Intrinsic Flexibility and Robustness in Adaptive Systems: A Conceptual Framework

Stephan Kubisch, Ronald Hecht, Ralf Salomon, Dirk Timmermann
University of Rostock, Institute of Applied Microelectronics and Computer Engineering
18051 Rostock, Germany
e-mail: {stephan.kubisch, ronald.hecht, ralf.salomon, dirk.timmermann}@uni-rostock.de
Phone/Fax: ++49 (0)381 498 - 7271/7252

Abstract— **Dealing with uncertainty is an important issue in a broad range of research fields and industrial applications. For example with outer space missions or unmanned vehicles in perilous environs, today's and future systems have to adapt to changing environmental conditions. But high specialization of today's systems also results in decreased robustness and flexibility. These systems show high performance. But in dynamic environments, they lack flexibility as well as robustness. In case of failure or changing conditions, also referred to as uncertainty or uncertain events, flexibility and robustness are crucial. Based on the proverb "Prevention is better than cure!", mechanisms have to be implemented in technical systems to diminish the contrary behavior of robustness and flexibility compared to specialization. A conceptual framework based on a library related to Concurrent Version Systems is proposed. Every change and modification on the system configuration is minuted over time. Thereby, the ability to fall back on a stable system state in critical situations, traps or impasses assures robustness and survivability.**

Keywords—Robustness, Flexibility, Adaptive Systems

I. Introduction

Dealing with uncertainty is an important issue in a broad range of research fields and industrial applications. In economics [1], in industry [2], and in future sophisticated technical applications [3], [4], uncertain events, dynamic environs, and unstable parameters reveal the urgent need for (autonomous) adaptability to unforeseen changes. Thereby, adaptation means specialization to a certain temporarily stable setting or niche—in nature, to optimally exploit that niche and to achieve high fitness and in technical systems, to show autonomous behavior, best possible performance, or to reach mission objectives in any way. Unfortunately, flexibility and robustness decrease the more a system is specialized as sketched in Figure 1. Both suffer from high specialization as pointed out in [5].

But in case of failure or changing conditions, flexibility and robustness tip the scales. In [2], robustness and flexibility are classified as two sides of the same coin. Both are principles to address uncertainty but they are of different nature. On the one hand, robustness reduces uncertainty. No actions are necessary when environmental changes occur because the system can withstand and tolerate the changes. Ku refers to it as resistance or immunity to change. On the other hand, flexibility does not reduce uncertainty but is the ability to react to change

1-4244-0166-6/06/$20.00 @2006 IEEE

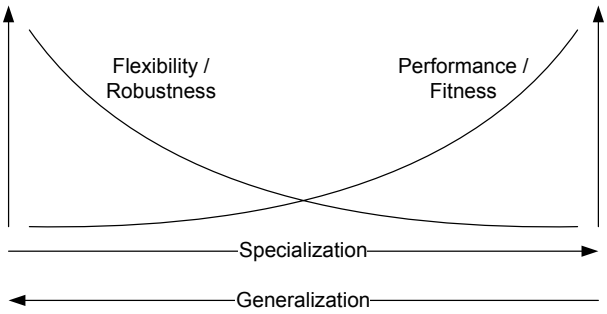

Fig. 1. Specialization versus Generalization

and therefore to achieve the best possible performance in a dynamic environment.

In nature and biological systems, specialization through adaptation is also connected to irreversibility due to the loss of the nascent and unspecialized morphology [5], although there is (just) one known and proved exception [6]. But unlike to natural systems, mechanism supporting reversibility can be implemented in technical systems as focused in this paper. Thus, the system is capable of specialization but also of generalization. The goals of the specialization process are improving the performance and at least not to downgrade over time. But at any rate, generalization secures basic functionality and protects from system failure. Both mechanisms are mandatory to reach a mission's objective as far as possible in uncertain dynamic environments. In [7], reducing the functional spectrum is also referred to as graceful degradation.

In [4], the need of combining techniques for soft and hard computing is pointed out. The term *hard computing* refers to traditional computing techniques and methods. The term *soft computing* refers not to software but to computing techniques that deal with uncertainty and adaptation. Further, the processes of adaptation and learning are the very purposes of neural networks and genetic algorithms of various sorts. In essence, a neural network determines its behavior in dependence of a number of parameters called connections [8]. In order to work properly, a neural network adjusts those weights with respect to a (given) set of trainings patterns. Depending on the network type (model) and the chosen learning algorithm, a neural network may exhibit some online adaptation capabilities. These adaptation capabilities require some sort of redundancy: every parameter requires some memory in order to be

modifiable, and also may store different values or system parameters in order to evaluate the benefits of changes.

