Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 182 (2007) 201–217
www.elsevier.com/locate/entcs

Self Management and the Future of Software Design

Peter Van Roy[1,2]

Department of Computing Science and Engineering
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Abstract

Most software is fragile: even the slightest error, such as changing a single bit, can make it crash. As software complexity has increased, development techniques have kept pace to manage this fragility. But today there is a new challenge. Complexity is increasing rapidly as a result of two factors: the increasing use of distributed systems as a result of the sufficient reliability and bandwidth of the Internet, and the increasing scale of these systems as a result of the addition of many new computers to the Internet (e.g., mobile phones and other devices). To manage this new complexity, we propose an approach based on *self-managing systems*: systems that can maintain useful functionality despite changes in their environment. The paper motivates this approach and gives some ideas on how to build general self-managing software systems. An important part of the approach is to build systems as hierarchies of interacting feedback loops. We give examples of these systems and we deduce some of their design rules. The SELFMAN project is elaborating these ideas into a programming methodology and an implementation.

Keywords: Software development, self management, distributed system, complexity

1 Introduction

Software is fragile and highly nonlinear: even a minor error can have catastrophic effects. Major disasters have occurred due to minor errors such as omitted commas in Fortran programs or changed bits because of alpha rays [11]. So far, this has not unduly hampered the quantity of software being developed. As software complexity has increased, software development techniques have kept pace. This situation is analogous to the Red Queen's behavior in Alice [10]: we are running as fast as we can in order to stay in the same place. Software development is now facing a new

[1] This paper is intended to stimulate discussion; all comments are welcome! This work is funded by the European Union in the SELFMAN project (contract 34084), EVERGROW project (contract 001935), and CoreGRID network of excellence (contract 004265). We thank Luis Quesada, Boriss Mejias, Raphaël Collet, Yves Jaradin, Kevin Glynn, and Seif Haridi for comments on drafts of this paper.

[2] Email: pvr@info.ucl.ac.be

doi:10.1016/j.entcs.2006.12.043

challenge: complexity is increasing quickly because of two reasons. First, the reliability and bandwidth of the Internet infrastructure has reached a point where it is feasible to build large distributed applications. Examples of such applications include a wide variety of file-sharing programs (Napster, Gnutella, Morpheus, Freenet, Bit Torrent, etc.), collaborative tools (Skype and other messenger tools), Massive Multiplayer Online Role Playing Games (MMORPGs) (World of Warcraft, Dungeons & Dragons, etc.) and research testbeds (SETI@home [25], PlanetLab [12], etc.). Technologies for building such applications now exist, e.g., Web services and Grid software. The second reason is the increase in the number of small devices connected to the Internet. For example, mobile phones are now full-fledged computing nodes with Internet connectivity, and protocols such as Zigbee, Bluetooth, and Wifi facilitate network connectivity among small devices.

How can we address the problem of programming large-scale distributed systems? Such systems have new properties that greatly increase the complexity of programming: scale (large numbers of independent nodes), partial failure (part of the system fails), security (multiple security domains), resource management (resources are localized), performance (harnessing multiple nodes or spreading load), and global behavior (emergent behavior of the system as a whole). Each of these properties has been studied in isolation. For example, the area of distributed algorithms has solutions for handling partial failure in many cases. But the properties have not been looked at together. The purpose of this paper is to give some ideas how this can be done.

Global behavior is particularly relevant for large systems. They must be designed carefully, otherwise the system will not behave well when stressed. Ideally, it should converge rapidly to its desired behavior and stay there despite changes in the system's environment. But it may instead collapse, oscillate, or show chaotic behavior. Such erratic behavior has been observed for power grids and has resulted in large-scale power outages [15]. One reason for this is because the power grid's behavior was designed for a situation close to equilibrium; it was not studied far from equilibrium.

2 Self-managing systems

To build large-scale distributed systems with good behavior, we need a framework in which to think about them. What should such a framework look like? To reduce the complexity of the system, it should be able to manage its own problems as much as possible. This leads us to propose self-managing systems as a suitable framework. A self-managing system is one that can maintain its functionality despite changes in its environment, in a general sense.

Self-managing systems have recently been brought to the forefront because of IBM's Autonomic Computing initiative [19]. When computer systems become large then the cost of managing them becomes prohibitive. The initiative aims to reduce this cost by removing humans from the management loop. The role of humans is then to manage the policy and not to maintain the mechanisms. This greatly

reduces the need for manual intervention.

Another area that is building self-managing systems is structured overlay networks [1]. This research is inspired by the popular protocols of peer-to-peer networks. Many of the applications mentioned in the introduction are based on these peer-to-peer networks. Unlike peer-to-peer networks based on random neighbor communication, structured overlay networks provide both guarantees (information is guaranteed to be found if it exists) and efficiency (broadcast does not flood the network as it does in, e.g., random neighbor networks such as the one used in Gnutella). Structured overlay networks provide primitive self-managing behavior: they reorganize themselves to maintain their functionality in reaction to environmental changes such as failures and overloads. Structured overlay networks have led to robust software that is being used in various areas, such as the construction of robust distributed communication networks and robust storage services that continue to provide service despite high node turnover (node "churn").

These two research areas, autonomic systems and structured overlay networks, have attracted attention once again to self-managing systems. But self-managing systems are actually a very old idea. The beginning of the area as a discipline can be dated to the definition by Norbert Wiener of cybernetics in the 1940's [29] and by Ludwig von Bertalanffy of general system theory in the 1960's [5]. The basic idea of system theory is to study the concept of a *system*, its properties and design. There are various ways to define the concept of a system [24]. For this paper, we define a system recursively as a set of components (called subsystems) connected together to form a coherent whole. The main problem is to understand the relationship between the system and its subsystems: can we predict the system's behavior and can we design a system with a desired behavior.

