An Observer/Controller Architecture for Adaptive Reconfigurable Stacks

Thorsten Schöler and Christian Müller-Schloer

Institute of Systems Engineering – System and Computer Architecture,
University of Hannover, Appelstraße 4, 30167 Hannover
{schoeler,cms}@sra.uni-hannover.de

Abstract. In this paper, we discuss the necessity of new observation and control structures for organic computing systems starting from the basic contradiction between bottom-up behaviour and top-down design. An Observer/Controller architecture serves the purpose to keep emergent behaviour within predefined limits. As an illustration, a framework for reconfigurable protocol stacks is introduced, which contains an agent-based monitoring framework as well as a reconfiguration manager. After describing a TCP/IP protocol stack implementation, based on the framework, similarities between the introduced framework and the Observer/Controller architectural pattern will be pointed out.

1 Introduction and Outline

1.1 Organic Computing

Due to the increasing complexity of today's software systems, new ideas for the design and management of those systems have to be found. Most software architects still design their systems following the top-down approach, trying to master this increasing challenge. While a good part of software complexity stems from once well-meant ideas to tackle the complexity like reuse of object oriented components, a considerable part of the complexity is caused by the general attitude of engineers to keep control of all details of the system under design.

To ease this potentially dangerous situation, organic computing tries to mimic key phenomena observed in nature for computer systems, such as self-organisation, selfconfiguration, self-healing, self-protection, self-explanation, and context awareness. Thus, an organic computer is a technical system, which adapts dynamically to the current conditions of its environment [1] without being exactly programmed to certain environment changes.

An exciting example of an organic system is disclosed by a look into the human brain. Low-level structures (brain stem), which react on sensory inputs thus, implementing a subconscious stimulus-response pattern, can be considered as an execution function.

Higher-level structures (e.g. the limbic system), observe and manipulate this execution function in several ways (especially emotions and regulation of visceral motor activities) [2]. Initial discussions on organic computing systems [1, 3] suggest that a two-level architecture with a low level execution and a higher-level control layer represents a general pattern present in natural as well as artificial organic systems. We call

M. Beigl and P. Lukowicz (Eds.): ARCS 2005, LNCS 3432, pp. 139–153, 2005.

this architectural pattern Observer/Controller architecture. It will be described in this paper and illustrated with an example from telecommunication research.

1.2 Software Defined Radio, Mobile Adaptive Terminals

This paper will introduce a software framework for organic protocol stacks of mobile terminals, so-called software defined radio (SDR) devices. Therefore, a short introduction to SDR will be given in this section.

A software-defined radio is a system, which uses software for the modulation and demodulation of radio signals [4]. This normally includes the RF front end, the analogue/digital as well as the digital/analogue converters, and the base band processing (BB). In our research, we understand SDR in a much broader sense. An adaptive mobile terminal contains not only a reconfigurable radio system but also a reconfigurable operating system, reconfigurable protocol stacks, middleware, services and applications.

The idea of reconfiguration on software levels yields higher flexibility by better resource utilisation. For example, an adaptable mobile terminal is able to choose from different implementations of a multimedia CODEC, depending on the current resource situation. Such a terminal will favour a less energy consuming hardware CODEC when the battery power is getting low, whereas a CODEC implemented in software is preferred, when the availability of processing power in dedicated hardware (DSPs, FPGAs, etc.) is crucial and when there is plenty of CPU power at hand. In this paper, we will focus on the protocol stack software as a reconfigurable software system, which will benefit from organic computing ideas. Such a protocol stack can be used in SDR devices and other mobile terminals as well.

1.3 Outline

First, we want to introduce the Observer/Controller architectural pattern. We will begin with a short characterisation of the phenomenon of emergence and will point out the arising basic contradiction of top-down design vs. bottom-up development, which we will run into if we try to exploit emergence in technical systems. Then we will show how we can escape this contradiction, basically by combining creative bottom-up mechanisms with a mechanism which enforces top-down constraints.

An example for a complex software system – a reconfigurable protocol stack – will be used to explore analogies to organic feedback systems. We will describe the introduced protocol stack framework, its fundamental components, and exemplary protocol stack implementations.

Having described the theoretical background and the system implementation, we will review the implementation of the Observer/Controller structure. We will end up with implementation results, conclusions and an outlook on future work.

