RZ 3487 (# 99302) 01/06/03
Computer Science 10 pages

Research Report

On Autonomic Computing Architectures

Jana Koehler, Chris Giblin, Dieter Gantenbein and Rainer Hauser

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{koe,cgi,dga,rfh}@zurich.ibm.com

On Autonomic Computing Architectures

Jana Koehler Chris Giblin Dieter Gantenbein Rainer Hauser
IBM Zurich Research Laboratory
CH-8803 Rueschlikon, Switzerland
email: koe | cgi | dga | rfh @zurich.ibm.com

Abstract

We discuss the key features of autonomic computing and their relationship to AI systems. We present a generic architecture for autonomic computing systems and propose a computational model based on communicating automata networks to implement such architectures. We illustrate this approach with an intelligent device discovery tool that analyzes the inventory and topology of large computer networks.

Technical Report RZ 3487

January, 6th 2003

1 Introduction

Over the last few decades, computers have revolutionized and automated many of our work processes, allowing humans to address ever more challenging tasks and leaving routine tasks to machines. But an unavoidable byproduct of evolution via automation is complexity, as demonstrated especially by computing systems. In a recent document [8], IBM identified the complexity of current computing systems as a major obstacle to the growth of IT technologies:

> *" . . . incredible progress in almost every aspect of computing— microprocessor power up by a factor of 10,000, storage capacity by a factor of 45,000, communication speeds by a factor of 1,000,000— but at a price. Along with that growth has come increasingly sophisticated architectures governed by software whose complexity now routinely demands tens of millions of lines of code. . . . Even if we could somehow come up with enough skilled people, the complexity is growing beyond human ability to manage it. As computing evolves, the overlapping connections, dependencies, and interacting applications call for administrative decision-making and responses faster than any human can deliver. Pinpointing root causes of failures becomes more difficult, while finding ways of increasing system efficiency generates problems with more variables than any human can hope to solve."* [8]

The answer to this problem lies in more intelligent systems and computing infrastructures called *autonomic computing* (AC) that facilitate and automate many system management tasks currently done by humans.

2 Vision of Autonomic Computing

Autonomic computing can be seen as a holistic vision that enables a computing system to "deliver much more automation than the sum of its individually self-managed parts". A system is considered a collection of computing resources working together to perform a specific set of functions.

> *"It's time to design and build computing systems capable of running themselves, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads we put upon them. These autonomic systems must anticipate needs and allow users to concentrate on what they want to accomplish rather than figuring how to rig the computing systems to get them there. . . . It is the self-governing operation of the entire system, and not just parts of it, that delivers the ultimate benefit."* [8]

Eight key features characterize any AC system, cf. [8]:

1. An AC system possesses *system identity*, i.e., it has knowledge of its components, current status, functions, and interactions with the environment.
2. An AC system has the ability of *self-configuration and reconfiguration*, i.e., it can automatically perform dynamic adjustments to itself in varying and unpredictable environments.
3. An AC system performs *constant self-optimization*, i.e., it monitors its constituent parts and adapts its behavior to achieve predetermined system goals.
4. An AC system is *self-healing*, i.e., it is able to discover the causes of failures and then finds alternative ways of using resources or reconfiguring the system to keep functioning smoothly.
5. An AC system is capable of *self-protection*, i.e., it detects, identifies, and protects itself against various types of attacks to maintain overall system security and integrity.

6. An AC system uses *self-adaptation* to find ways to best interact with neighboring systems, i.e., it can describe itself to other systems and discover those systems in the environment.

7. An AC system is a *non-proprietary open solution based on standards* that provide the basis for interoperability across system boundaries.

8. An AC system has *hidden complexity*, i.e., it automates IT infrastructure tasks and relieves users of administrative tasks.

In the following we introduce a generic architecture that provides a very general framework for the development of AC systems.

3 Generic AC Architecture

The design of technical systems usually focuses on the intended functionality of the system and often obeys the "design follows function" principle. Consequently, a system is organized into components that implement the application-specific functions. Such a system is then embedded into some runtime environment that deals with execution failures and captures exceptions. We believe that such a design cannot satisfy the requirements of AC systems and therefore introduce a generic architecture that introduces system components not at the level of application-specific functionalities, but at the level of functionalities derived from the key features of AC systems, see Figure 1.