System theory is still very much in its early stages. Recent research results have not been systematized in a textbook and the ideas have not been applied to computer science in a systematic way. W. Ross Ashby wrote an introductory textbook in 1956 that is still worth reading today [4]. Gerald M. Weinberg wrote an introduction in 1975 explaining how to use system theory to improve general thinking processes [28]. In the area of computer systems, textbooks exist only for specialized subfields such as distributed algorithms [21]. We consider that it is high time to apply system theory to software construction. This paper gives examples of realistic systems to motivate this goal and to explore how to build software according to system theory.

3 Designing self-managing systems

How does one design a self-managing software system? We do not yet have a general set of design techniques, but we can talk about several important aspects: feedback loops, global properties, and a general architectural framework. It turns out that designing with feedback loops is fundamental. Feedback loops are currently being used for the autonomous management of computing clusters, for example they are being used in J2EE clusters [6] and Grid systems [2]. But feedback loops are much

more generally applicable in system design. We give examples of systems built with feedback loops to see what they can teach us for the general case. The paper by Andrzejak *et al* [2] gives a broad introduction to the different disciplines that can be useful when designing adaptive systems with feedback loops. The present paper is narrower: it restricts itself to the architectural questions of how the loops are organized and how they interact with each other and with distributed programming.

3.1 Feedback loops

The notion of a *feedback loop* is a basic element of system theory. A feedback loop consists of three elements that together interact with a subsystem (see Figure 1): an element that monitors the state of the subsystem, an element that calculates a corrective action, and an element that applies the corrective action to the subsystem. For the purposes of this paper, we consider these elements to be concurrent software agents that interact through asynchronous message passing. The complete system can be described as a graph of interacting feedback loops. Feedback loops can interact in two main ways. The simplest interaction is where both loops affect interdependent system parameters, i.e., they interact through their environment. This is called stigmergy. A second form of interaction is where a loop manages another loop, i.e., the first loop continuously adapts the policy implemented by the second loop. In both cases, the system's global behavior depends on all the feedback loops taken together.

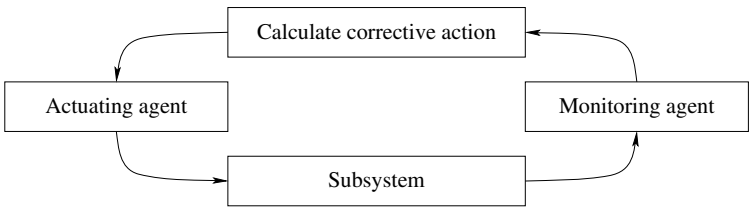

Fig. 1. Basic structure of a feedback loop

3.1.1 Two simple examples

The first example is taken from Wiener [29] and is shown in Figure 2. It consists of two interacting feedback loops with counterintuitive global behavior: in an air-conditioned hotel, a primitive tribesman attempts to warm himself by starting a fire. This causes the airconditioning to work harder, so the result is that the harder he stokes the fire, the lower the temperature becomes. In this example, the two loops affect system parameters that depend on each other, namely the temperature in different parts of the lobby. Each block in the figure is a concurrent agent continuously sending asynchronous messages to the other agents in the direction of the arrows. Even though each loop taken in isolation uses negative feedback and

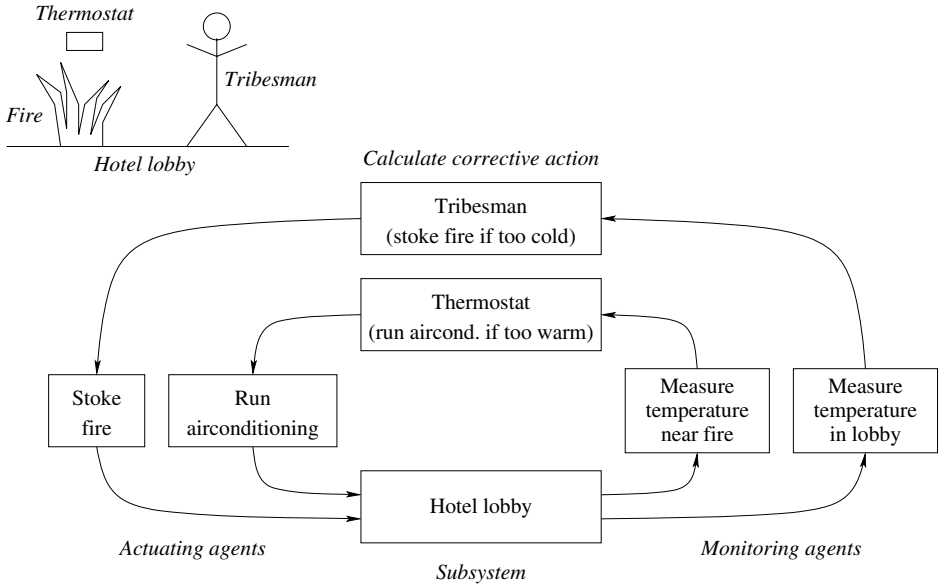

Fig. 2. Wiener's example of two feedback loops interacting through stigmergy

is stable,[3] the result of both loops is that the system becomes unstable, i.e., the temperature will continue to decrease (until the system reaches a boundary, and then its behavior will change again). We conclude that it is not enough to add a negative feedback loop to an existing system to ensure stability! The result may well be unstable because of the new loop's interaction with the system.