2 How Can We Control Emergence?

2.1 Self-organisation

Properties of complex systems have been investigated predominantly with the help of natural systems like dissipative structures [5], autocatalytic cycles [6] or ant hives. Lans-

ing and Kremer [7] have analysed the organisation patterns of rice growing on the island of Bali. Their assumption of a locality-based co-adaptation coordination algorithm ("Do as your best neighbour does!") leads to a simulated distribution of cropping patterns very similar to the one observed in reality. Ilya Prigogine [5] has investigated dissipative structures – chemical autocatalytic reactions far from thermal equilibrium –, which show self-organisation effects of high aesthetic appeal. Even technical systems like the Internet have been shown to reveal self-organisation patterns. A visualisation of communication patterns reveals a surprising degree of order although the Internet has developed without an explicit master plan. It has been shown that these traffic patterns are subject to the so-called small world effect [6, 8, and 9] which seems to be characteristic for complex communication systems like the brain.

Another experiment, which has been carried out with miniature robots as well as in simulations is that of the "candle movers". A robot is able to move one or two tea candles. If it encounters more than two it stops, turns in an arbitrary direction and moves straightly until it finds the next candles and so on. Intuitively, a random distribution of candles on a floor subject to a collection of randomly moving robots should result in a random distribution of candles. The experiments, however, show consistently that under these conditions the candles are assembled into a very small number of heaps. This is a nice example of very simple local rules leading – in cooperation with many autonomous components – to a global pattern exhibiting higher degrees of order.

Systems capable of self-organisation seem to have the following properties: They consist of *autonomous* processes, which use *local* information (context) for local behaviour. They develop by *evolutionary* mechanisms (recombination, selection, and mutation), using *trial and error*. This implies *large populations* of elements. The stability of such systems seems to increase with an increasing *interconnectedness* [10].

2.2 Emergence

A central notion of self-organising systems is the concept of emergence. Emergence is a property of our world. The development from chaotic starting conditions towards systems exhibiting higher degrees of order can be described as an effect of emergent behaviour. Emergence is defined as a property of a total system, which cannot be derived from the simple summation of properties of its constituent subsystems. Emergent phenomena are characterised by (i) the interaction of mostly large numbers of individuals (ii) without central control with the result of (iii) a system behaviour, which has not been "programmed" explicitly into the individuals (www.beart.org.uk/Emergent/).

An example of an emergent system property is the resonance frequency of a resonant circuit. It is a system property, which cannot be explained in terms of the properties of a single constituent. It develops from the dynamic cooperation of the capacitor and the inductivity.

Emergent behaviour in a complex technical system is an ambivalent property, especially if this system has safety-critical tasks. We must develop strategies, which leave sufficient degrees of freedom for self-organisation while keeping control over the emergent system to avoid unwanted results. This requirement leads to the problem of *controlled emergence* as discussed below. But how can we build emergent systems in the first place? There are quite a few "toy" systems showing creative behaviour in com-

puter simulations. Examples are Tom Ray's Tierra system [11], Karl Sims' "Virtual Creatures" [12], Lindenmayer Systems [13], or "Woods"-like environments [14]. Characteristic for these algorithms is the evolutionary or genetic paradigm: It works with large populations of individuals (representing solutions), random mutations, recombination and a selection mechanism based on an objective function. Due to their low speed, genetic algorithms [15] and similar approaches like simulated annealing or fuzzy classifier systems [16] have been used so far predominantly for off-line optimisation. It is a topic of future research to enable such algorithms to run under real time conditions and with restricted resources.

2.3 Top-Down vs. Bottom-Up

The classical top-down design process is based on the assumption that the developer is in principle able to predict all possible system states. In order to achieve this goal, the design process is organised strictly hierarchically. It consists of a sequence of modelling steps starting with a high level specification leading through a number of refinements finally to a model, which can serve directly to control a manufacturing machine or generate executable code. Today's technical systems, however, begin to show complexities to such an extent that a complete prediction becomes impossible.

Emergent behaviour reflects bottom-up constraint propagation. In the candle mover example, it is not possible to predict the exact positions of the robots or the candles but we get some similar kind of order every time we run the experiment. From the technical perspective it would be highly desirable to be able to predict more exactly the final outcome, in other words: We would like to be able to describe the relationship between local and global behaviour.

The top-down procedure is at the very heart of the engineering paradigm: We (the designers) set the goals, which have to be reached by the technical system. It is not very realistic to expect a collection of metal parts to assemble into a Mercedes car! This means that the exploitation of emergence within technical systems leads to a basic contradiction, namely the requirement of "controlled emergence". It is not clear today how controlled emergence can be realised but the Observer/Controller structure is a possibility.