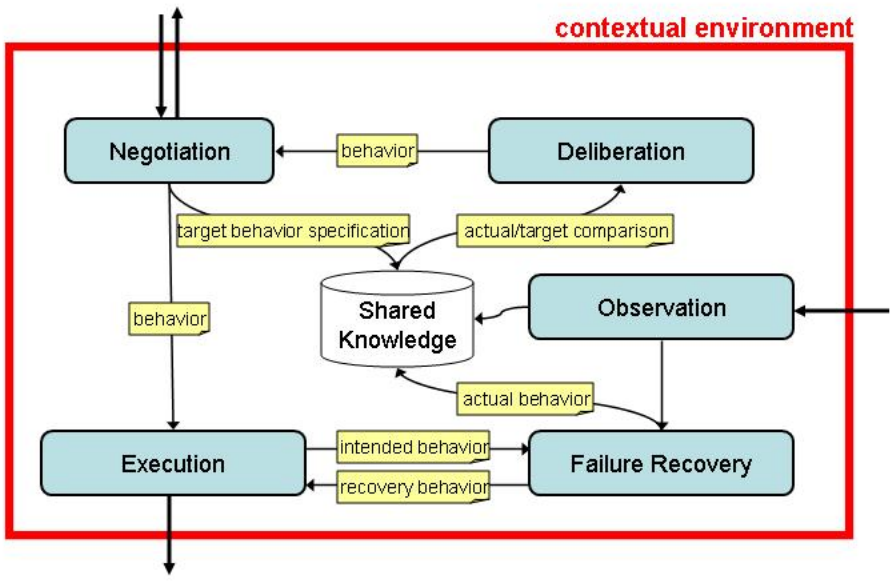

Figure 1. A generic AC architecture.

Each AC system is situated in some environment or context. The interaction between the system and its environment occurs through three system components: *negotiation*, *execution*, and *observation.*

The *negotiation* component has a *two-way* interaction with the environment that allows the system to receive requirements from the environment, negotiate the fulfillment of the requested requirements, make itself known to other systems, or communicate its own requirements to other AC systems it is aware of. The main purpose of this component is to receive and actively construct a *target behavior specification* based on its interaction with the environment. This target behavior specification is added to the shared knowledge of the system components.

Our architecture highly abstracts from the knowledge contents and format, and the sharing mechanisms between the various AC system components. We only assert that the knowledge base contains a representation of the actual system behavior, the system itself and the environment as perceived by the system. When a new target behavior is added to the shared knowledge, which differs from the actual behavior, a *deliberation* process is triggered that will produce a new behavior. We explain this *deliberation* process in more detail below. The *deliberation* process sends the new behavior to the *negotiation* component that decides whether this behavior should be executed. The decision can for example be based on whether new requirements have been received that make the behavior already obsolete.

The *execution* component has a *one-way output* interaction with the environment to execute any behavior that was determined by the *deliberation* component and forwarded by the *negotiation* component. The *execution* component concentrates solely on executing the behavior in a specific environment, e.g., on expanding high-level action descriptions in sequences of lower-level system commands.

The *observation* component has a *one-way input* interaction to receive status information from the environment. The component observes the effect of what the *execution* component is executing without knowing what was actually executed. It adds its observations to the shared knowledge and produces a representation of its observations for analysis by the *failure recovery* process.

Limiting the interaction between the AC system and its environment helps to address the key factors of *self-protection* and *hidden complexity*. A system with a controlled interaction is less vulnerable to attacks and hides its internal complexity by exposing only clearly defined interfaces to its environment. The types of interaction we introduced (one-way, two-way) emphasize the predominant, not necessarily the only flow of information.

Two components that do not interact directly with the environment occur in this architecture: *deliberation* and *failure recovery*. As discussed briefly above, the *deliberation* component computes new behaviors for the AC system and encapsulates the “normal” application-specific functional components. It is responsible for fulfilling the key factors of *self-adaptivity* and *self-optimization.* Two major fields of AI will play a dominant role in the development of *deliberation* components: machine learning and AI planning.

