

1.4.2 Design of self-organizing resource-flow systems

So far only the hardware of the adaptive production cell is described. But at least as interesting is the software for this example. For the two organic layers a software engineering guideline exists, which guides the engineer through several steps for developing self-organizing resource-flow systems [30]. The presented robotic cell is one simple instance of the class of resource-flow-systems. Other instances are all kinds of production automation, where you have a product running through a manufacturing process. One core concept of the guideline is the Organic Design Pattern (ODP, see Fig. 1.7), which determines the architecture and behavior of the system. It identifies the different components and artifacts of this domain and their relations.

The central components in the system are the *agents*, representing the robots and carts. They are processing the *resources* according to a given *task*. In case of the

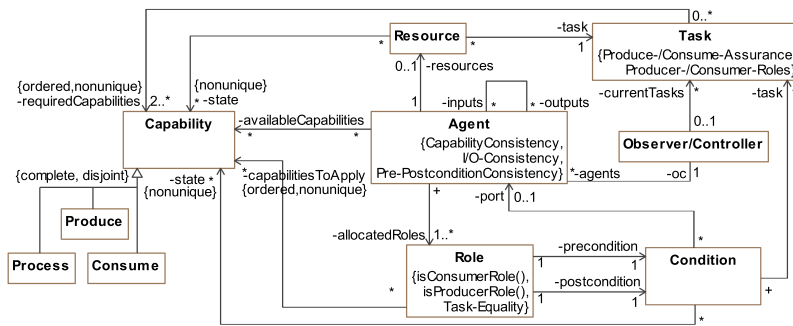


Fig. 1.7 Components of a Resource-Flow System

production cell every agent has several *capabilities*, divided into producing, processing, and consuming capabilities (*produce*, *process*, and *consume*). Consequently, the task is a sequence of capabilities beginning with a producing capability and ending with a consuming capability. Furthermore, the agent knows a couple of agents he can interact with and hand over resources (in case of the production cell, the workpieces). This is encapsulated in the *inputs* and *outputs* relation. The *role* concept is introduced to define correct resource-flows through the system. This means an agent has roles allocated telling him from which agent he receives the resource (*precondition/port*), which capabilities to apply, and then to which agent to hand over the resource (*postcondition/port*). Thus, the roles establish the connections between the agents and the combination of all roles forms the resource-flow. A system configuration is then a specific set of roles allocated to the agents (in this case robots and carts). For more details on the SE process and modeling of self-organizing resource-flow systems see [30]. In this case study, self-organization is done by role allocation. In case of a failure the system calculates a new valid set of roles, which is sufficient to fulfill the task again.

1.4.3 Specifying self-x through behavioral corridors

To receive correct behavior the allocation of roles to the agents is curbed as already mentioned in 1.3.4 by the specification of behavioral corridors. This is realized with OCL constraints, which are annotated to the ODP (Fig. 1.7). This means the configuration of the system which is planned by the Organic Planning Layer is restricted to configurations within the specified corridor. This is sufficient for a correct behavior, as the execution semantics of roles is predefined. In other words it is specified how roles are executed. Therefore the challenge is to restrict the roles which are assigned to the robots or carts in such a way that the resulting behavior leads to the desired system goal – in our example the correct production of the workpiece and the completion of the defined task.

One example for a consistency constraint for the robots or carts is the *Capability-Consistency*. In OCL this constraint is evaluated in the context of an agent as it is annotated to the *agent* concept, therefore *self* refers to a robot respectively cart.

```
(self.availableCapabilities -> includesAll(
   self.allocatedRoles.capabilitiesToApply))
```

The *Capability-Consistency* ensures that the robots and carts only accept and perform roles they are able to. In this case, only roles that need capabilities which are available.

Another interesting consistency constraint is the *I/O-Consistency* for a robot or cart (referred as `self`):

```
    (self.inputs -> includesAll(
       self.allocatedRoles.precondition.port))
and (self.outputs -> includesAll(
       self.allocatedRoles.postcondition.port))
```

The agents know a couple of neighboring agents. They are documented in their "input" and "output" relation. These indicate with whom robots and carts can exchange workpieces. The pre- and postconditions of the particular roles determine from which robot/cart the workpiece comes from and to which the workpiece should be given. The *I/O-Consistency* indicates that the robots/carts recognize the breakdown of their required partners in the actual resource-flow. The required partners are the ports in the pre- and postconditions of the roles and must be part of the input respectively output relation of the agents. During runtime the robots/carts ping these neighbors to ensure that they are still available for receiving workpieces.

The carts are constrained in the way that they only take transport assignments between robots that are reachable for them. All these constraints can be monitored during runtime by the agents themselves as they can be evaluated locally. In general, also quantitative constraints for a configuration can be of interest, such as the assigned capabilities will not exceed a defined load or that the throughput has a certain threshold. These constraints usually are not monitored as they are violated if a failure occurs, which implies a previous violation of another monitored constraint.