2.4 Observer/Controller Architectures

The solution seems to be in the middle between pure top-down and pure bottom-up. In future, we expect a gradual increase of the degrees of freedom, which a technical system has during run time. The behaviour of a complex system will then be a combination of preset objectives and constraints, as defined by the system developer and adaptive "islands" where the system is allowed to make its own decisions. In order to develop such adaptive systems, we must introduce new system architectures, which allow the replacement of hard coding by goal setting (or motivation). This means, however, that our system now runs under the assumption of best effort and it can deliver sub-optimal results or even make mistakes. Hence we must take provisions to guide the system towards the optimum and guarantee that certain error conditions can never occur.

A basic structure supporting such a pattern is the Observer/Controller architecture. It borrows from a very simplified analogy to the human brain [17], where we find low level "circuits" (the brain stem) with the task to immediately react to sensory input (especially to pain in order to escape dangerous situations). These reactions occur subconsciously. Conscious decisions, generated by the neocortex, are filtered by certain areas of the interbrain – with the limbic system adding the emotional colouring –, before they are transformed into motion or other reactions by the brain stem. This filtering constitutes the influence of emotions on our actions. In case of a blocking decision by the limbic system the instruction from the neocortex is not executed.

In a technical system the Observer/Controller plays the role of the limbic system: It observes the external environment via the sensory input as well as the internal behaviour of the low level execution unit and manipulates it in several ways, as we will discuss below. The usage of Observer/Controller structures can be regarded as the introduction of emotions into technical systems. A computer might then decide (by means of its "limbic system") that it will not act as commanded by the user! Here, a word of caution seems appropriate: In a technical system we expect the goals to be set by the user (i.e. top – down) even if the exact procedure of execution is left to a creative lower level mechanism. In the brain it is not at all clear if the neocortex, the seat of consciousness, is the (single) origin of the commands. There seems to exist evidence that the brain stem could be the active part while the neocortex is asked for a second opinion from time to time [17].

The basic Observer/Controller structure is shown in Figure 1. In addition to the low level execution unit responsible for the working level stimulus/response mechanism, we have now a higher level Observer/Controller. The observer part receives input from the environment as well as from the execution unit (e.g. data about the present load conditions). The controller compares the situation reported by the observer to the goals set by the user and reacts by reconfiguring the execution unit.

In order to discuss this mechanism in more detail, we have to introduce a few terms:

- iStructure and iBehaviour mean the internal relationship of the components of a system and their procedural behaviour in time. An example is the composition of a CMOS inverter from an n-channel and a pchannel transistor (iStructure) and their cooperation in terms of electrical voltages and currents (iBehaviour).

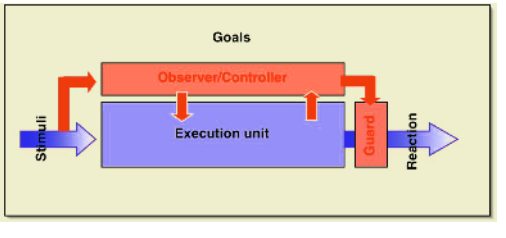

Fig. 1. Basic Observer/Controller structure. The controller intervenes by changing the behaviour of the execution unit and/or by setting filter functions in the guard to prevent erroneous outputs.

- eBehaviour (external behaviour) means the abstracted view from the outside on this system. In the above example of the inverter circuit the eBehaviour could be a logical view with one input, one output and the two states 0 and 1.[1]

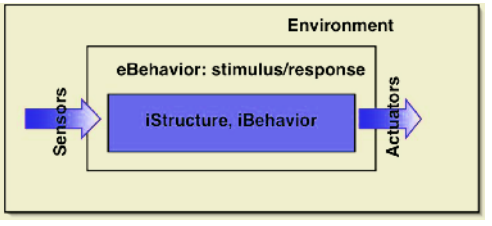

Fig. 2. Fixed system: The embedded system reacts to external stimuli. No modification of its external behaviour (eBehaviour) is possible at runtime. Internal structure and behaviour are fixed.

A conventional embedded system (such as an ABS[2] controller) shows a fixed external behaviour (eBehaviour in Figure 2) based on a fixed internal structure and specified internal behaviour (iBehaviour, iStructure): The ABS controller receives sensory input, processes it and reacts through actuators.

In a first step towards adaptivity we allow an external controller (possibly the human designer) to modify the eBehaviour by changing internal parameters of the system (Figure 3).

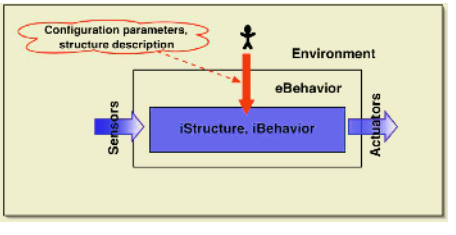

Fig. 3. Adjustable system: External (human) controller sets internal parameters such that the desired eBehaviour is reached.