The *failure recovery* component adds *self-healing* and *self-protection* capability to the AC system. Interestingly, it does not interact directly with the environment, but interacts with the *execution* and *observation* components only. The reason for this design principle lies again in the need to reduce the complexity of the system and enhance its robustness at the same time. The *failure recovery* receives information about the intended behavior of the system from the *execution*, i.e., the *execution* component tells it, for example, what action or command it intends to execute next. This information is used by the *failure recovery* to build an internal expectation of what will happen next in the system environment. The *observation* component tells the *failure recovery* what it actually observed happening in the environment. As *execution* and *observation* are completely decoupled in this architecture, they cannot inadvertently influence each other. The *failure recovery* analyzes the deviations between the intended and the independently observed changes occurring in the environment. For minimal deviations (that need to be precisely defined when implementing this architecture), it computes simple *recovery behaviors* that it sends to the *execution* component for immediate recovery. In the second part of this paper, we will sketch concrete examples of recovery behaviors. If greater deviations occur, it updates the shared knowledge with a new actual behavior. This will trigger an actual/target comparison and a new *deliberation* process that may lead to the replacement of the behavior in the *execution* component.

A particular AC system will be based on a sophisticated implementation of the generic architecture. In particular, the sharing of knowledge or information between the various components will usually distinguish between *globally* shared knowledge between all components and *locally* shared knowledge between only selected components. Furthermore, we can expect to see more than one instance of each component or complex components that are AC systems themselves. In particular, the *deliberation* component will probably involve a hierarchical decomposition into application-specific functional components, which is already common in realistic application systems. *Self-configuring* AC systems can be expected to involve several deliberation components—specialized in computing system behaviors or computing new system configurations. We regard this architecture more in the sense of a general design principle that will always require refinements and even modifications when instantiated for a particular IT application.

4 Computational Model for AC Architectures

When implementing a specific AC system, we need to provide computational models to wrap each of the components (or the implementation thereof) and to model the interfaces and the interaction among the components and between the system and its environment. Our main concern is the reliability and robustness of an AC system, because we need to provide guarantees concerning the behaviors generated by the system. Consequently, simple and precise, yet powerful computational models would be ideal.

Agent-based systems provide a very appealing solution. Each agent would (1) encapsulate a specific reasoning method that implements one of the system components and (2) provide the communication and interaction mechanisms with the other agents. Although powerful approaches exist, the *adaptive agent architecture* [10] or the *open agent architecture* [11], there exist no methods that would allow us to formally *verify* that such a complex agent-based system indeed implements the intended behavior. Furthermore, the unique qualities of agents—their autonomy, mobility, and adaptivity—can imply serious legal consequences [6]. Similar considerations also apply to cognitive and behavioral architectures [14, 2], which have a much broader coverage than our generic architecture that we especially target at AC systems in technical environments.

An AC system will often be a distributed system comprising communicating processes encapsulated as the system components. Such a complex system can only be controllable if we consider components with limited capabilities and a finite space of internal states. We therefore propose the use of *communicating finite state machines* (CFSM) [1] as a computational model for AC systems. Deterministic CFSM as the simplest automata model together with a set of communication channels, which are shared by the machines and carry messages of a particular type, were originally used to specify and verify protocols between concurrent processes. Later, the basic CFSM model was extended to networks of powerful automata models, for example nondeterministic, pushdown or timed automata. Automata models, which can even instantiate new communication channels or other automata that then run concurrently with the already existing automata, have also been proposed together with powerful verification tools to analyze the behavior of the automata networks [7]. This computational model has been shown to subsume general Petri nets [13] and to implement process algebraic approaches [15, 9].[1]

By using communicating automata networks we can break a large system into smaller subsystems that compute autonomously and interact with each other via well-defined interfaces only. Interestingly, this vision has been pioneered by the AI community in the seminal paper [12]. McCarthy and Hayes proposed the use of *interacting automata* to model intelligent systems. In contrast to our approach based on *communicating automata networks*, they assumed deterministic automata with explicit state representations and fixed interconnections between the automata based on state transitions. This model suffered especially from the state explosion problem and also exhibited limitations to model complex behaviors due to the complete determinism.