Life long adaptation has been successfully demonstrated in the field of autonomous mobile robots. Examples are the subsumption architecture [9], the distributed adaptive control (DAC) architecture [10], and the self-organization-through-proprioception (STP) architecture [11]. Common to all three architectures is that they are not operating in isolation but in cooperation with the robot and its environment. Due to the robot's morphology and the current environmental properties, any robot action induces some feedback that the robot's control architecture utilizes in order to apply suitable modifications to the network. By incorporating all a certain amount of network plasticity, the robot's morphology, and the environment, the robot can constantly adapt to change that it can sense, either directly or indirectly through the results of the intrinsic robot-environment interaction.

The paper proposes a conceptual framework for an adaptive and reconfigurable hardware/software (HW/SW) system. The basic system is a Field Programmable Gate Array- (FPGA) and Network-on-Chip- (NoC) based architecture, which is capable of partial dynamic reconfiguration. Adaptation steps and modifications are stimulated by a feedback loop containing quantification functionality and learning capability. The central point within this proposal is a system library comparable to Concurrent Version Systems (CVS). Especially on occurrence of abrupt and critical changes, this library provides the ability of generalization—referred to as fall back capability.

The remainder of the paper is organized as follows: Section II briefly introduces CVS and highlights the feasibility as a system library and knowledge base. Section III shortly details the nature of reconfigurable hardware devices and our previous work as base for the proposed framework. This framework is characterized in Section IV. Section V concludes the paper.

II. CVS Approach

A. From SCCS to Subversion

The introduction of the first SCCS (Source Code Control System) was influenced by two facts. On the one hand, with the first mass storage devices during the 1970s memory was incredibly expensive. On the other hand, software development was getting more complex and sophisticated because the functional spectrum increased, new high performance processing devices have been developed, and design teams grew from just one person to more than a triple-digit number. For companies working on large software projects, it was necessary to minimize the amount of memory needed to store the sources. AT&T developed a first SCCS, which was already capable of version numbering and branching [12]. But the main benefit was the diffing function. Only the differences between two versions of the same file were stored to save memory. In [13], the RCS (Revision Control System), which allowed more than just one user to work with a copy of certain file, was introduced. But only one user at the same time was able to write a new version into the main repository requiring exact coordination within the team. This drawback of locking was eliminated with CVS [14]. Here, more than just one user could simultaneously upload changes to the main repository due to the merging functionality. Today, RCS and CVS are standard mechanisms in, e.g., Linux operating systems. They can be used with the *ci*, *co* and *cvs* commands. Recently, another system called Subversion was released [15]. Subversion is similar to CVS but provides advanced functionality and features. Most CVS commands are compatible with Subversion and CVS is still widely used and acknowledged due to its successful and longtime employment in the field.

B. A CVS System Library & Knowledge Base

The vital aspects of CVS in the light of our framework are not its platform independence or its network-compatible client/server structure but the incremental logging of changes and modifications over time, version numbering, branching, and the ability to return to a previous minuted version. Additionally, memory, which is limited in most embedded systems, can be saved because only the differences between two consecutive versions are to be stored except with binary files. This is also referred to as Delta-Coding. In our context, the information stored in the CVS library will not (only) be source code, what is the original purpose of RCS and CVS. The information to be stored will contain plaint text files, binary files, and system configuration parameters as representation of a certain system state at a given point of time. So to say—the system's context will be minuted. The essence of the systems context is further highlighted in Section IV.