Instead of modifying a large number of those parameters, it would be more desirable for the (human) external controller to declare or specify the eBehaviour on an higher abstraction level and leave it to an internal controller to translate those high level goals to low level parameter changes (Figure 4). These goals are comparable to motivations if we regard our technical system as an autonomous "animal". Now it is the task of the internal controller to permanently check the consistency of the current goals, the state of the environment and the internal state of the embedded system (like error or over-

[1] There is also an external structure (eStructure), which we will neglect for simplicity.
[2] Anti Lock Braking system.

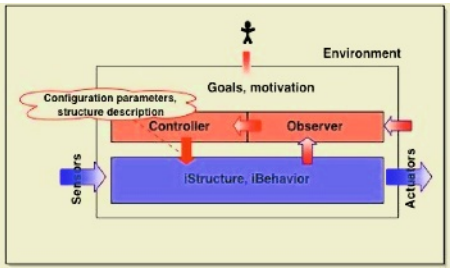

Fig. 4. Constant goals (homeostasis) are maintained by the Observer/Controller.

load conditions). In analogy to living systems we can speak of homeostasis[3]. This adds an additional higher level feedback loop to the system, in addition to the lower level productive feedback loop responsible for the actual function (e.g. the ABS system).

It remains largely open at his point in time how we can formulate the high level goals and how they are transformed into low level structural and behavioural changes. A promising approach to this problem is based on the software technology of design-bycontract and assertions (as introduced by [18]). Assertions are if-then rules inserted into code. The if-part can contain logical ($a \leq 5$) or timing (event2 @ 5 ms after event1) conditions. The violation of an assertion leads to a reaction as specified in the then-part. The reaction could simply be a call to the observer, which takes appropriate action. The assertion mechanism has been shown to be very useful especially in combination with system simulations [19] since in a simulated virtual prototype, we can instrument not only the software with assertions, but also the modelled hardware.

A second solution to the control of emergent mechanisms is the introduction of a guard. A guard receives the output from the adaptive control mechanism and filters it according to rules set by the Controller. An example for such a filtering action would be the detection of the creative attempt of a learning traffic light controller to set all traffic lights to "green". A reinforcement learning mechanism will probably punish this attempt but in a real world situation we must guarantee that such dangerous outputs are not realised.

The system behaviour – when following the Observer/Controller architectural pattern – will be a combination of preset objectives and constraints as well as adaptive "islands" where the system is allowed to make its own decisions (and errors!).

To implement an organic system, hard coded behaviour must be replaced by fuzzier goal settings, which describe the overall systems motivation leading to additional degrees of freedom. This should lead to a satisfactory behaviour even under unprecedented conditions but it may also yield suboptimal results or even allow the system to make errors. For the research of organic computer systems, an observable system described

[3] Homeostasis is one of the most remarkable and most typical properties of highly complex open systems. A homeostatic system (an industrial firm, a large organization, a cell) is an open system that maintains its structure and functions by means of a multiplicity of dynamic equilibriums rigorously controlled by interdependent regulation mechanisms. Such a system reacts to every change in the environment, or to every random disturbance, through a series of modifications of equal size and opposite direction to those that created the disturbance. The goal of these modifications is to maintain the internal balances (http://pespmc1.vub.ac.be/HOMEOSTA.html).

by internal structure and behaviour must be found and extended with observer and controller components. The following sections will introduce a reconfigurable protocol stack architecture as exemplary execution unit of an organic computer system. Subsequently it will be shown, that an agent-based monitoring system is capable of observing the protocol stack structure and behaviour. Furthermore, the described reconfiguration component implements the controller entity as projected in the Observer/Controller architectural pattern.

3 Reconfigurable Protocol Stack Architecture

The following section describes a reconfigurable protocol stack architecture similar to architectures like [20], but specially tailored for mobile terminals [22] and enhanced with ideas from organic computing. After a general overview, basic framework components will be described, enabling the design and implementation of actual protocol stack instances, which in turn are supervised by an agent-based monitoring framework.

3.1 Overview

Modern mobile terminals are supporting a huge number of various communication standards such as GSM, GPRS, EDGE, WLAN, Bluetooth, and so on. To manage the increasing software complexity, new approaches for designing such complex software systems and for managing such systems during run-time have to be found.