Our networks of communicating automata with explicit communication channels avoid these limitations. First, we use a symbolic representation instead of representing states explicitly. This yields a compact representation that specifies complex behavior and that is well suited for automatic analysis, validation and formal verification. If the capacity of channels is finite, model checking can be used to verify properties of the automata network [3]. Although the complexity cannot be eliminated, it is significantly reduced. Second, instead of hard-wiring the automata, our network is loosely coupled via communication channels carrying messages of a particular type. This approach achieves greater flexibility in the interaction among the system components, which is no longer

[1]The computational model of communicating automata networks also complements recent standardization efforts. As mentioned above, a key feature of AC systems is that they should be *open solutions based on standards*. Web services [4] provide a first standardization of interfaces for the synchronous or asynchronous message-based communication between concurrent processes.

specified explicitly, but results from the exchange of typed messages over the communication channels and the (possibly nondeterministic) reaction of the system components to these messages. In the following section, we will demonstrate our approach on a system for the automatic discovery of devices in computer networks.

5 AC Architecture for an Intelligent Device Discovery System

The analysis and surveillance of large computer networks is a current challenge in the IT industry. For example, when IT infrastructure is outsourced, it is important for the service provider to obtain as much information as possible about the different devices in a network, e.g., mobile and fixed work stations, application and network servers, and the topology of the network itself. Very often, customers cannot provide accurate enough information about their computer assets, so an intelligent device discovery (IDD) tool that can fill this information gap would be very helpful. We have developed such an IDD tool [5] as an autonomic system: it collects information from the unknown network environment without disturbing the network operation and it adapts itself to unknown situations it encounters. The tool encapsulates several network scanning utilities, for example a simple *ping* command that tests whether a machine is alive or the *nmap* scanning utility that allows remote OS identification by TCP/IP fingerprinting the remote stack.

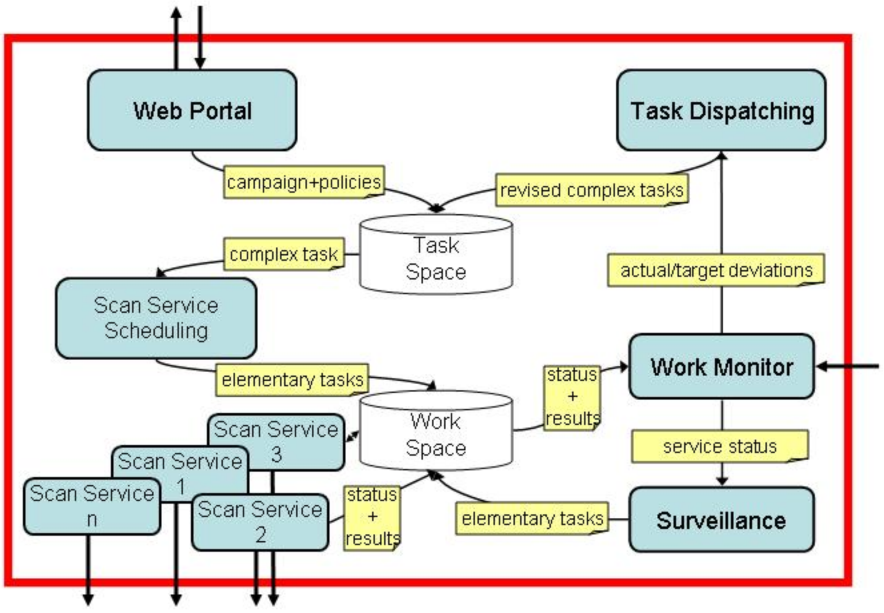

Figure 2. AC architectural view of the IDD tool.

Figure 2 shows an AC architectural view of the IDD tool. Each of the system components is encapsulated as a communicating automaton specifying the behavior of that component. The communication between the automata proceeds via a set of predefined channels. The *task space* and the *work space* are the major global communication channels among the components. The *task space* contains the complex scan tasks, whereas the *work space* contains lower-level tasks that are directly executable by the scan services and also aggregates information returned from the services. Local communication channels exist in particular between the *work monitor* and the *surveillance* and *task dispatching* components.