Figure 2 sketches the development of an example system over time. We refer to it as the system's life cycle. Every system configuration that results in good performance and every change on the system is marked with a point on

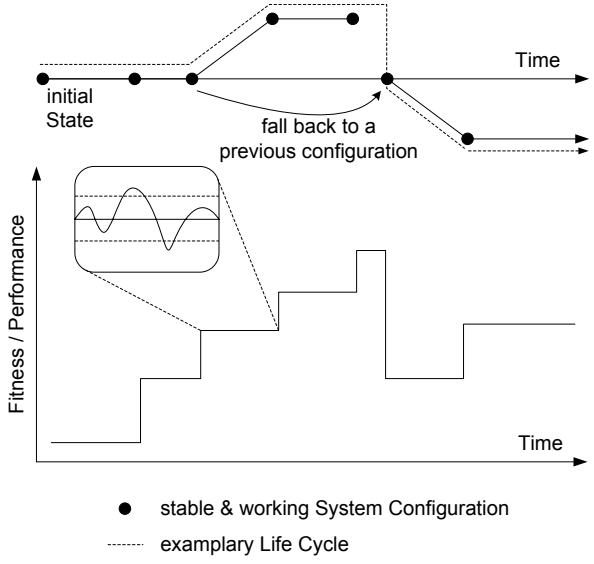

Fig. 2. Example System Development over Time

the timeline and is minuted in the CVS library with a unique version number. Thereby, different branch levels may indicate more or less important changes. Points on the same branch level may indicate just small parametric changes. Adaptation to the environment is indicated within the figure by the course of the system's fitness and performance as a general measurement for the outcome of a change on the system's configuration. We refer to specialization when the mean fitness increases. However, as outlined during the introduction, the system also needs the ability to generalize to secure at least basic functionality, to provide a clean starting point for further adaptation steps, or for automatic recovery. This is needed in critical situations and occurrences of abrupt changes. By using the fall back ability of the CVS library, meaning a check-out of a retrograde revision, system robustness can again be guaranteed. We refer to generalization when fitness decreases due to falling back to a formerly minuted system configuration.

The library contains at least significant chronologically ordered milestones up to the complete system life cycle and can be considered as the system's knowledge base. Further, this knowledge base may also be used for analytical estimations and anticipation of upcoming events or changes within the environment. Periodically occurring changes or proactive modifications may be prepared and stimulated in advance with such a-priori information. In [11], this is referred to as proprioception. So, the systems behavior and further development are not only determined by, e.g., the current configuration and real-time sensor information but also by its previous development. Within the growing knowledge base, already encountered problems and learned behavior patterns are stored, which increase that systems generality [16]. This way, the system's history is part of its structure and future development.

At this point, the multi-user feature of the CVS is practical. In our case, there are not multiple user in the sense of large design teams. But multiple internal entities need access to the system library at the same time to independently update or analyze information, e.g., two concurrent instances of the system model as further described in Section IV.

III. Previous Work and Experiences

A. FPGA Design Flow and Reconfiguration

In general, FPGAs show a matrix-like structure of reconfigurable blocks containing basic logic elements. Useful embedded hardmacros like microprocessors, special arithmetic units and fast I/O interfaces are also integrated into latest platform FPGAs. This fine grained architecture is flexible and versatile. To change the functionality, the device can be reconfigured. Older devices just support complete reconfiguration while the latest FPGAs, for example the Virtex-family of platform FPGAs, also facilitate row- and tile-based partial dynamic reconfiguration.

With the help of a synthesis flow, a formal description of the design, e.g., written in VHDL (Very high speed integrated Circuit Hardware Description Language) is translated into an optimized device specific netlist. The netlist is merged with user constraints and mapped onto the available reconfigurable resources of the target device. After placing and routing the design to conform with timing constraints, a bitstream file is generated that can be downloaded to the device and assigns the functions and connections within the logic blocks.

B. A deeper Look into Partial Dynamic Reconfiguration

Partial dynamic reconfiguration holds the potential to build autonomous and highly flexible Systems-on-Chip. Individual design parts or functional modules can be replaced on demand without interfering the operation of the remaining design units. On the one hand, this is beneficial for applications, which usually do not allow any interrupts. On the other hand, erroneous parts can be fixed, design parts can be updated or modified, and distinct functional units can be swapped or relocated. The latter one is already described in [17] and [18] as virtual circuitry. Similar to virtual memory, virtual hardware can be shared. Therefore, an appropriate management entity is necessary. Today, this is usually an operating system (OS), which runs on an embedded microprocessor within the FPGA or on an external device, e.g., to manage the reconfigurable area.

Partial dynamic reconfiguration and the relocation of partial design blocks are feasible but still too tricky. OS and tool support is still under development and detailled technological information is mostly not published by the manufactures or protected by NDAs. The full potential of partially and dynamically reconfigurable systems can not yet be tapped. Thus, we developed a framework, which relies on well-know and approved mechanisms and is independent from a certain technology. It is briefly presented in the next section.