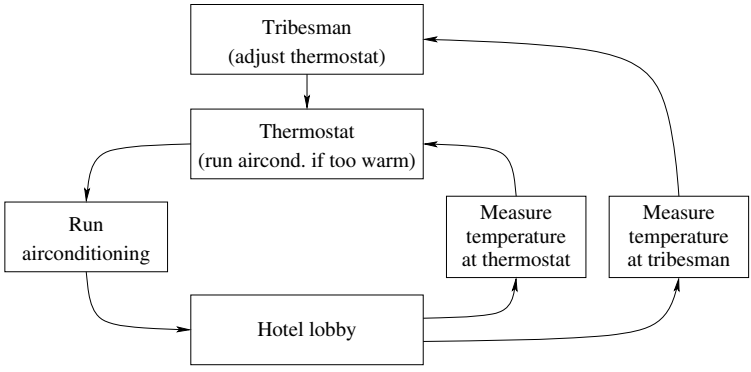

Fig. 3. Wiener's example modified to use management instead of stigmergy

The correct solution is given in Figure 3. Instead of starting a fire, the tribesman simply adjusts the thermostat. This maintains the stability of the airconditioning loop. This is an example of one loop managing another. This illustrates a design

[3] In negative feedback, an increase in the monitored value of a system parameter causes a corrective action that decreases the system parameter. In positive feedback, the corrective action increases the system parameter.

rule: to modify a system's behavior, the right way is to work with the system and not to try to bypass it.

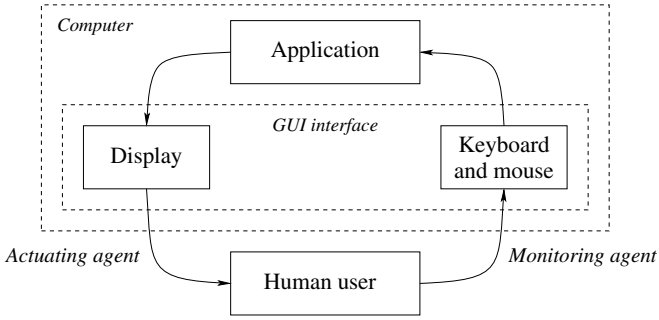

Fig. 4. A single-user application shown as a feedback loop

The second example is shown in Figure 4. This shows a generic single-user application as a feedback loop structure. We give this example to illustrate that feedback loops are generally useful in programming and not just for contrived examples such as Figure 2. Feedback loops are omnipresent in software systems if one looks with the right mindset. The three elements of the loop in Figure 4 all run on a single computer, and the subsystem being managed is a human user. The monitoring and actuating agents are the computer's GUI interface. Remark that we consider the user and not the application to be the managed subsystem. This viewpoint is advantageous because it lets us extend the feedback loop structure in interesting ways. We can put a second loop around the first to monitor the application's behavior and apply corrections if something goes wrong. When the user runs two applications and passes information between them then we have two loops interacting through stigmergy. The rest of this paper gives more substantial examples of systems shown as feedback loop structures, including systems that were not originally conceived in this way.

3.1.2 Using program properties

Designing systems with feedback has been extensively studied in electronics, typically with building blocks such as operational amplifiers and phase-locked loops. These systems exploit the fact that there is a good (piecewise) linear approximation of the building blocks' behavior. This is a strong condition that can be exploited. But linearity is probably too strong a condition to impose on computer systems, which are highly nonlinear by default, e.g., changing a single bit can have major effects. It may be possible to use a weaker property than linearity that can be satisfied by computer systems and that gives a satisfactory design theory. The approach then is to choose first a property that facilitates reasoning about the program and its global behavior, and then to build a program that satisfies the property. This can greatly simplify program design. Note that one possible failure mode is that the property itself no longer holds.

One example property is monotonicity or strict monotonicity. In a strict monotonic system, when the input changes in one direction (e.g., increases, in a general sense), the output will also change in the same direction. Using monotonicity as the basic property is sufficient for designing systems with feedback. A negative feedback amplifier can be built using strict monotonicity. Another property weaker than linearity that may be useful is continuity, but continuity is in general not enough to guarantee stability. We note that two further properties that may be useful in a theory of feedback program design are determinism and confluence.

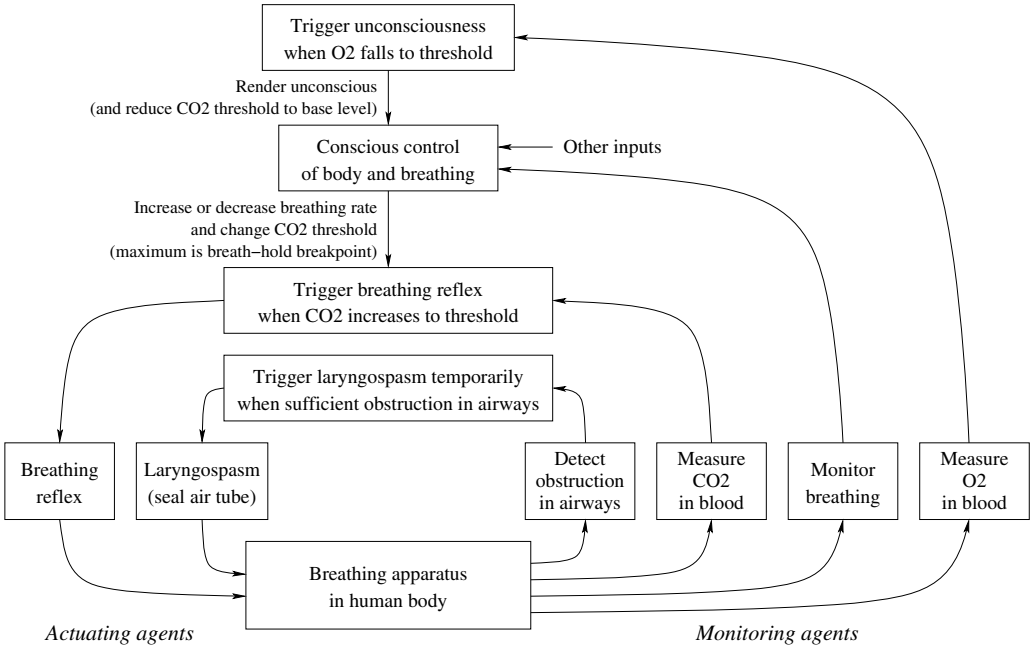

Fig. 5. Feedback loop structure of the human respiratory system

3.2 System design with feedback loops: the human respiratory system

Let us give a detailed example of a practical design that uses feedback loops. Our example is taken from a biological system, namely the human body. Biological systems have to survive in natural environments, which can be particularly harsh. For that reason, we consider that studying biological systems is a useful way to get insight in how to design software for a more complex system. Our example is the human respiratory system. Figure 5 shows the different components of this system and how they interact. We derived this figure from a precise medical description of the system's behavior [31]. The figure is slightly simplified when compared to reality. We have left out interactions with the rest of the body. Nevertheless it is complete enough to give many insights. There are four feedback loops: two inner loops (breathing reflex and laryngospasm), a loop controlling the breathing reflex (conscious control), and an outer loop controlling the conscious control (falling