A user or a system interacts with the IDD tool via the *web portal* that implements a simple variant of the *negotiation* component from the generic architecture. The web portal allows users to define so-called *campaign* and *policies* or invoke predefined campaigns, and to access the reporting documents generated by the tool.

The *campaign* and *policies* describe top-level discovery tasks, e.g., *“investigate the network comprising the IP address range 10.4.16-17.*”* and constraints, for example *“do not scan the subset 10.4.16.* before 5 PM”*. The web portal translates them into complex scanning tasks that it adds to the *task space*, which represents one of two the major communication channels via which the IDD components interact with each other.

Scan services implement the execution component from the generic architecture. A service is a concurrent process executing a specific scan task and working in parallel with other services. A service can wrap a simple scanning utility or it can encapsulate a complex AC system that exhibits a similar architecture as the IDD tool. *Scan service scheduling* is a sophisticated execution component that listens to the task space and derives lower-level tasks, e.g., *“scan 10.4.16.123”* or *“walk the domain name tree (DNS) of this network”*. It can also take a complex task, compute an appropriate parallelization of that task into a set of subtasks and then schedule a periodic scan of these parallel tasks, which will then be executed by simpler scan services. *Scan service scheduling* communicates the generated lower-level tasks to the *work space* to which the simpler scan services are listening. Each service waits for task messages to arrive that fit its input requirements. The *work space* is the second major communication channel.

Scan services communicate their status and results directly back to the *work space* to which the *work monitor* is listening. The *work monitor* implements the *observation* component from the generic architecture. The *work monitor* has to listen to the *work space* because it cannot observe the results of the scans by observing only the network. In fact, none of the *scan services* is supposed to have a visible impact on the network.

The *work monitor* collects the scan results from the services and determines whether the scan tasks have been correctly executed. The scan results and any actual/target deviations are communicated to the *task dispatching* component, which implements the *deliberation* component from the generic architecture. The status information from the scan services is communicated to the *surveillance* component, which enables it to watch the progress of the scanning.

The *task dispatching* component encapsulates a sophisticated deliberation process that analyzes and summarizes the scan results returned from the various scan services. It generates the scanning report and also dispatches new complex scanning tasks arising from the results. For example, when a DNS scan identifies a specific server, e.g., a mail server, a new task for a specific mail server scanning service is derived and added to the *task space*. The reporting subcomponent of the *task dispatching* is able to resolve ambiguities and contradictions in the observed data. For example, several services scan in parallel to discover the operating system that runs on a particular machine, but they can report different results, which must be summarized into a most accurate analysis of the network.

The *surveillance* component implements the *failure recovery* process envisioned in the generic architecture. It merges three different types of information:

1. It observes the *work space* to track tasks that have been added for execution.

2. It receives status notifications about the *scan services* from the *work monitor*.

3. It receives progress information from the *work monitor*, which it compares to the status notifications.

Two main failures can occur in the IDD system: First, *scan services* can become disconnected from the IDD system and fail to pick up and execute a task from the *task space*. Second, a *scan service* can start executing, but fail to return a reasonable result. The *surveillance* process takes an action-oriented approach to recover from these failures. When a task is added to the *work space* it builds up an expectation that it should receive status notifications from a matching service. If those notifications are not received, it concludes that the service is malfunctioning and tries to restart it. If a *scan service* fails during execution, i.e., the status information and the progress information communicated from the *result monitor* do not match, the *surveillance* can take action to reconfigure the set of services that is currently active by either adding another task to the work space that will retrigger an existing service or trigger a scan service not yet active or by making a service inactive that is considered to malfunction. More sophisticated reconfigurations require deliberation and therefore take place via the *task dispatching*, which adds new complex tasks to the task space. For example, contradicting scan results can lead to the addition of a different scheduling service for repetitive scans.

The AC architecture for the IDD tool provides a consistent interface for interacting with the tool via the *web portal* component independently of the configuration of services the system is running. The points of control are within the *web portal* and the *task dispatching* that add new tasks to the *task space*, the *scan service scheduling* that maps complex tasks into more fine-grained tasks and adds them to the work space from where they are picked up by the *scan services*, and the *surveillance* component that can modify the operation of the scan services.