C. A reconfigurable HW/SW Framework

In [19], we already presented our system architecture for dynamically reconfigurable runtime systems. It is based on a NoC, which is used for communication and structures the reconfigurable area of the FPGA in distinct parts. On an embedded microprocessor, a general-purpose OS, which is modified for partial dynamic reconfiguration, is running [20]. Therefore, a Debian Linux was chosen. Its basic architecture is briefly sketched in Figure 3. The hardware abstraction layer is extended with additional device drivers to access the NoC and the FPGA. The kernel layer contains a number of instances to manage the IP cores, e.g., registration, execution, interruption, and swapping. Within the user space, a library of IP cores exists and applications can be executed.

As previously mentioned, partial dynamic reconfiguration is beneficial in a lot of ways but still has its drawbacks. To circumvent these drawbacks and to be independent from technology and manufactures, we employ a model of the FPGA and the NoC [21] to simulate the behavior of the system within the Linux system itself. The model is written in SystemC. It is located in the OS' user space and directly connects to the FPGA and NoC drivers as pictured in Figure 3. Multiple instances of the model

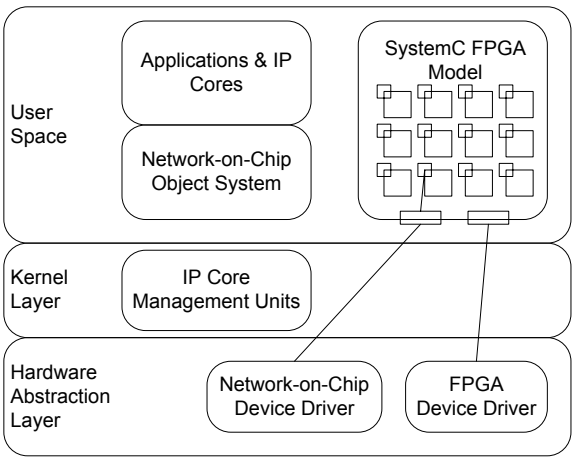

Fig. 3. Operating System Structure

can be referenced and used in parallel. Thus, we can do simulation and testing of the behavior of a HW/SW co-design on a common Linux workstation without using a real FPGA, for example.

Each functional unit within the HW/SW framework, also called IP core, can have a number of different implementations. As long as the interface of an IP core remains the same across its different implementations, communication with and between IP cores is fully transparent. One class of IP core representations are called accelerators and are written in a hardware description language, e.g., VHDL. Accelerators are designed for reconfigurable hardware, which is also the original reason of the framework. A second class of representations is called decelerators. A decelerator represents a soft-implementation of an IP core's functionality, e.g., using C++. Accelerators can be loaded into the FPGA to benefit from hardware performance and true parallel execution. But if desired, the same functionality can also be executed in software within the OS' user space using the decelerated version if no speedup is required at the moment or hardware resources are currently used otherwise. Thus, hardware can be scheduled like processes and threads; functionality can be swapped.

IV. The Adaptive System

A. The System Context

To minute and store the system's context at a given point of time sounds easy but leaves open what the structure of the context is like. A detailed listing of the elements required to unambiguously specify the system context at a certain point of time can only be made for a concrete application. But a general structure is proposed.

We suggest a set of three types of information. At first, a time stamp is used to mark a specific context on the timeline. Secondly, the internal system state and configuration is to be minuted, e.g., the loaded IP cores, the representations of the IP cores that are loaded, and the current values of the system variables and the parameters of the learning mechanisms. Thirdly, it is indispensable to store the characteristics of the input data or the values of input parameters along with the internal system configuration, which is required to recognize a certain environmental state and also to recognize a change in the externalities the system can sense. Only with suchlike information, it is possible to choose and switch to a former system configuration. For adaptive systems in dynamic environs, the internal system configuration correlates with the input data. The system is intended to be adaptive but is *not* intended to be random. To summarize, the system context consists of the internal configuration and environmental conditions. This is illustrated in Figure 4.

Another type of information is required to realize adaptation. The system must know its own constraints and functional scope. A high level specification must exist within the system to define its literal tasks and goals. These parameters are fix and the same for every minuted context. All these specified properties are compared against the quality of the current system outputs within the feedback loop. Thereby, the system adapts into an intended direction instead of developing undirected and unregulated or even not at all. Speaking in terms of autonomic computing—it shows some degree of self-awareness.