Supported by the depicted framework approach, the software designer has an extensive library of components and modules at his hands to choose from whilst implementing actual protocol stack software. The experience gained so far shows that building protocol stacks from a component library specially tailored for protocol stack design, eases the complex implementation task because the implementer can concentrate on protocol stack specific parts or components. Furthermore, a protocol stack library of generic components shortens implementation time as well as debugging time.

During run-time, the framework offers the ability to exchange protocol stack modules on the fly without loosing network connection. Protocol stack modules in the portrayed framework are much more fine granular than e.g. kernel modules of the Linux kernel. This offers a much higher degree of reconfiguration options of the protocol stack structure. For example, in the proposed TCP/IP implementation, there are only four layers. Assumed that there are three different implementations to choose from, this leads to $3^4 = 81$ possible configurations. In future, reconfiguration will be supported on class level, yielding a barely limited number of possible configurations. In addition, to fix errors or just to update to a newer version, protocol stack modules even can be downloaded from a device management server situated somewhere in the network.

3.2 Protocol Stack Framework

The protocol stack framework, as shown in Figure 5, consists of three major components. The framework itself, providing management and supervision functionalities, the component library, providing the protocol stack modules and the protocol stack instances itself, executing in the framework.

The following paragraphs describe the major framework components in detail.

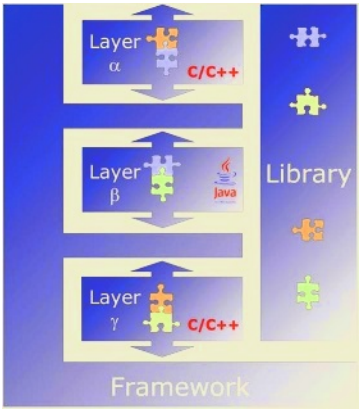

Fig. 5. Software architecture of protocol stack framework.

3.2.1 Framework Component

The protocol stack framework bundles all of its management and supervision functionality in the framework component. The main framework component is the configuration and control manager. The configuration manager parses XML descriptions of protocol stacks. A protocol stack description is a graph consisting of nodes (modules, layers) and edges (connectors). Layers have to implement defined data and control interfaces to become exchangeable. Furthermore, the configuration manager decides which protocol stack will be constructed from the textual descriptions to satisfy the applications request for a certain network connection. The matching protocol stack for the requested network connection is matched by using QoS[4] parameters that an application requests by opening a network socket.

Applications running on top of the protocol stack use a socket interface (similar to the BSD socket interface [21]) to communicate with the network. For protocol stack internal message passing, the framework offers data message passing functionality as well as a mailbox system for control message passing between protocol stack layers. Because the framework uses the thread-per-message model for internal data message passing [22], thread management is another major functionality implemented by the protocol stack framework.

Another special component of the framework is the monitoring component, which will be described in detail in Section 3.4.

3.2.2 Library

The framework library provides generic classes for basic tasks, which are carried out in every protocol stack, implemented for the framework. Provided classes range from simple byte arithmetic/manipulation, checksum, fragmentation classes up to data and control message passing classes. Further to the generic classes already provided in the library, specific protocol stack classes can be stored in the library or downloaded over the air from a configuration management server as well and can be subsequently used in protocol stack configurations as well.

[4] Quality of Service.

3.2.3 Protocol Stack Instances

Protocol stack instances, as created by the configuration manager, may contain any combination of modules interconnected by defined connectors. Due to the fact, that traditional protocol stacks are implemented in native programming languages such as C and C++, native protocol stack modules are supported by implementing a module wrapper. Such a wrapper is a Java class implementing the necessary data and control interfaces, connecting the native implementation to the stack framework by JNI[5]. A number of protocol stack configurations, some of them contain only Java modules, some of them contain Java modules and wrapped native modules, have validated the described mechanism.

3.3 Protocol Stack Examples

As mentioned before, protocol stack instances are built from generic components found in the frameworks library as well as protocol stack specific components. The protocol stack architecture has been validated with a number of TCP/IP protocol stack configurations. Figure 6 shows an example TCP/IP protocol stack configuration.

The example TCP/IP protocol stack consists of four layers derived from generic library components (Layer class). The partitioning is more or less carried out according to the ISO OSI seven-layer model [23]. There is a dedicated layer for each of the TCP/IP protocol suite protocols. The application communicates with the protocol stack using a BSD socket interface. The framework "fingers", between adjacent layers, provide monitoring and supervision capabilities, which help in detecting degradation in network performance or rogue terminal[6] behaviour as achieved by the agent-based monitoring component.