unconscious). From the figure we can deduce what happens in many realistic cases. For example, when choking on a liquid or a piece of food, the larynx constricts and we temporarily cannot breath (this is called laryngospasm). We can hold our breath consciously: this increases the CO_2 threshold so that the breathing reflex is delayed. If you hold your breath as long as possible, then eventually the breath-hold threshold is reached and the breathing reflex happens anyway. A trained person can hold his or her breath long enough so that the O_2 threshold is reached first and they fall unconscious without breathing. When unconscious the normal breathing reflex is reestablished.

We can infer some plausible design rules from this system. The innermost loops (breathing reflex and laryngospasm) and the outermost loop (falling unconscious) are based on negative feedback using a monotonic parameter. This gives them stability. The middle loop (conscious control) is not stable: it is highly nonlinear and may run both with negative or positive feedback. It is the most complex of the four loops by far. We can justify why it is sandwiched in between two simpler loops. On the one side, conscious control manages the breathing reflex, but it does not have to understand the details of how this reflex is implemented. This is an example of nested feedback loops that implement abstraction. On the other side, the outermost loop overrides the conscious control so that it is less likely to bring the body's survival in danger. Conscious control seems to be the body's all-purpose general problem solver: it appears in many (but not all) of the body's feedback loop structures. This very power means that it needs a check.

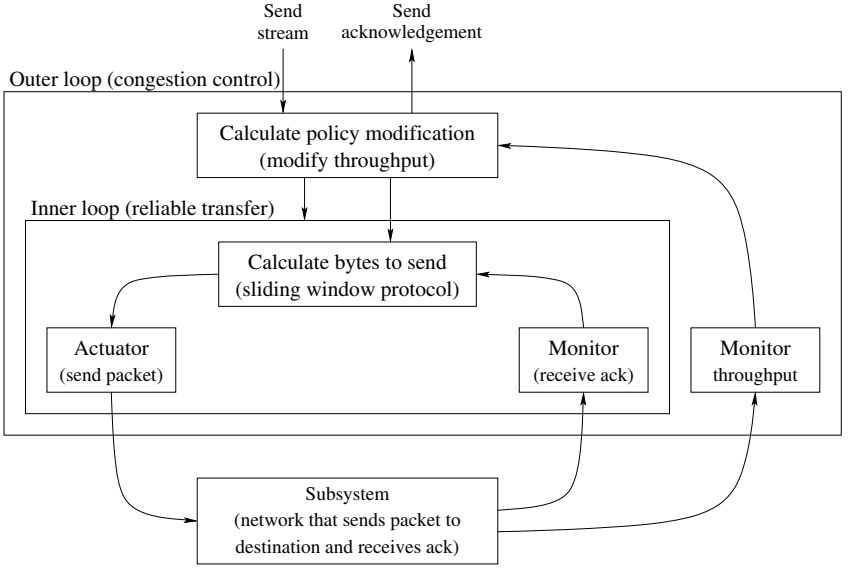

Fig. 6. An example programming pattern with two nested feedback loops

3.3 A new way of designing programs

The style of system design illustrated in the last section can be applied to programming. Programming then consists of building hierarchies of interacting feedback loops. Let us give a simplified example with two nested feedback loops that implements a reliable byte stream transfer protocol with congestion control (this is a variant of the TCP protocol). The protocol sends a byte stream from a source to a destination node. Figure 6 shows the two feedback loops as they appear at the source node. The inner loop does reliable transfer of a stream of packets: it sends packets and monitors the acknowledgements of which packets have arrived successfully. The inner loop manages a sliding window: the actuator sends packets so that the sliding window can advance. The sliding window can be seen as a case of negative feedback using monotonic control. The outer loop does congestion control: it monitors the throughput of the system and acts by either changing the policy of the inner loop or by changing the inner loop itself. If the buffered send stream grows too big or the rate of acknowledgements decreases, then it modifies how the inner loop works, for example by reducing the rate of send acknowledgement or the rate of sending. If the transfer stops then the outer loop may terminate the inner loop and abort the transfer.

This structure is a special case of a multi-agent system. Each block in Figure 6 is a single agent acting concurrently with the others and sending messages asynchronously to the others. Each of the two feedback loops implements one task according to a given policy. The policy of the inner loop is determined by the outer loop. Because the system is distributed over two nodes, part of the design consists in situating each agent on a node.

The example of Figure 6 has just two nested feedback loops. In a real system, there will typically be more nested feedback loops. In particular, the outermost loop determines the main interface between the system and its environment.