More details about specification of behavioral corridors by constraints can be found in [8]

1.4.4 System behavior at runtime

The system starts with an initially calculated role allocation as depicted in Fig. 1.6. The needed tool applications are spread to the robots and the carts are assigned different routes to move the workpieces around. If a failure occurs, e.g. the drill tool of the drilling robot breaks, the robot monitors this violation of its *Capability-*

Consistency constraint and starts a reconfiguration. It collects information about the neighboring robots and carts, calculates a new distribution of tool assignments and re-routes the carts in a way that production can continue. A traditional system would stop and a human interaction would be needed here. The reconfigured situation is depicted in Fig. 1.8.

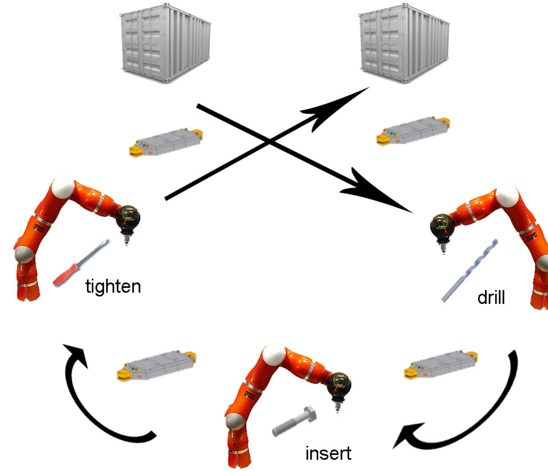


Fig. 1.8 Adaptive production cell after reconfiguration

In this case study, the robots and carts have only local rules and interaction possibilities. The resulting system is a self-organizing production cell which is capable of reacting to changes in the environment and new work plans. The configurations which are calculated by the robots or carts in case of a local constraint violation (e.g. capability or input, output loss) fulfills the constraints specified for the system. This means that the occurring emergence is restricted to positive emergence as claimed in Sect. 1.2.1.

1.4.5 Realizing self-reconfiguration

There are several possibilities to implement self-reconfiguration. Currently, reconfiguration is done using a constraint solver, here Kodkod [35], to receive valid configuration for each robot and cart. Therefore the actual system state and the OCL constraints are converted into a formal model representing a constraint satisfaction problem (CSP). The model can directly be derived from the design pattern and the annotated constraints (see Sect. 1.4.2). This solver then tries to find a solution fulfilling all constraints, which is then spread to the agents. More details about the

transformation and the use of constraint solvers for this class of systems can be found in [21, 22].

The advantage of integrating common techniques in contrast to stochastic algorithms is, that it is easier to give behavioral guarantees and ensure correct reconfiguration.

But also heuristic and stochastic algorithms, like genetic algorithms [7], can be used to realize the self-reconfiguration. For large problems they are often faster and allow to integrate self-optimization by defining adequate fitness functions, which also takes the quality (e.g. load balancing or throughput) of a solution into account.

For the production cell example a distributed coordination mechanism was developed, which realizes reconfiguration by applying a wave-like self-organization strategy [33]. The agent which recognizes a failure starts a self-reconfiguration by asking its neighbors if they can help solving the problem. If not, the search is propagated forward to the next but one neighbors and so forth. If a solution is found the agents switch to their new configuration and continue processing. In the best case two agents just switch their roles and only the adjacent carts are re-routed. Here reconfiguration is only needed for local subset of the system, which is advantageous in larger scale systems.

1.4.6 Proof of concept

The organic layers are implemented with a multi-agent framework called Jadex [27], which also provides the communication infrastructure. On each robot and cart one Organic Control Layer agent is running and coordinating them via the interfaces provided by the Robot Programming Layer. Whenever a failure occurs or a reconfiguration request of another agent is received it spawns an Organic Planning Layer agent which then is handling the reconfiguration for this agent. There are different implementations (see 1.4.5) which are integrated into these planning layer agent and can be used for reconfiguration. The reconfiguration is based only on local knowledge and after successful reconfiguration the Organic Planning Layer terminates itself. Thus, there is no global knowledge base generated during runtime.

For the production cell scenario we implemented a prototypical implementation using Microsoft Robotic Studio. It provides a physical simulation environment for robotic applications and allows for prototypical testing of the developed concepts. A first version is described in [14].

1.5 Conclusion

In the field of Organic Computing, we were looking at the domain of production automation, in particular the field of adaptive production cells. Further in robotics research, we looked at facilitating the software development for industrial robots

and improving software quality. In this chapter we presented how both worlds can fit together and how Organic Computing principles can be used to realize a flexible automation system. To be more concise, how the architecture of a self-organizing robotic cell can look like and how it can be implemented.

