

robust distributed systems: achieving self-management through inference

Willem de Bruijn

wdb@few.vu.nl

Herbert Bos

Vrije Universiteit Amsterdam

herbertb@cs.vu.nl

Henri Bal

bal@cs.vu.nl

Keywords: distributed systems, resources management, autonomic computing

Abstract

Self-management can reduce administration complexity in distributed systems. We introduce a method based on task automation: formally encode best practices so that they can be reasoned about and adapted at runtime. A key advantage of the chosen approach is its support for existing hard- and software.

We present the methodology and an architecture. The method's applicability is tested by applying a preliminary implementation to a handful of practical problems.

1 introduction

Distributed systems are growing, both in size and in heterogeneity. As changes in the infrastructure can have ever more widespread and unforeseen consequences, stability is at risk. Because of this, managing networked systems is becoming increasingly complicated. Network resilience can be strengthened by decreasing dependence on human intervention. Not only is manual labor costly, it is the main cause of security related issues in enterprise networks [1] and a performance bottleneck, for instance when dealing with worm containment.

We propose to reduce *perceived* administration complexity by automating management tasks. Applicable tasks range from the very simple (e.g., copying files) to the slightly more complex (periodically backing up files) to the very complex (controlling a wide-area grid). The ultimate goal is to make manual management disappear completely, but we take a bottom-up approach: automate tasks one-by-one, from simpler to more complex. Automation is introduced by supplying a decision-making process with blueprints for task-handling, or templates. Templates either encode a single action (e.g., `copy X to Y`), or are

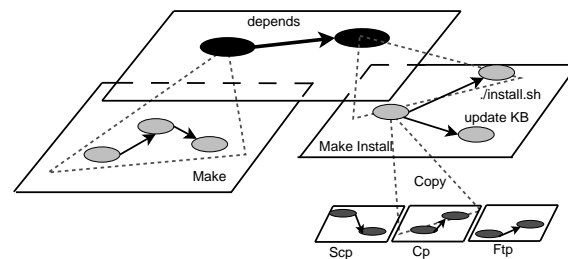


Figure 1: Composite task example: install software

compositions of other, more simple templates. Because there are often multiple ways of solving a problem (e.g., `scp`, `cp` and `ftp` can all be used for file duplication) the solution space contains redundancy. Optimal solutions can then be chosen by taking into account runtime system state. An example composite template is shown in Figure 1.

The precise actions to take depend on application-specific constraints, resource scarcity and interdependence of processes. For this reason a decision-making process must have detailed knowledge of the runtime environment, the templates and the administrative policies. Management tasks are generally well understood and decomposable, therefore they lend themselves well to automation.

We use explicit model-based reasoning for the decision-making step. This approach, we argue, has advantages over other forms self-management. In particular, it scores well on the following desirable properties: understandability, reusability, interoperability and extensibility.

We will first give an in-depth discussion of the methodology. We then continue with presenting the architecture, after which the details of a first prototype are shown. Finally, a handful of example problems are dealt with and conclusions about the method's applicability are drawn.

¹A shorter version of this paper will be presented at the first IEEE Workshop on Autonomous Communications and Computing (ACC2005).

2 methodology

To remove the human from the loop his domain knowledge must be transferred to an automated environment. Traditionally, micro-management tasks have been automated using ad-hoc shell scripts. Scripting can help solve simple problems, but it falls short in the face of increased complexity: because imperative programs encode static, rigid procedures they need manual intervention to adapt to changes in the environment. What we propose can be seen as a next step in scripting: *adapting task handling code automatically*.

Automation is introduced by mimicking human deduction processes. An explicit decision making process (i.e., an expert system) has a library of task-handling templates from which it can create instances on-demand. It uses its model of the environment to select and compose templates to optimally solve given problems. By taking into consideration system state expert systems can carry out complex management tasks more robustly than static scripts. Secondly, their formal knowledge representation can aid code and domain knowledge reuse. This is especially of use in networked environments, where it can benefit for instance QoS negotiation and worm containment.