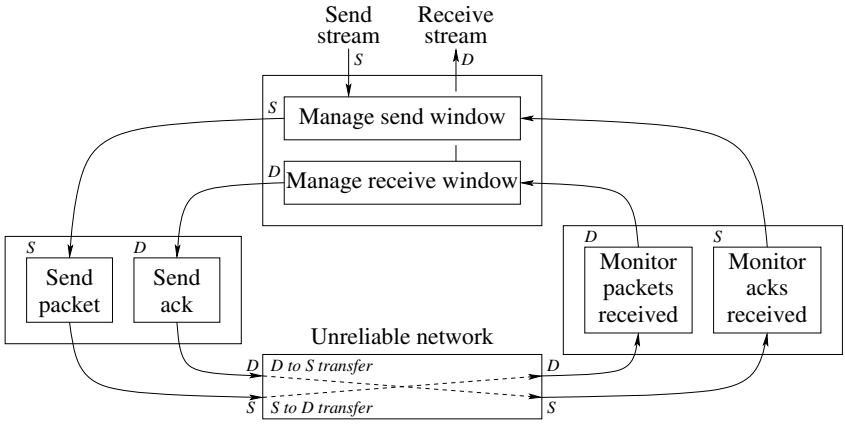

Fig. 7. Inner loop of the reliable byte stream protocol showing distribution

3.4 Interaction between feedback loops and distribution

The protocol of Figure 6 runs on a distributed system consisting of two nodes. Figure 6 only shows what happens at the source node. Figure 7 gives a more complete depiction of the inner loop of Figure 6 that shows the execution on both nodes. In Figure 7, each component is annotated with S or D depending on whether it executes on the source or destination node. This protocol can be seen as two feedback loops (the S loop and the D loop), each executing on one node (S or D), interacting through stigmergy over the unreliable network. If one node fails, then its loop disappears and the other loop sees a change in the behavior of the network. Another way to see the protocol is as a single distributed feedback loop, with parts executing on both source and destination nodes.

An interesting open question raised by this example is how to design distributed feedback loops. This is nontrivial because of the interactions between the design of the loop, its distribution, and the partial failures that it is intended to tolerate. Designing these systems is still mostly an open research question. Structured overlay networks are an interesting special case that is presented below. Other special cases include parts of distributed algorithm theory such as self-stabilizing systems [32]. These systems are able to survive large classes of transient faults.

3.5 Feedback loops in a structured overlay network

We complete our series of examples by outlining how a structured overlay network can be formulated in terms of feedback loops. The most primitive functionality of a structured overlay network is to self-organize a large number of computing nodes to provide reliable and efficient routing despite nodes continuously joining and leaving the network [1,18]. A node can leave in two ways, either by a deliberate action or by failure of the node or its network connections. At all times, routing between non-failed nodes must be correct and efficient.

Figure 8 shows the feedback loop structure of a structured overlay network with n computing nodes numbered from 0 to $n-1$. Node 0 is drawn in detail; the other nodes are shown schematically. The routing organization of the structured overlay network consists of two levels. The first level is a ring in which each node has direct communication links (called *fingers*) to a fixed number f of successors. This ensures correctness (each node can reach all the others by walking the ring) and fault tolerance (failure of $f-1$ nodes does not affect reachability). The second level adds additional links to improve efficiency. The routing algorithm uses a convergence criterium to ensure that eventually the destination node is reached. Each routing hop reduces the distance to the destination until the distance reaches zero. Many well-known structured overlay networks, such as Chord and DKS, are organized in this way.

The communication links provide failure detection. When a node detects the failure of a link then it reorganizes its local finger table. Correct operation of the structured overlay network is therefore based on *two* convergence properties:

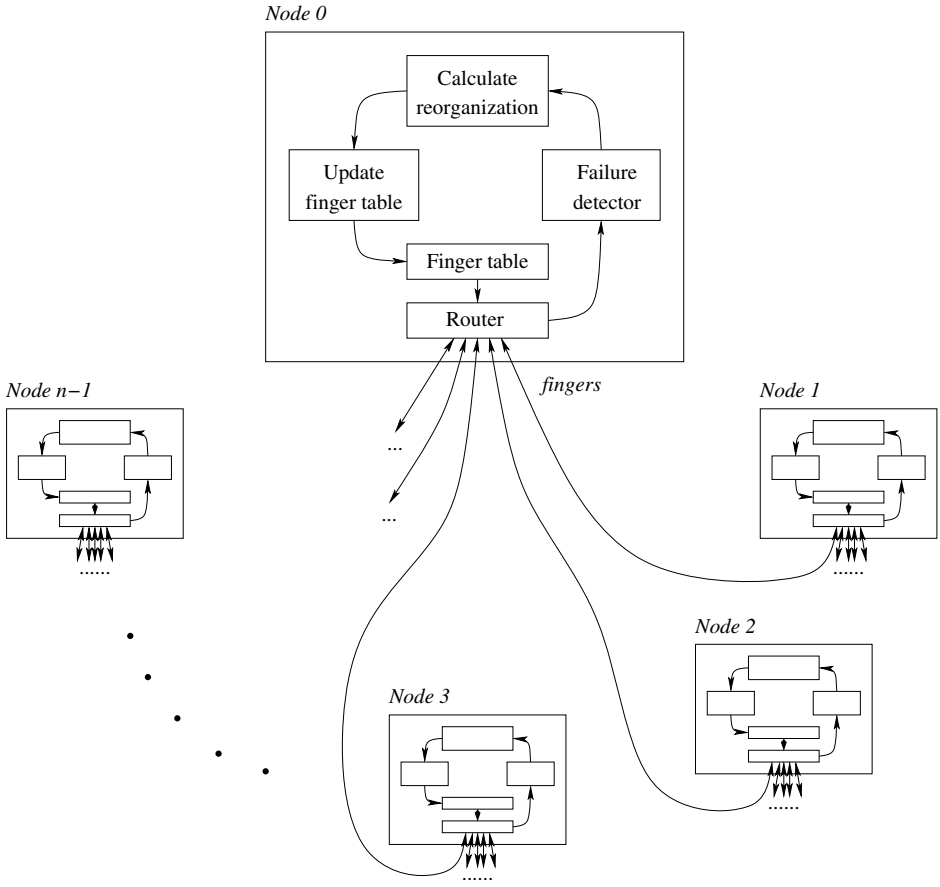

Fig. 8. Feedback loop structure of a structured overlay network

- Within each node, the finger table converges to a correct content.
- Among nodes, a message in transit converges to its destination node.