The subsequent figures show the simplified state machines of a very simple scheduling service, a scan service and a service-specific surveillance process. The state transitions are annotated with the messages that are sent by the state machine. For example, *taskspace?complexTask* means that a *complexTask* message is read from the *task space*, whereas *workspace!task* means that a message *task* is sent to the *work space*.

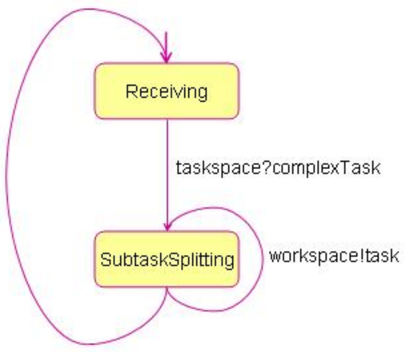

Figure 3. State machine for a service generating a nondeterministic set of subtasks for further processing.

Figure 3 shows a non-deterministic state machine that splits a complex scan task into several subtasks. How many subtasks will have to be generated is not known at design time, but only becomes known at runtime when the splitting service processes the complex task. The number of subtasks needed might for example be represented as a parameter in the specification of the complex task.

Figure 4 shows a part of the CFSM of a scan service that first executes a port scan command and then executes a more specific scan command depending on the result of the port scan. After picking up the task from the *work space*, the service communicates the task-specific status notifications and scan results back to the *work space*.

Figure 5 shows a state machine that is used to supervise the service shown in Figure 4. The cyclic *surveillance* process (not shown) spawns such a service-specific state machine for each task that is added to the work space. These state machines wait for the status notifications communicated from the *work monitor*. The *surveillance* also instantiates a specific timer for each running scan service. If a state notification is not received within a certain period of time, this timer will send a timeout message, which causes the surveilling CFSM to enter a failure state. This will trigger further recovery actions. In our example, it simply adds the same task again to the work space.

Using networks of communicating automata allows us to formally specify the behavior of each of the system components and formally verify important properties of their behavior using an automatic model-checking tool. An example of such a behavioral property of the IDD system is that all components should terminate in their end states only when the task and work spaces have been emptied. Owing to space limitations we cannot discuss the verification issues in further detail nor show the CFSMs in their full complexity.

Currently, the approach is used at the design level to specify and verify the system components. Each CFSM is

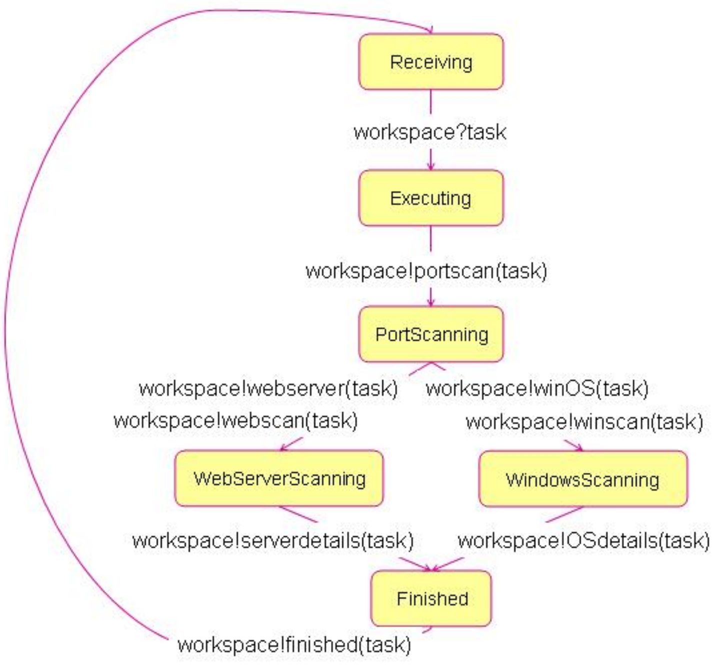

Figure 4. State machine for an example scan service.

then implemented and deployed in a distributed system including the IBM Websphere Platform. The system has been used successfully to analyze large computer networks comprising up to hundreds of thousands of machines. In a next step, we plan to develop an execution engine that drives the entire system and allows us to directly execute system components specified at the CFSM level.