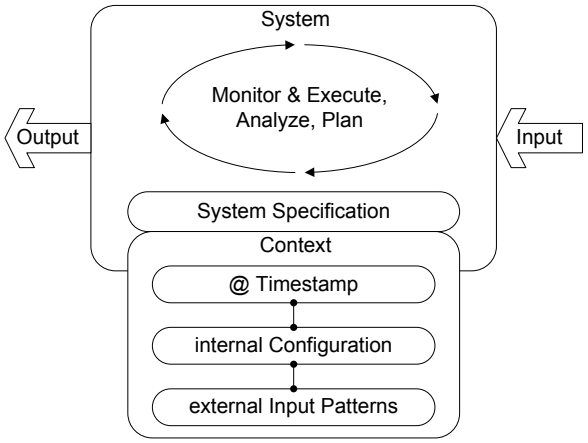

Fig. 4. The System Context

B. Redundancy and Shift Work

Quick online adaptation in nearly realtime is tricky but necessary in dynamic environments. A system is stressed with both performing its tasks and adapting to the environmental dynamics at the same time. This leads to interruptions and other drawbacks like chattering [16] and trashing when changes on the system configuration are scheduled during operation. Using redundancy and scheduled shift work can find a remedy here. Therefore, two redundant instances of the system model as described in Section III are used. One instance is the primary model and fulfills the system's literal tasks while the second instance is active in the background. The multi-user capability of the CVS library allows for concurrent operation and access to the knowledge base. Both instances exist concurrently and operate in shifts meaning rotation to alternate shifts in the loop shown in Figure 5a. We define two separate shift types similar to the RAINBOW system

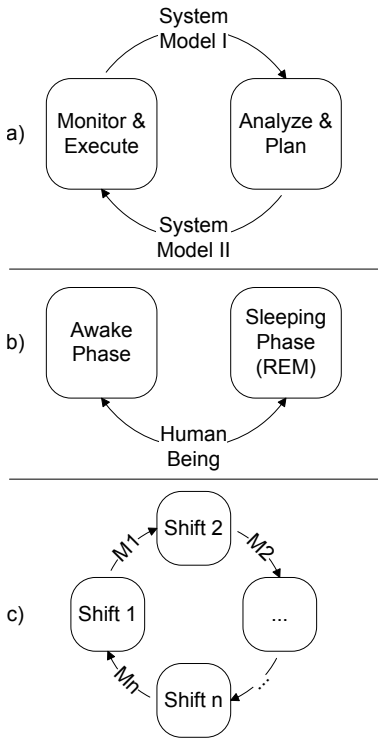

Fig. 5. Redundancy and Shift Work

in [22] or the partitioning of the MAPE-loop introduced by IBM [23].

The first shift type in the loop is the primary operating state and of active nature. The system model currently scheduled for that shift performs the application-oriented tasks and monitors incoming data in parallel. Input data is merely aggregated, maybe preprocessed, and minuted within the system library. Monitoring can be done in a streaming way or be based on uniform samples. Concurrently, the system fulfills its literal tasks. This can be compared to the awake phase of human beings, which is outlined in Figure 5b. During this phase, we act and live based on our conditioning and learned patterns of behavior. Additionally, we collect impulses, stimuli and input from our environment. However, only a fraction of these information can immediately be managed because we have to cope and interact with the environment. A fitting example are training units for athletes. One could argue that a two-hour unit is less effective than four or six hours of practice. But one can also notice that the athlete does not perform much better after six hours of training than after two hours. Thus, we define a second shift type, which is a passive operating state regarding the system's literal tasks. During this shift, the corresponding system model analyzes, parses, and purifies the minuted input information and prepares changes on the system configuration within the intended functional scope if they appear adequate. Therefore, a weighting function is needed, which compares specified and current characteristics. Regarding the human being example, this shift is similar to the sleeping phase of the athlete. The trained patterns are arranged and internalized within the brain structure and during the next awake phases, the athlete will perform better. In biology, this phase is also referred to Rapid-Eye-Movement (REM) phase. It is characterized with paralyzed body movement (no execution) but high cerebral activity (analyze) and fluttering eye movements.