From the viewpoint of each node, the subsystem being managed consists of the set of nodes it is linked to. When a node leaves or fails, it is eventually dropped from each set containing it. When a new node joins, it is given an initial set that depends on its position in the ring. Since these operations are common, this means that the feedback structure is undergoing frequent changes. Ghodsi [18] gives algorithms and an implementation of a structured overlay network, DKS, that has the above structure and proves that it does correct routing assuming that the failure detectors are strongly complete [17], i.e., every node crash will eventually be detected permanently. The structure modifications done by DKS are designed to be atomic and preserve the topology of the overlay network.

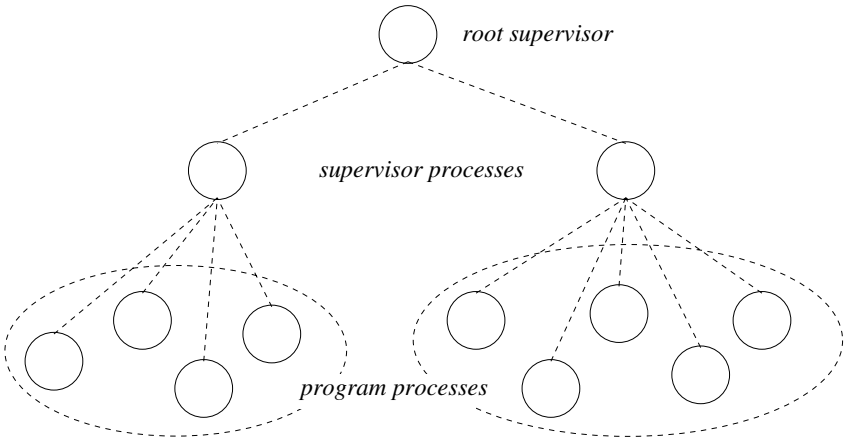

Fig. 9. Supervisor tree architecture of an Erlang program

4 Related work

Several areas of computer science already use a feedback loop architecture. This section gives two examples, namely the Erlang fault-tolerance architecture and the subsumption architecture for implementing intelligent behavior, and discusses them as instances of a feedback loop architecture.

4.1 The Erlang system

The Erlang system is designed to build distributed systems that survive software and hardware faults [3]. It has been successfully used to build systems of extremely high dependability, for example the AXD301 ATM switch which has a claimed down time of only 30 milliseconds per year [30]. Erlang is designed according to the hypothesis that software faults cannot be eliminated completely. Instead of trying to eliminate them, Erlang allows programs to survive them. An Erlang program is organized as a set of concurrent agents (called *processes* in Erlang terminology) that communicate through asynchronous message passing. When a problem occurs in a process, the Erlang philosophy is to let the process fail and to let another process handle recovery. Erlang uses a concept called *supervisor tree* to manage this. The program agents form the leaves of the supervisor tree (see Figure 9). Each internal node in the supervisor tree corresponds to a feedback loop in our architecture. The first internal level in the tree consists of supervisor agents that observe pools of agents in the program's execution. If a program agent fails, then a supervisor agent will restart it in a consistent state, using a database to get the consistent state. There are two kinds of supervisors, AND supervisors that restart all processes in a pool if one fails and OR supervisors that restart just the failed processes. The second internal level in the supervisor tree consists of a root agent that handles failures of the supervisor agents. This root agent must be completely reliable. This is possible because it is a very small program.

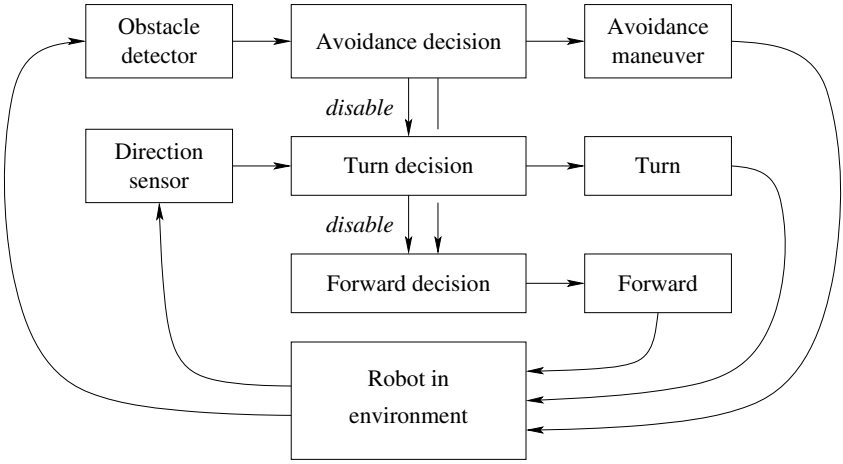

Fig. 10. Feedback loop structure of an obstacle-avoiding robot in the subsumption architecture

4.2 The subsumption architecture

The subsumption architecture of Rodney Brooks is a way to implement intelligent systems by decomposing complex behaviors into layers of simple behaviors that interact through their environment [7,8]. Knowledge is not represented directly inside the system, but indirectly through the state of the system in its environment. The subsumption architecture has been used to successfully implement systems that interact with their environment in a life-like fashion. For example, an obstacle-avoiding robot can be designed with three layers: a move forward layer, a turn layer, and an obstacle-avoiding layer. Each layer is a feedback loop that observes the world continuously. The layers are given priorities. If a layer can react, then it disables the lower layers and performs its own actions. In the terminology of Brooks, it suppresses inputs to the lower layers and inhibits outputs from the lower layers. The default behavior is to move forward. If the direction is wrong, then the turn layer disables the move forward layer to turn. If there is an obstacle, then the obstacle-avoiding layer disables the other two layers and performs an obstacle avoidance maneuver. Figure 10 shows this obstacle-avoiding robot as a feedback architecture. This is a simple example that shows the basic principle. There exist more refined versions of the architecture.