6 Conclusion

We propose a generic architecture for autonomic computing systems based on communicating processes that each encapsulate a specific higher-level functionality such as *deliberation*, *observation*, *negotiation*, *execution*, and *failure recovery*. The computational model we propose is a network of communicating automata to implement the autonomic computing architecture. Such a network has several advantages: First, communicating automata allow the complexity of the system design to be reduced significantly. Symbolic representations yield a compact representation of the state space and in particular allow us to apply formal symbolic model-checking methods to verify the behavior of an autonomic computing system. Second, the interaction between the system components proceeds over locally or globally shared communication channels that carry messages of a particular type. This yields clearly defined interfaces between the system components and the environment, which improve the interoperability of autonomic computing systems.

We demonstrate the approach with an intelligent device discovery system. Future work will address the scalability of the approach to application systems in the area of e-business process integration and automation based on web services. A specific challenge in this area is the existence of concurrent processes that need to be spawned dynamically and the representation of business rules and policies.

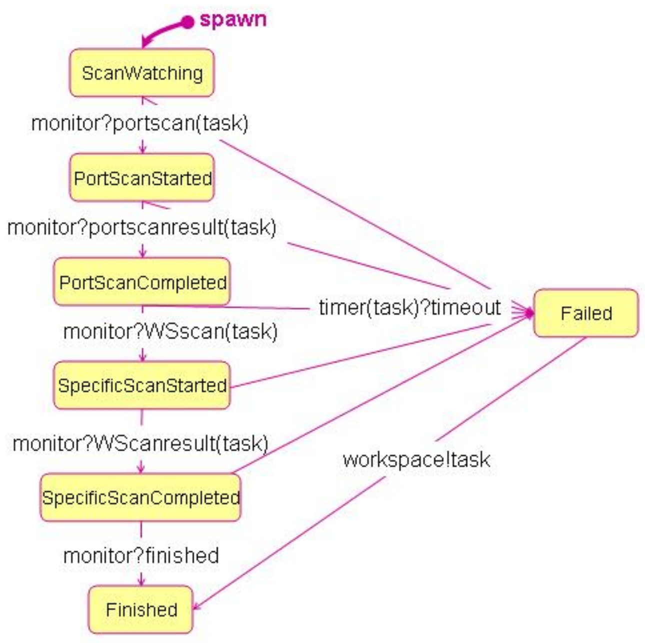

Figure 5. State machine for surveilling a service.

References

[1] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[2] R. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. The web services description language WSDL. http://www-4.ibm.com/ software/ solutions/ webservices/ resources.html, 2001.

[5] D. Gantenbein and L. Deri. Categorizing computing assets according to communication patterns. Tutorial on Asset Inventory and Monitoring in a Networked World at the 2nd IFIP-TC6 Networking Conference, 2002. http://www.zurich.ibm.com/csc/ibi/idd.html.

[6] C. Heckman and J. Wobbrock. Liability for autonomous agent design. *Autonomous Agents and Multi-Agent Systems*, 2(1):87–103, 1999.

[7] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991.

[8] IBM Corporation. Autonomic computing - a manifesto. www.research.ibm.com/autonomic, 2001.

[9] G. Karjoth. Implementing LOTOS specifications by communicating state machines. In *Proceedings of the Third International Conference on Concurrency Theory*, volume 630 of *LNCS*, pages 386–400. Springer, 1992.

[10] S. Kumar, P. Cohen, and H. Levesque. The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams. In *Proceedings of the 4th International Conference on Autonomous Agents*, pages 459–466. ACM Press, 2000.

[11] D. Martin, A. Cheyer, and D. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1):91–128, 1999.

[12] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[13] W. Peng and S. Purushothaman. Analysis of a class of communicating finite-state machines. *Acta Informatica*, 29(6/7):499–522, 1992.

[14] P. Rosenbloom, J. Laird, and A. Newell. *The SOAR papers: Readings on Integrated Intelligence*. MIT Press, 1993.

[15] D. Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, volume 369 of *LNCS*. Springer, 1989.