Adaptation can be achieved through other methods, such as parametric optimization or Bayesian networks. But explicit model-based reasoning has these advantages:

1. *understandability* A line of reasoning, by virtue of its logical representation, can be easily followed and understood. This makes the method potentially less controversial than implicit decision-making techniques.
2. *reusability* Logical statements can be correlated. The ensued network effect causes the sum to be more than its parts. This cannot be said of fragmented, embedded knowledge.
3. *interoperability* Translation between formal languages is trivial, while making ad-hoc representations interoperate is cumbersome.
4. *extensibility* High-level languages encourage modification and extension of knowledge by end-users.

3 architecture

Principal to our view are local experts acting in a shared environment: the network. Design of the system can be split in two: (1) construction and instruction of individual experts and (2) communication between the actors in the network. We pursue a bottom-up, hands-on approach, taking up increasingly com-

plex practical management challenges. The first priority is building a local expert. Collaboration will be addressed in a later phase of our research.

3.1 local experts

The local expert receives runtime statistics from its environment and policies from its controllers. Low-level data is extracted by refactoring existing monitoring tools such as SNMP or Tivoli. Higher-level policies can be entered in a formal syntax, or indirectly through a friendlier (web) interface. Much research has already been undertaken into policy-based network and grid management [2, 3].

We introduce adaptation through model-based reasoning, similar to Muscettola [4] and Garland [5]. In model-based reasoning an expert builds a model of an environment from live status information. Through correlation of low-level data it can deduce more meaningful facts. And to move the system to a desired state it can intervene, creating a closed-loop system or "self-managing habitat".

Tasks are our intervention mechanisms. To be able to create them without human assistance a formalized representation of system tools is defined: templates. Templates can encode simple actions, but also abstract patterns: for instance parameter-sweep, or workflows. They separate low-level tool instruction from higher-level architectural design [6]. More importantly, they stimulate code reuse: tools can be exchanged, e.g., `cp` for `scp`. Patterns are swapped just as easily: a parallelizable task can be sent to a batch scheduler, instantiated using the `parameter_sweep` pattern or processed locally in series.

Figure 1 shows an example task: software must be installed from source. The compilation subtask can be handled on high-performance hardware, from where the installation process on the end-hosts copies the binaries, using one of 3 available tools. Each square represents a template that can be exchanged and reused elsewhere.

Model-based reasoning reduces management to a constraint satisfaction problem, but one that is not easily caught in formal logic. Recurring problems in distributed systems are communication disruption, hardware crashes and security breaches. How can we encode these problems and their solutions? Many expert systems in use today are based on predicate logic parsers. A problem of predicate logic is that all deductions must either be true or false, while in reality there is often too little information to draw a conclusion with certainty. There are ways of dealing with uncertainty [7], but there is no silver bullet. By explicitly modeling incomplete knowledge we avoid most issues related to logical deduction, while retaining the attractive qualities listed in Section 2,

The main challenges in building a local expert are

(1) defining a language for task encoding and environment modeling, (2) building an expert system that reasons in this language and (3) connecting the expert to the environment through sensors and templates.

3.2 global partners

Experts are meant to be deployed in large environments such as grids and the internet, because it is here that cooperation through knowledge exchange is most effective. But scaling to millions of nodes poses additional challenges: where in the network should we position the experts? How many experts do we need? How can they communicate safely and practically? How maintainable is such a distributed system?

Research in this field is active. On the one hand we have systems researchers implementing the vision of a global ‘knowledge plane’ [8]. On the other, we are beginning to see results from the knowledge representation society’s efforts in building global knowledge-exchange languages, notably the semantic web [9]. Building on these are plans for next generation integrated fabrics, among which are knowledge-oriented [10] and semantically driven grids [11] and virtual organizations of autonomous agents [12]. While these are exciting endeavors, our project attacks the problem from another, bottom-up, angle by expanding upwards from the individual expert.