In the subsumption architecture, the feedback loops interact through stigmergy. E.g., in a robot, all the loops detect the robot's position and control the robot's movements. In the feedback loop architecture, feedback loops can also have a policy/mechanism relationship, where each loop modifies the policy that is implemented by the next innermost loop.

5 General architectural framework

Let us now take a step back from the above examples and summarize what a general architectural framework can look like for building a self-managing system. The system is organized as a set of concurrent components that communicate through

asynchronous events. The default behavior is that the components are independent. Any synchronous or dependent behavior must be programmed explicitly. This default gives good results in many cases: for fault-tolerant systems such as Erlang [3], for network-transparent distributed programming systems such as Mozart [13], and for secure distributed programming systems such as E [22]. It also matches well with the complex systems approach taken in physics [14] and used, e.g., in approaches such as belief propagation for solving inference problems [33].

Following the examples of Sections 3.2–3.4 and Section 4, the system consists of a hierarchy of interacting feedback loops, where each feedback loop is implemented by several agents and each agent is an instance of a component. Feedback loops interact either through stigmergy or through management.

5.1 Higher-order component model

In a self-managing system, the system is able to monitor and reconfigure itself, that is, install and update parts of itself while it is running. If the system is built as a set of interacting components then it is possible for components to install other components. Components are therefore first-class entities that can be passed as arguments to other components. This is called *higher-order* component programming. The Fractal component model is an example of such a component model [9]. This model is already being used as a framework for building self-managing systems [6]. In a higher-order component model, it takes some care to determine what component is to blame when a subsystem fails. This has been studied by Findler and Blume [16].

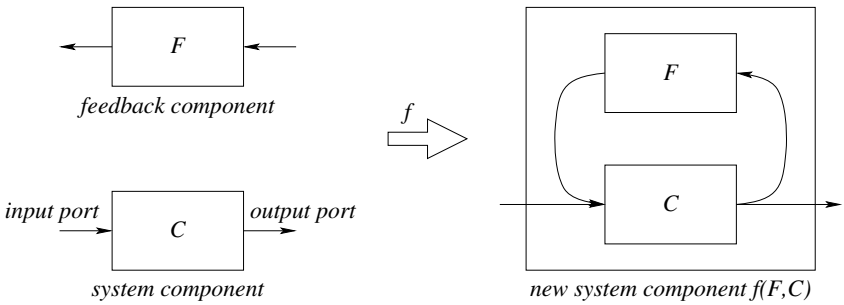

Fig. 11. A component combinator for programming with feedback loops

5.2 Programming with feedback loops

With the right abstractions, a programming language can make programming with feedback loops simple. Each component is a concurrent entity with one input port that accepts a stream of input events and one output port that returns a stream of output events. Components ignore irrelevant events. Both control and content events pass through the same ports. These properties make it easy to compose components in a modular way. This programming model is similar to the model used

by Guerraoui and Rodrigues for defining distributed algorithms in a compositional way [17].

Figure 11 shows a component combinator f that takes two components F and C and returns a component $f(F, C)$ that combines F and C in a feedback arrangement. The combinator f satisfies properties such as $f(F_1, f(F_2, C)) = f(F_2, f(F_1, C))$. We can define an operator $||$ such that $f(F_1, f(F_2, C)) = f(F_1||F_2, C)$. This operator is a form of parallel composition that connects the input and output streams of F_1 and F_2. There are variations of f depending on whether C is explicit (part of the program) or implicit (part of an environment) and depending on whether the feedback loop is managed or not. The semantics of the combinator f needs to take into account two effects:

- The interleaving of the input and output streams. That is, C's input is the merge of $f(F, C)$'s input and F's output and $f(F, C)$'s output is also the input to F.
- Both C and F have a propagation delay, i.e., an output event does not appear instantaneously when an input event is given.

5.3 Global properties

An important part of any general system theory concerns the global properties of a system. Can they be determined for an existing system and can we design systems with desired global properties? The latter question is especially important for large-scale computer systems, such as the Internet or distributed systems built on top of the Internet. Some of the important points are the system's stability, its behavior when stressed, and whether the system's imminent collapse can be detected before it happens. Answers to some of these questions exist for complex systems in physics. Such systems consist of large numbers of very simple components, but they can sometimes be a useful approximation to computer systems. For example, Krishnamurthy *et al* [20] have done an analytic study of the Chord structured overlay network using a master equation approach. Another example is the belief propagation algorithm. This algorithm is defined in terms of message passing between large numbers of simple nodes [33]. It has been used to give solutions to the SAT problem and other problems. Belief propagation is a general technique that can determine global properties of a system in terms of local properties. It can be used for monitoring global properties as part of a feedback loop.

6 Conclusions

This paper motivates that a good approach for building large-scale distributed systems is to consider them as general self-managing systems. We propose to build self-managing software systems as sets of concurrent agents interacting through asynchronous events and implemented using a component model with first-class components and component instances. In this framework, self-managing systems are built as hierarchies of interacting feedback loops. The first design rule is that the whole system (except perhaps a small kernel) should be inside a feedback loop.

Feedback loops interact through two mechanisms, stigmergy (shared environment parameters) or management (one loop controls another). The global behavior of the system depends on all feedback loops taken together and should be predictable from the loop structure. We relate this proposal to two other architectures, namely the Erlang fault-tolerance architecture and the subsumption architecture for implementing intelligent behavior.

These ideas are being realized in SELFMAN, a project in the European 6th Framework Programme that started in June 2006 [27]. We intend to elaborate these ideas into a programming methodology together with an implementation. It should be as easy to program with and reason about a feedback loop as it is for an object or a component. We will design and formalize a component model that is based on the Oz kernel language extended with elements from the Fractal model. We will use this component model as the basis of a programming model along the lines of Section 5 and implement this model in Mozart [26,9,13,23]. We will build a feedback loop architecture on top of this implementation and use it as the basis for a self-managing replicated transactional storage service.