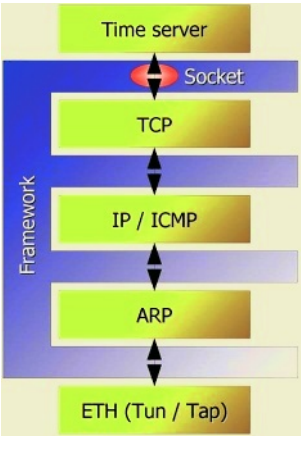

Fig. 6. TCP/IP protocol stack as configured by the protocol stack framework.

[5] Java Native Interface.

[6] The term rogue terminal stems from computer terminals or ID card terminals becoming rogues. They try to steal resources (i.e. network capacity, bandwidth), confidential information (i.e. passwords, encryption keys), or try to harm the network, user or equipment.

Various TCP/IP stack configurations have been validated by prototype implementations. The reconfiguration of protocol stack instances has been exemplarily validated by exchanging the ARP[7] layer during run-time. The ARP layer exists in two implementations: A pure Java implementation and an ARP implementation in C (wrapped in Java), which have been exchanged during run-time without loosing the applications open socket connection.

The reconfiguration process has been triggered by the agent-based monitoring framework, which will be described in the following section.

3.4 Agent-Based Monitoring

The main characteristic of an organic computer is its ability to adapt dynamically to current conditions or its environment [1]. For that, a technical system has to contain a component, which monitors the systems current internal structure and behaviour. For the protocol stack framework, this task is carried out by the agent-based monitoring component as seen in Figure 7.

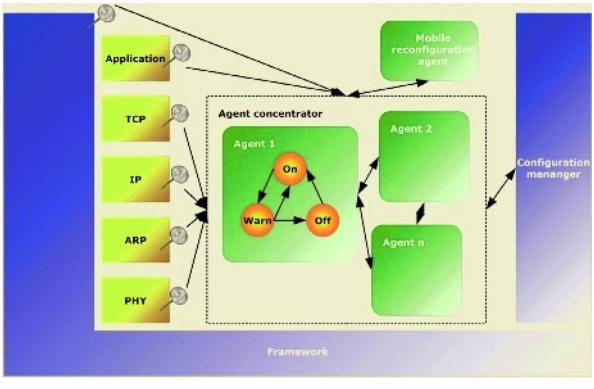

Fig. 7. Agent-based monitoring software architecture.

The main component inside the agent-based monitoring framework is the so-called agent concentrator. As shown by Tarkoma and Laukkanen in [24], agents do not have to be heavy-weight, they can be suitable for resource-constraint mobile devices as well. By dynamically concentrating one or more light-weight agents in a networked manner, the characteristic features of agents, such as autonomy, adaptiveness, collaborative behaviour, and mobility [25] can be utilised for processing observed structure and behaviour information, gained from the currently installed or executing protocol stack instances. In the depicted protocol stack framework, the agents observe communication and thread status information (internal behaviour) such as sent and received packets/bytes and number of send/receive threads currently executing in the framework. Furthermore, the agents can request the current protocol stack configurations from the configuration manager (internal structure) to decide how they have to be reconfigured.

[7] Internet Address Resolution Protocol as described in RFC 826 (Request For Comments 826, Internet Standards).

Observable internal protocol stack structure or behaviour information is produced by socalled software probes, provided by the framework, and inserted into executing protocol stack instances. The agents in turn evaluate the information they have subscribed for by the observer pattern (see Section 4.1 for details).

For implementation of the internal behaviour, we recommend a finite state machine model. For that, the monitoring agent framework provides generic agent, state and transition classes, as well as classes, which can host rule-based decision-making algorithms. This approach is lightweight enough to allow multiple agents to execute on mobile terminal platforms.

The intelligence of the reasoning system is currently captured in rules that contain conditions and actions. Because rule-based systems are limited in their level of intelligence, smarter reasoning algorithms like classifier systems [26] can be easily integrated into the connected agent's finite state machine approach. Even external agents (provided by third parties) can be integrated by over-the-air download/synchronisation from a device management server.

The protocol stack framework configuration manager component provides a reconfiguration interface for reconfiguration actions, issued by the agent-based monitoring system. Currently executing protocol stack instances can be tuned (e.g. change parameters, etc.) or even structurally reconfigured (e.g. exchange of protocol stack layer implementations) during run-time. Completely new protocol stack configurations can be requested as well, in order to optimise network communication performance or to accomplish the user preferences better.