Not only is this a pragmatic approach, we also feel it may be complementary to the aforementioned initiatives as the knowledge that we build up may subsequently serve as core concepts for any of the other approaches.

Regarding expert placement we note the following: centralized, or hierarchical approaches to distributing control are ill-suited to the internet as it inherently lacks centralized control (notwithstanding archaic tools such as DNS). Administrative boundaries are becoming increasingly vague, with groups of actors temporarily pooling their resources (p2p), individuals using many resources at once (ubiquitous computing) and - eventually - systems becoming ever more interwoven (grids).

While knowledge may be globally true, optimization strategies are inherently biased. Also, third-party information may be unreliable, intentionally or not. Therefore experts in a global space are to be *user-centric* and *skeptical*: able to fulfill their goals independently, although possibly through cooperation, with knowledge that is not 100% reliable. Thus, while not focusing directly on competing and cooperating autonomous entities (e.g., ‘agents’), conceptually our work fits nicely within the view of virtual organizations.

4 implementation

We have built a prototype expert that is based on the presented architecture: BetaGIS¹.

Expert systems have been proposed for wide-area automation before in the vision for a semantic web [9]. We differ from that vision by taking a goal-directed approach to inference: our reliance on real-time data makes precomputing correlated facts (‘forward-chaining’) a wasted effort. Because of this BetaGIS could be built on top of SWI-Prolog, a Prolog interpreter that has many of the features we require, such as an in-built HTTP server with Semantic Web extensions.

As our problem space is large and Prolog is a versatile language we must take care not to end up with a chaotic bag of unrelated code. For this reason we are continuously refining a set of base primitives. It is precisely the reusability of these primitives that breeds potential adaptation. The primitives found so far can be divided into two groups: relationships and tasks.

4.1 relationships

Relationships interconnect concepts and their instances. As a practical example, let us look at a C compiler. Useful things to know are what sort of input it requires (`gcc` **accepts** C sourcecode), what sort of output it produces (`gcc` **produces** x86 object code), on what machines it is available, whether the machine is up, etc. Next, when implementing (for instance) a parallelizing version of make an expert can automatically find appropriate nodes on which to compile.

The implemented relationships cover not just the well-known `InstanceOf` and `ChildOf` relationships from description logic, but also less generic relationships, such as those concerned with ordering: `needs`, `wants` and the already introduced `accepts` and `produces`. Other relationships that have proven useful are `implements` for task to tool translation (`gcc` **implements** a compiler) and `available` for retrieving real-time information (`gcc` is **available** at node X).

An example is shown in Figure 2. The snippet of pseudo-code searches for a C to x86 compiler. Note that in Prolog words starting with capitals are free terms (variables), while all others have been bound. In the example we are trying to find such values for `TOOL` and `HOST` that all given relations evaluate to true. The first line searches for a tool that implements a compiler. The next two lines constrain the type of compiler: it must accept files of mime-type `text/x-csrc` and return files having the filename extension `‘.x86.o’`. Then, the last two lines select a suitable

¹up-to-date information can be found at <http://www.few.vu.nl/~wdb/betagis>

```
implements(TOOL, compiler),
accepts(TOOL, IN), mime(IN, text/x-csrc),
produces(TOOL, OUT), ext(OUT, '.x86.o'),
available(TOOL, HOST),
node_status(HOST, up).
```

Figure 2: example query for a C to x86 compiler

location for execution: a reachable host at which the requested tool is available.

The example shows another drawback of predicate logic parsers: they only search for a correct answer, not an optimal one. However, such shortcomings can be circumvented. The example could find an optimum by selecting the node with the most free cycles.

4.2 task-handling framework

The second part of the core concepts deal with active processes. For this work we were able to build upon earlier research into the design of a flexible processing framework (FFPF [13]). Process descriptions (i.e. templates) are fairly simple: they detail what input, output and options processes require or accept. As mentioned before, there are two basic types of templates: those encoding atomic tasks and those encoding composite ones. Atomic templates make it possible for BetaGIS to intervene in the environment with the same tools as *human* experts, such as `grep`, `net-snmp` and `procds` by trivially encoding them: `task(name, input, output, options)`.