References

[1] Aberer, K., L. Onana Alima, A. Ghodsi, S. Girdzijauskas, M. Hauswirth, and S. Haridi, *The essence of P2P: A reference architecture for overlay networks*, 5th International Conference on Peer-to-Peer Computing (P2P 05), IEEE Computer Society, 2005.

[2] Andrzejak, Artur, Alexander Reinefeld, Florian Schintke, and Thorsten Schütt, *On Adaptability in Grid Systems*, Future Generation Grids, Springer LNCS.

[3] Armstrong, Joe, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, Royal Institute of Technology (KTH), Kista, Sweden, November 2003.

[4] Ashby, W. Ross, "An Introduction to Cybernetics," Chapman & Hall Ltd., London, 1956. Internet (1999): http://pcp.vub.ac.be/books/IntroCyb.pdf.

[5] von Bertalanffy, Ludwig, "General System Theory: Foundations, Development, Applications," George Braziller, 1969.

[6] Bouchenak, S., F. Boyer, D. Hagimont, S. Krakowiak, N. de Palma, V. Quéma, and J.-B. Stefani, *Architecture-Based Autonomous Repair Management: Application to J2EE Clusters*, 2nd International Conference on Autonomic Computing (ICAC'05), 2005, pp. 369–370.

[7] Brooks, Rodney A., *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, RA-2, April 1986, pp. 14–23.

[8] Brooks, Rodney A., *Intelligence without representation*, Artificial Intelligence 47, 1991, pp. 139–159.

[9] Bruneton E., V. Quéma, T. Coupaye, M. Leclercq, and J.-B. Stefani, *An Open Component Model and its Support in Java*, Proceedings 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Springer LNCS 3054, 2004.

[10] Carroll, Lewis, "Through the Looking-Glass and What Alice Found There," 1872 (Dover Publications reprint 1999).

[11] Ceruzzi, Paul E., "Beyond the Limits: Flight Enters the Computer Age," MIT Press, Cambridge, MA, 1989.

[12] Chun, B., D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, *PlanetLab: An Overlay Testbed for Broad-Coverage Services*, ACM SIGCOMM Comp. Comm. Review, 33(3), 2003.

[13] Collet, Raphaël, and Peter Van Roy, *Failure Handling in a Network-Transparent Distributed Programming Language*, in Recent Advances in Exception Handling Techniques, C. Dony *et al* (Eds.), Springer LNCS 4119, 2006.

[14] *EVERGROW: Ever-growing global scale-free networks, their provisioning, repair and unique functions*, Integrated Project, European 6th Framework Programme, 2004-7. Internet: http://www.evergrow.org.

[15] Fairley, Peter, *The Unruly Power Grid*, IEEE Spectrum Online, Oct. 2005.

[16] Findler, Robert Bruce, and Matthias Blume, *Contracts as Pairs of Projections*, FLOPS 2006, April 24-26, 2006.

[17] Guerraoui, Rachid, and Luis Rodrigues, "Introduction to Reliable Distributed Programming," Springer-Verlag Berlin, 2006.

[18] Ghodsi, Ali, "Algorithms for Large Scale Self Managing Overlay Networks," Ph.D. dissertation, Royal Institute of Technology (KTH), Kista, Sweden, 2006.

[19] IBM, *Autonomic computing: IBM's perspective on the state of information technology*, 2001. Internet: http://researchweb.watson.ibm.com/autonomic/.

[20] Krishnamurthy, S., S. El-Ansary, E. Aurell, and S. Haridi, *A statistical theory of Chord under churn*, The 4th International Workshop on Peer-to-Peer Systems (IPTPS'05), 2005.

[21] Lynch, Nancy, "Distributed Algorithms," Morgan Kaufmann, San Francisco, CA, 1996.

[22] Miller, Mark, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," Ph.D. dissertation, Johns Hopkins University, Baltimore, Maryland, May 2006.

[23] Mozart Programming System, version 1.3.2, June 2006. Internet: http://www.mozart-oz.org.

[24] Principia Cybernetica Web. Entry "system," August 2006. Internet: http://pespmc1.vub.ac.be/ASC/SYSTEM.html.

[25] SETI@home, August 2006. Internet: http://setiathome.berkeley.edu/.

[26] Van Roy, Peter, and Seif Haridi, "Concepts, Techniques, and Models of Computer Programming," MIT Press, Cambridge, MA, 2004.

[27] Van Roy, Peter, Ali Ghodsi, Seif Haridi, Jean-Bernard Stefani, Thierry Coupaye, Alexander Reinefeld, Ehrhard Winter, and Roland Yap, *Self Management of Large-Scale Distributed Systems by Combining Peer-to-Peer Networks and Components*, CoreGRID Technical Report TR-0018, Dec. 14, 2005.

[28] Weinberg, Gerald M., "An Introduction to General Systems Thinking: Silver Anniversary Edition," Dorset House, 2001 (original edition 1975).

[29] Wiener, Norbert, "Cybernetics, or Control and Communication in the Animal and the Machine," MIT Press, Cambridge, MA, 1948.

[30] Wiger, Ulf, *Four-fold increqse in productivity and quality* industrial-strength functional programming in telecom-class products, Proceedings of the 2001 Workshop on Formal Design of Safety Critical Embedded Systems, 2001.

[31] Wikipedia, the free encyclopedia. Entry "drowning," August 2006. Internet: http://en.wikipedia.org/wiki/Drowning.

[32] Wikipedia, the free encyclopedia. Entry "self-stabilization," August 2006. Internet: http://en.wikipedia.org/wiki/Self-stabilization.

[33] Yedidia, J.S., W.T. Freeman, and Y. Weiss, *Understanding Belief Propagation and Its Generalizations*, Exploring Artificial Intelligence in the New Millennium, Chap. 8, Jan. 2003. Also MERL Technical Report TR-2001-22, Jan. 2002.