4 Observer/Controller Implementation

As mentioned in Section 1.2, a closed loop control pattern, consisting of a controlled process, sensors, an observer and a controller, can be typically found in organic computing systems. The main components of the protocol stack architecture can be mapped quite naturally onto Observer/Controller entities.

An executing protocol stack instance can be regarded as observed process. The analysis of protocol stack structure and run-time information, such as parameters, stack status, statistics, configuration, etc., will be carried out be the agent concentrator component and its intelligent agents. Whenever the multi agent system concludes, that the protocol stack's current structure or behaviour is not optimal, it informs the protocol stack framework configuration manager (controller component). The configuration manager in turn can then change configuration, parameters, and structural description of the executing protocol stack instance.

Although the user is still in control of the protocol stack framework by issuing abstract goals, motivations, or preferences, such as QoS requirements like bandwidth, accepted costs, etc., the organic computing system has taken over the fine-tuning of concrete protocol stack configurations and their settings from the designer or user of the system.

4.1 Observation: Implementation for Monitoring Agents

As seen in the last section, the agent concentrator component of the protocol stack framework implements the observer component of the earlier introduced Observer/Controller architectural pattern.

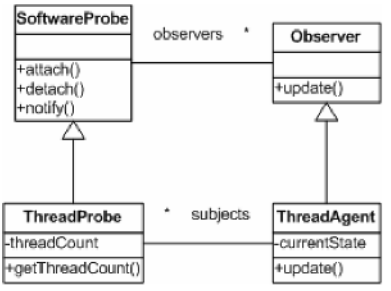

Fig. 8. Observer pattern as implemented by ThreadProbe and Thread-Agent.

The concrete monitoring agents, accommodated in the agent concentrator, are part of the observer pattern [27] for getting system status information from software probes. As seen in Figure 8, the observable object ThreadProbe is inherited from the abstract SoftwareProbe class. Observers like the ThreadAgent (inherited from the abstract Observer class) can subscribe to SoftwareProbe objects by calling the SoftwareProbe.attach() method. If the information stored in a SoftwareProbe e.g. ThreadProbe changes, all attached Observers will be informed by calling their update() method.

The use of the observer pattern eases the attachment and detachment of monitoring agents to software probes and reduces the information distribution complexity. Furthermore, monitoring agents again can become observable objects, allowing other agents to subscribe to their processed information. This mechanism is the foundation for building complex multi agent networks inside the agent concentrator component for monitoring the protocol stack instances.

4.2 Control: Implementation of a Configuration Manager

The controller component of the Observer/Controller architectural pattern is responsible for actually changing the processes internal behaviour and structure by issuing new parameters or process configurations. In the depicted protocol stack architecture, the reconfiguration manager component is responsible for tuning and configuring protocol stack instances. In the current approach, the actual decision that a reconfiguration is necessary is currently completely taken by the monitoring agents (observer). This leaves only the execution of a reconfiguration to the reconfiguration manager (controller). The reconfiguration manager consequently only implements the generation of protocol stack instances from given protocol graphs, stored in the XML protocol stack descriptions. Furthermore, the reconfiguration manager implements additional methods, which provide information about possible protocol stack configurations, and their properties to assist the monitoring agents in their reconfiguration decision.

The Observer/Controller architecture pattern also allows splitting the decision-making in a different way. The decision-making can be carried out completely in the controller entity; mixed approaches are feasible as well.

5 Conclusion and Outlook

The concept of observers and controllers has been introduced in this paper. A framework for reconfigurable protocol stacks and its agent-based monitoring framework have

been portrayed. It has been shown, that the Observer/Controller architectural pattern is a natural way of implementing an organic, self-monitoring, self-organising and selfconfiguring software system. Each major component of the protocol stack framework has a matching counterpart in the Observer/Controller architectural pattern.

The current implementation status of the protocol stack framework shows, that the framework approach is feasible and furthermore speeds up implementation for future protocol stacks. Due to similarities in protocol stack design, the generic components from the library are a welcome simplification of the implementation process. Specific protocol stack functionalities were easily implemented based on the framework. The prototype TCP/IP implementation has served as a good validation platform for the portrayed ideas and will serve in future as a valuable platform for further research.

The described agent-based monitoring implementation has demonstrated, that intelligently controlled reconfiguration of a protocol stack is possible and will yield performance advantages. It also decreases the user's effort for performance corrections.

It has also become clear that the degree of freedom, which the "creative" part of the reconfigurable stack architecture is given, should be increased. We are considering to using a rule-based approach like fuzzy classifier systems for this purpose. But it must also be clear that there exists a delicate balance between increased freedom and security concerns. Future work in the protocol stack framework will comprise the integration of more intelligent reasoning technologies into the agent-based monitoring. Accompanying the work on the organic capabilities, the framework will be extended to support more networking standards and protocols (Bluetooth, WLAN, etc.).