Composite templates combine others through operators for sequential and parallel execution (resp. `seq` and `par`), similar to Occam [14].

The following simplified code snippet shows a template for the UNIX `diff` command. It contains the name, the command template and room for input, output and option. The last argument, written on the second line, is a constraint set. In this case, the input must be a list of two elements and the output a single element.

```
template(diff, 'diff $opt $in1 $in2 > $out1',
In, Out, _, ( length(In, 2), atomic(Out) ))
```

At runtime, templates are matched with task requests to find a suitable fully bounded solution. As binding free terms is a two-way process in Prolog, both users and resources can constrain the option space. Let's reinspect the example in Figure 2. The `accepts` and `produces` relationships can be embedded in the following gcc template:

```
template(gcc, 'gcc $opt -o $out $in',
In, Out, '-m386',
(mime(In, text/c-csrc), ext(Out, '.x86.o')))
```

This template will for instance combine with task-request

```
event(node_down, [host, time])
event_handler(email, [Subject, Body, Recipient]),

event_arg(node_down, email, Subject, time),
event_arg(node_down, email, Body,
[event(host), static('is down')]),
event_arg(node_down, email, Recipient, 'wdb@localhost'),
event_attach(node_down, email).
```

Figure 3: Connecting an event to a handler

```
task(compiler, Cmd, 'a.c', 'a.x86.o', _)
into
task(compiler, 'gcc -m386 -o out.x86.o in.c',
'a.c', 'a.x86.o', '-m386')
```

Complementing these adaptable script snippets are predicates for specifying recurring tasks and events, which are used in background tasks, such as monitoring and maintenance.

Conceptually, the event-handling system implements a reliable message-passing interface. Event messages are relatively free-form, consisting of a locally unique name and a logically unbounded list of Prolog terms. Event-handlers are tasks with a few additional restrictions. They may not expect input files, for example. If they fit the mold, tasks can be registered and unregistered from the message queues at runtime. When an event fires its registered listeners are called. Each listener is passed its own set of parameters in accordance with its template. These parameters are either a variable assigned statically during binding, a term from the event message, or a script embedding the two. A combination of compile-time and run-time checks guarantees the safety of the calling mechanism. The code snippet in Figure 4.2 shows an example connection between an event, `node_down`, and a handler, `email`.

The notation's elegance is debatable, but the example shows how the mechanism works. This cumbersome programming does not have to be carried out manually, however, as the web interface exports a point and click front-end to these predicates.

Together, these concepts are powerful enough to encode many day-to-day management tasks. However, we may extend them with other language constructs, such as flow-control (loops, switches) or even support a more formally grounded coordination language like Manifold [15].

Templates are translated into 'static' scripts at runtime. A drawback of this approach is a lack of runtime feedback and consequent loss of control. On the other hand, removing the inference engine from the runtime system makes the process more easily traceable.

The maturity of the task-handling framework is best demonstrated by the fact that the expert system's own tasks are encoded in it. For instance,

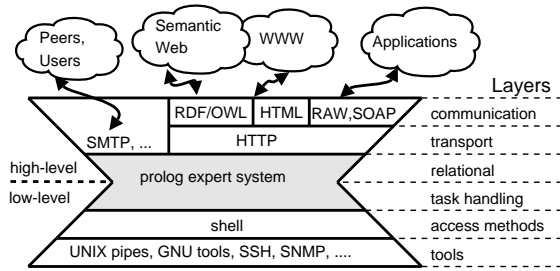


Figure 4: local expert design

a knowledge-acquisition task is used to recursively build up information about the network: foreign devices, their access methods and resources (tools, data and hardware) are queried in a `configure`-like manner. New information, e.g., obtained from the hosts file, can lead to more inspections, theoretically *ad infinitum*, while stale data is overwritten or removed. This process is written down using the task-handling code so that it automatically adjusts to new locations.