The lower layers were prototypically substituted by a simulation and coupled to the implementation of the organic layers within a multi-agent system, as described in Sect. 1.4.6. Nevertheless, these systems can benefit from the application of Organic Computing principles, especially in terms of failure tolerance and flexibility. One major advantage of the proposed architecture and its implementation is that it is formally grounded and, therefore, allows to give behavioral guarantees with respect to the assigned configurations, which always leads to correct processing of the resources. The definition of a behavioral corridor and the assurance of remaining inside this corridor allows flexibility and gives firm guarantees about the system, which is very important for the acceptance in industrial applications. However, the drawback of moving self-organization into high-level layers is that no real-time critical behavior can be considered. Hence, only non real-time critical reconfigurations are possible.

Different production scenarios and factory settings need diverse reconfiguration mechanisms, e.g. completely decentralized coalition formation or wave propagations. We are currently working on different plug-ins for the Organic Planning Layer to enhance it by several reconfiguration algorithm implementations. In order to meet the flexibility requirements as proposed in Sect. 1.2, robotic software architecture (see [12]) which corresponds to both the Robotic Programming Layer and the Robotic Control Layer is currently extended.

Acknowledgements This work has been partly sponsored by the priority program *Organic Computing* (SPP OC 1183) of the German research foundation (DFG).

References

1. Angerer, A., Hoffmann, A., Schierl, A., Vistein, M., Reif, W.: The Robotics API: An object-oriented framework for modeling industrial robotics applications. In: Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010), Taipeh, Taiwan. IEEE Computer Society Press (2010)
2. Black, J.T., Musunur, L.P.: Robotic manufacturing cells. In: S. Nof (ed.) Handbook of Industrial Robotics, chap. 35, pp. 697–716. John Wiley & Sons, Hoboken, NJ, USA (1999)
3. Branke, J., Mnif, M., Müller-Schloer, C., Prothmann, H., Richter, U., Rochner, F., Schmeck, H.: Organic Computing – Addressing complexity by controlled self-organization. In: Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), Paphos, Cyprus, pp. 185–191. IEEE Computer Society Press (2006)
4. De Schutter, J., De Laet, T., Rutgeerts, J., Decré, W., Smits, R., Aertbeliën, E., Claes, K., Bruyninckx, H.: Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. Int. J. Rob. Res. **26**(5), 433–455 (2007). DOI http://dx.doi.org/10.1177/027836490707809107

5. Finkemeyer, B., Kröger, T., Wahl, F.M.: Executing assembly tasks specified by manipulation primitive nets. Advanced Robotics **19**(5), 591–611 (2005)
6. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. IBM Systems Journal **42**(1), 5–18 (2003)
7. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning, 1 edn. Addison-Wesley Professional (1989)
8. Güdemann, M., Nafz, F., Ortmeier, F., Seebach, H., Reif, W.: A specification and construction paradigm for organic computing systems. In: Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008), Venice, Italy, pp. 233–242. IEEE Computer Society Press (2008)
9. Güdemann, M., Ortmeier, F., Reif, W.: Safety and dependability analysis of self-adaptive systems. In: Proceedings of ISoLA 2006. IEEE CS Press (2006)
10. Hägele, M., Nilsson, K., Pires, J.N.: Industrial robotics. In: B. Siciliano, O. Khatib (eds.) Springer Handbook of Robotics, chap. 42, pp. 963–986. Springer-Verlag, Berlin, Heidelberg, Germany (2008)
11. Hägele, M., Skordas, T., Sagert, S., Bischoff, R., Brogårdh, T., Dresselhaus, M.: Industrial robot automation. White paper, European Robotics Network (2005)
12. Hoffmann, A., Angerer, A., Ortmeier, F., Vistein, M., Reif, W.: Hiding real-time: A new approach for the software development of industrial robots. In: Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2009), St. Louis, MO, USA, pp. 2108–2113. IEEE Computer Society Press (2009)
13. Hoffmann, A., Angerer, A., Schierl, A., Vistein, M., Reif, W.: Towards object-oriented software development for industrial robots. In: Proceedings of the 7th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2010), Funchal, Portugal. INSTICC Press (2010)
14. Hoffmann, A., Nafz, F., Ortmeier, F., Schierl, A., Reif, W.: Prototyping plant control software with microsoft robotics studio. In: Proceedings of the Third International Workshop on "Software Development and Integration in Robotics" (SDIR-III). IEEE International Conference on Robotics and Automation (2008)
15. Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
16. Mason, M.: Compliance and force control for computer-controlled manipulators. IEEE Transactions on Systems, Man, and Cybernetics **11**(6), 418–432 (1981)
17. Mehrabi, M., Ulsoy, A., Koren, Y., Heytler, P.: Trends and perspectives in flexible and reconfigurable manufacturing systems. Journal of Intelligent Manufacturing **13**(2), 135–146 (2002)
18. Müller-Schloer, C.: Organic computing: on the feasibility of controlled emergence. In: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Stockholm, Sweden, pp. 2–5. ACM (2004)
19. Müller-Schloer, C., Sick, B.: Controlled emergence and self-organization. In: Würtz [39], pp. 81–104
20. Müller-Schloer, C., von der Malsburg, C., Würtz, R.P.: Organic computing. Informatik Spektrum **27**(4), 332–336 (2004)
21. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.P., Reif, W.: A generic software framework for role-based organic computing systems. In: Proc. Intl. Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 96–105 (2009)
22. Nafz, F., Ortmeier, F., Seebach, H., Steghöfer, J.P., Reif, W.: A universal self-organization mechanism for role-based organic computing systems. In: ATC '09: Proceedings of the 6th International Conference on Autonomic and Trusted Computing, pp. 17–31. Springer-Verlag, Berlin, Heidelberg (2009). DOI http://dx.doi.org/10.1007/978-3-642-02704-8-3
23. Nafz, F., Seebach, H., Steghöfer, J.P., Bäumler, S., Reif, W.: A Formal Framework for Compositional Verification of Organic Computing Systems. In: Proceedings of the seventh International Conference on Autonomic and Trusted Computing (ATC-10) (2010)
24. Okamura, A., Smaby, N., Cutkosky, M.: An overview of dexterous manipulation. In: Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA 2000), San Francisco, CA, USA, pp. 255–262. IEEE Computer Society Press (2000)