Human beings or other smart living things show no redundancy regarding a single individual and can be in only one phase at a time. But unlike to this example from biology, two ore more instances of the system model can be used within the HW/SW framework. The instance currently switched to the Execute & Monitor shift is the primary model. The other one(s) operate(s) in the background in parallel. Depending on the degree of redundancy that can be spent, which depends on the number of redundant system models, more than two shift types can be defined to keep the current tasks of each redundant model small, although the overall system complexity may increase. This way, the system becomes more fine-grained as drawn in Figure 5c. Here, the rotation from shift *n-1* to shift *n* is done with the models *M1* to *Mn*. Additionally, two or more redundant system models allow for parallel verification and pilot runs before applying potential changes. Thereby, performance deteriorating chattering is reduced and the system can smoothly switch over to a new configuration.

C. Intrinsic Flexibility and Robustness?

Robustness was referred to as the ability to tolerate changes. The redundant, concurrent system models allow for a quick context switching on occurrences of abrupt changes and also for concurrent simulation and testing to continuously adapt to the environment. By falling back to a previous stable but more general system configuration, periods of high dynamism can be endured and outlasted. Switching back to a stable setting means stabilizing the system, perhaps reducing fitness, but guaranteeing survivability. Due to the fall back capability, there is, theoretically, no point-of-no-return any more. Force majeure can no be calculated and remains uncertain, of course.

Flexibility was referred to as the ability to react on changes. Adaptation steps occur each time the redundant system models rotate to the next shift within the loop. Steps are not scheduled after fixed time intervals because a rotation also implicates certain costs. Costs arise, for instance, through context switching and latency and must be calculated when designing a concrete application. Thus, only in case of need or when improvements in fitness or performance are likely, the system is readjusted. Therefore, a weighting function is used to estimate the advantages compared to the current active system configuration.

The weighting function is a system-wide observer. It is directly connected to the learning algorithm. The learning algorithm is only needed during the Analyze & Plan shift. To this end, the learning algorithm stores the context information as well as both the system's reaction and the reaction's usefulness. In case of non-optimal outcomes, the system applies adaptation steps, such as error back-propagation or reinforcement learning [8]. In order for the learning algorithm to work properly, it is essential that

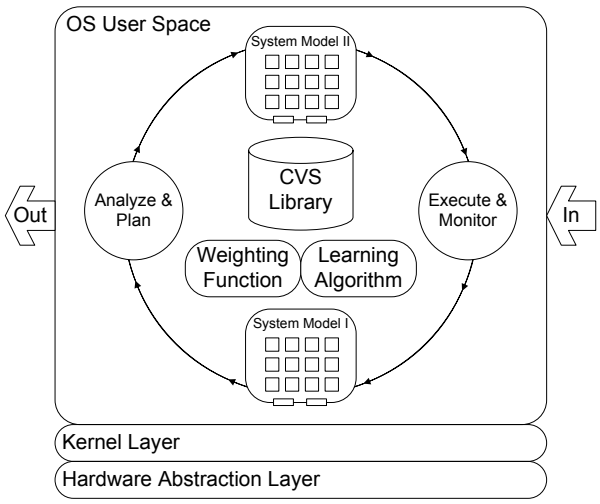

Fig. 6. Conceptual adaptive Framework with concurrent System Models

the system stores previously seen context data. This way, the system can constantly monitor its own behavior like watching into a mirror, and can adapt to new situations if it notices significant changes. In the latter cases, it has to replace those old data entries from the stored set that became invalid due to the changes. It should be noted that in most application, the system does not need to store all available data but just a compressed representative subset. As outlined in the introduction, online adaptation with neural networks necessitates a certain degree of redundancy and knowledge on previous system parameters. Both requirements are given with the proposed framework. Its coarse structure is displayed in Figure 6. Redundancy is provided by using at least two concurrent system models. Information memory is provided by the system library.

V. Conclusion

The paper submitted the concept of a framework for adaptive systems with focus on flexibility and robustness in dynamic environs. A suchlike system is able to adapt to environmental changes and to specialize over time. But particularly on occurrences of abrupt changes or impasses, the system also needs to generalize to protect itself from failure and secure basic functionality. Thus, a mechanism was introduced to improve robustness and flexibility. A central system library with features related to CVS provides a so-called fall back ability to return to a previously minuted system state. Based on an existing HW/SW architecture for dynamically reconfigurable systems, the combination of a feedback loop, an adaptive algorithm, a central system library, and redundant system models was characterized. The feasibility of a CVS library as knowledge base was detailed. The framework is applicable for pure software systems but also for joint HW/SW systems and reconfigurable hardware systems if, e.g., technological and theoretical prerequisites for partial dynamic reconfiguration are given.