References

1. Müller-Schloer, C.: Organic Computing – On the Feasibility of Controlled Emergence. Proceedings of CODES+ISSS 2004. ACM. Stockholm. September 2004.
2. Brainexplorer. Available at http://www.brainexplorer.org. September 2004.
3. Herkersdorf, A., et. al.: Towards a Framework and a Design Methodology for Autonomic Integrated Systems. In Dadam P., Reichert, M., editors, Proceedings of the Workshop on Organic Computing, Informatik 2004, Ulm, September 2004.
4. Software-defined radio (SDR). Available at http://en.wikipedia.org/wiki/Softwaredefined_radio. September 2004.
5. Prigogine, I. and Kondepudi, K.: Modern Thermodynamics: From Heat Engines to Dissipative Structures. John Wiley & Sons, Chichester, 1998.
6. Albert, R. and Barab'asi, A.: Statistical mechanics of complex networks. Review of Modern Physics, 74:47, 2002.
7. Lansing, J. and Kremer, J.: Emergent properties of balinese water temple networks: Coadaptation on a rugged fitness landscape. In C. Langton, editor, Proceedings of the Workshop on Artificial Life (ALIFE '92), (Santa Fe, NM, USA, June 1992), Reading, MA, 1994. Addison-Wesley.
8. Cohen, D.: All the world's a net. New Scientist, 174(2338):24, April 2003.
9. Mendes, J. and Dorogovtsev, S.: Evolution of networks: from biological nets to the internet and WWW. Oxford University Press, 2003.
10. Prigogine, I. and Stengers, I.: Dialog mit der Natur, page 181. Piper, 1990.
11. Ray, T. S.: An approach to the synthesis of life. In C. Langton, C. Taylor, J. Farmer, and S. Rasmussen, editors, Artificial Life II, volume X of SFI Studies in the Sciences of Complexity. Addison-Wesley, Redwood City, 1991.

12. Sims, K.: Evolving virtual creatures. In Computer Graphics (Siggraph '94 Proceedings), New York, July 1994. ACM Press.
13. Prusinkiewicz, P. and Lindenmayer, A.: The algorithmic beauty of plants. Springer Verlag, New York, 1990.
14. S. Wilson. Classifier fitness based on accuracy. Evolutionary Computation, 3(2):149–176, 1995. (Section 4.3).
15. Goldberg, D.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, Massachusetts, 1989.
16. Koblitz, D. and Müller-Schloer, C.: Extension of fuzzy classifier-mechanisms for adaptive embedded systems through a-priori-knowledge and constraints. In D. Polani, J. Kim, and T. Martinetz, editors, Fifth German Workshop on Artificial Life – GWAL-5, Berlin, 2002. Akademische Verlagsgesellschaft Aka.
17. Roth, G.: Aus Sicht des Gehirns. Suhrkamp, 2003.
18. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, 2nd edition, 2000.
19. Oodes T., Krisp, H., and Müller-Schloer, C.: On the combination of assertions and virtual prototyping for the design of safety-critical systems. In ARCS 2002/Trends in Network and Pervasive Computing (Karlsruhe). Springer, 2002.
20. Schmidt, D. C. et al.: ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. Journal of Concurrency: Practice and Experience 5 (4), 269–286. June 1993.
21. Stevens, W. R.: Unix Network Programming. 1990. Prentice-Hall.
22. Schöler, Th., et. al.: Design, Implementation and Validation of a Generic and Reconfigurable Protocol Stack Framework for Mobile Terminals. Workshop on Dynamic and Reconfigurable Architectures DARES 2004. Hachioji. 2004.
23. Zimmerman, H.: OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications COM-28, No. 4: April 1980.
24. S. Tarkoma and M. Laukkanen. Supporting software agents on small devices. In AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems, pages 565–566. ACM Press, 2002.
25. Feldman, S. and Yu, E.: Intelligent agents: A primer. Available at http://www.infotoday.com/searcher/oct99/feldman+yu.htm. October 1999.
26. Richards, R. A.: Zeroth-order shape optimization utilizing a learning classifier system. Section 3.1 3.1 Introducing the Classifier System. Available at http://www.stanford.edu/ buc/SPHINcsX/. 1995.
27. Gamma, E., et. al.: Design Patterns: Elements of Reusable Object Oriented Software. Addison Wesley Longman, Inc. October 1994.