25. OMG: Object Constraint Language, OMG Available Specification (2006)
26. Pires, J.N.: New challenges for industrial robotic cell programming. Industrial Robot **36**(1) (2009)
27. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A bdi reasoning engine. In: M.D.J.D. R. Bordini, A.E.F. Seghrouchni (eds.) Multi-Agent Programming, pp. 149–174. Springer Science+Business Media Inc., USA (2005). Book chapter
28. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for organic computing. In: GI Jahrestagung (1), pp. 112–119 (2006)
29. Schild, K., Bussmann, S.: Self-organization in manufacturing operations. Commun. ACM **50**(12), 74–79 (2007). DOI http://doi.acm.org/10.1145/1323688.1323698
30. Seebach, H., Nafz, F., Steghöfer, J.P., Reif, W.: A software engineering guideline for self-organizing resource-flow systems. In: Proceedings of the Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2010), Budapest, Hungary. IEEE Computer Society Press (2010)
31. Seebach, H., Ortmeier, F., Reif, W.: Design and construction of organic computing systems. In: Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2007), Singapore, pp. 4215–4221. IEEE Computer Society Press (2007)
32. Smits, R., De Laet, T., Claes, K., Bruyninckx, H., De Schutter, J.: iTASC: A tool for multi-sensor integration in robot manipulation. In: Proceedings of the IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI 2008), Seoul, Korea, pp. 426–433. IEEE Computer Society Press (2008)
33. Sudeikat, J., Steghöfer, J.P., Seebach, H., Renz, W., Preisler, T., Salchow, P., Reif, W.: Design and simulation of a wave-like self-organization strategy for resource-flow systems. In: 4th International Workshop on Multi-Agent Systems and Simulation (MAS&S) (2010). Accepted
34. Thomas, U., Finkemeyer, B., Kröger, T., Wahl, F.M.: Error-tolerant execution of complex robot tasks based on skill primitives. In: Proceedings of the 2003 IEEE International Conference on Robotics and Automation (ICRA 2003), Taipei, Taiwan, pp. 3069–3075. IEEE Computer Society Press (2003)
35. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: O. Grumberg, M. Huth (eds.) Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 4424, chap. 49, pp. 632–647. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). DOI 10.1007/978-3-540-71209-1_49. URL http://dx.doi.org/10.1007/978-3-540-71209-1_49
36. Tsang, E.: Foundations of constraint satisfaction (1993)
37. Vesely, W.E., Goldberg, F.F., Roberts, N.H., Haasl, D.F.: Fault Tree Handbook. U.S. Nuclear Regulatory Commission, Washington, DC (1981)
38. Vistein, M., Angerer, A., Hoffmann, A., Schierl, A., Reif, W.: Interfacing industrial robots using realtime primitives. In: Proceedings of the 2010 International Conference on Automation and Logistics (ICAL 2010), Hong Kong, China. IEEE Computer Society Press (2010)
39. Würtz, R.P. (ed.): Organic Computing (Understanding Complex Systems). Springer-Verlag, Berlin, Heidelberg, Germany (2008)
40. Zaeh, M., Ostgathe, M.: A multi-agent-supported, product-based production control. In: Proceedings of the 7th IEEE International Conference on Control and Automation, Christchurch, New Zealand, pp. 2376–2383. IEEE Computer Society Press (2009)