The future work will comprise the implementation and integration of the learning algorithm and the feedback loop within the base system. The definition of the system's context for a defined test application is necessary. Among other things, open questions target the degree of redundancy that is beneficial for the overall system architecture, the granularity of the system specification, and the type of training patterns for a neural network respectively the initial state of the whole system.

References

[1] J. Husdal, "Robustness and flexibility as options to reduce uncertainty and risk," Molde, Norway, 2004.

[2] A. Ku, "Modelling uncertainty in electricity capacity planning," Ph.D. dissertation, London Business School, London, UK, 1995, available at http://www.analyticalq.com/thesis.

[3] C. Rouff et al., "Autonomicity of NASA Missions," in *Proc. of the 2nd IEEE Int. Conf. on Autonomic Computing (ICAC'05)*, Seattle, Washington, USA, June 13-16 2005.

[4] B. Sick and S. J. Ovasaka, "Fusion of Soft & Hard Computing Techniques: A Multi-Dimensional Categorization Scheme," in *Proc. of the 2005 IEEE Mid-Summer Workshop on Soft Computing (SMCia05)*, Espoo, Finland, June 28-30 2005.

[5] C. Lundberg, "Segmentation and vertebrate origins," available at http://home.inreach.com/cliff_lundberg/tablecon.html.

[6] M. F. Whiting et al., "Loss and recovery of wings in stick insects," *Nature*, vol. 421, pp. 264–267, 2003.

[7] C. P. Shelton and P. Koopman, "Using architectural properties to model and measure system-wide graceful degradation," in *Proc. of the 2002 Workshop on Architecting Dependable Systems (ICSE'02)*, Orlando, Florida, USA, May 25 2002.

[8] R. Rojas, *Neural networks: a systematic introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1996.

[9] R. A. Brooks, "How to build complete creatures rather than isolated cognitive simulators," *K. VanLehn (ed.), Architectures for Intelligence*, pp. 225–239, 1991.

[10] P. F. M. J. Verschure et al., "Distributed adaptive control: The self-organization of behavior," *Robotics and Autonomous Systems*, vol. 9, no. 3, pp. 181–196, 1992.

[11] R. Salomon, "Achieving robust behavior by using proprioceptive activity patterns," *BioSystems*, vol. 47(3), pp. 193–206, 1998.

[12] M. J. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, vol. SE-1(4), pp. 364–370, 1975.

[13] W. F. Tichy, "RCS — a system for version control," *Software — Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[14] B. Berliner, "CVS II: Parallelizing software development," in *Proceedings of the USENIX Winter 1990 Technical Conf.* Berkeley, CA: USENIX Association, 1990, pp. 341–352.

[15] CollabNet, Inc., "CollabNet Subversion," available at http://subversion.tigris.org/.

[16] M. M. Kokar et al.r, "Control Theory-Based Approach of Self-Controlling Software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 37–45, May/June 1999.

[17] Gordon Brebner, "A virtual hardware operating system for the Xilinx XC6200," in *LNCS*, ser. 6th Intl. Workshop on Field Programmable Logic and Applications, vol. 1142. Springer, 1996, pp. 315–333.

[18] ——, "The Swappable Logic Unit: A Paradigm for Virtual Hardware," in *Proc. of the 5th IEEE Symp. on FPGA-Based Custom Computing Machines*, Napa Valley, California, 1997, pp. 77–86.

[19] R. Hecht et al., "Network-on-Chip basierte Laufzeitsysteme fuer dynamisch konfigurierbare Hardware," in *ARCS 2004 - Organic and Pervasive Computing, Workshops Proceedings*, March 26 2004.

[20] ——, "Dynamic Reconfiguration with hardwired Networks-on-Chip on future FPGAs," in *Proc. of the 15th Int. Conf. on Field Programmable Logic and Applications (FPL'05)*, Tampere, Finland, August 24-26 2005.

[21] ——, "A Distributed Object System Approach for Dynamic Reconfiguration," in *Proc. of the 13th Reconfigurable Architectures Workshop (RAW'06)*, Rhodes Island, Greece, April 25-26 2006.

[22] Dave Garlan et al., "RAINBOW: Architecture Based Self-Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[23] IBM Corporation, "An architectural blueprint for autonomic computing," white paper, June, 2005.