4.3 interfaces

While BetaGIS can directly control its environment by executing tasks, this mechanism isn't sufficient for interacting with end-users and peers. For that purpose it will need a more high-level method of communication. The SWI-Prolog shell is not sufficient for a myriad of reasons, among which are concerns about ease of use and automation. Instead, BetaGIS has a built-in HTTP server. As shown in Figure 4, the HTTP server supports three different interfaces.

The simplest interface accepts raw Prolog queries and generates responses wrapped in a minimal envelope. It allows BetaGIS to be asked questions from remote locations. Through command line HTTP requests (e.g., with `wget`) the interface even allows the embedding of BetaGIS actions into scripts. For instance, a host can add itself to the network at boot-time by executing the following shell script.

```
wget ${GISURL}?q=configure\($HOSTNAME\)
```

Entire toolchains can be replaced with their remote cousins in this fashion. For example, a `gcc` call can be intercepted and redirected to BetaGIS without the user knowing that other resources will actually carry out the request using (an extended version of) the following code snippet:

```
#accepts gcc OPT -o OUT IN
wget -O $2 ${GISURL}?q=task(gcc,_,{in1},_,$1)&in1=$3
```

The second interface concerns interaction with end-users. This BetaGIS *portal* can help less tech-savvy users specify tasks or information requests in an easy-to-grasp graphical manner. Difficult queries are

embedded in opaque HTML links that can be bookmarked for later use. Wizard-like interfaces encourage step-by-step knowledge acquisition, for instance for composite task specification. For monitoring purposes, the interface has also been given plot generation support. It visualizes data in the manner of IBM's Tivoli or its open-source cousin Nagios, but as yet less refined.

A currently unused third interface allows communication with Semantic Web applications through RDFS+OWL language parsing. Many of our concepts are not compatible with RDFS or OWL at the moment, but we are contemplating changing this after the dictionary stabilizes.

Future extensions already shown in Figure 4 but not yet implemented concere SMTP for sending alerts and SOAP for integrating our task-handling framework with Web Services.

5 case studies

BetaGIS is a research vehicle. Through its deployment we hope to find out which concepts are useful to management tasks. So far we've experimented with a small but diverse set of problems that we believe indicate the potential of expert systems such as BetaGIS.

5.1 end-user experiences

Personalization is a building block of ubiquitous computing. Our formalized tasks can help achieve this goal. Let's take, for instance, the mundane task of printing. In general a user wants to print a file at the nearest printer, preferably without having to specify the device explicitly (he may not even know its name).

Within BetaGIS, the location of an object can be specified using an extensible set of predicates. At the moment locations on the Vrije Universiteit campus, zipcodes and URLs are understood. These can be correlated, as in `vuloc(FEW, 'R5.23')` is near to `zipcode(nl('1081HV'))`. At runtime, the `near` relationship and its cousin `nearest` can then be used to send the job to the printer nearest to the machine from which it was sent. And to tell the user where to fetch his print-outs, of course.

It is difficult to store and load statements in this manner using scripts. Because relations are hard-wired, extensions to the system, for instance additional buildings, will need editing of the script. Only when the knowledge is externalized can the script (which is then an expert system) remain static.

5.2 distributed compilation

One highly decomposable task that has been the target of parallelization before is the `make` process.

Tools such as DistCC and Prom [16] distribute compilation. But they do not address multi-step compilations (`yacc`) or tool selection (`icc` vs `gcc`). BetaGIS *can* search for the optimal toolchain at runtime. Relationships such as `produces` and `needs` construct a composite task, while `implements` and `available` ensure the task can be handled by the system.

This process generates code. While scripts can do the same, it is not trivial to get all the background information necessary to optimize the task. Centralized background information, in a knowledge base, reduces duplication. And we can then improve the system by adding knowledge in the form of new templates. In scripts we would have to edit the code-generating logic itself.

5.3 monitoring OpenPBS

As a prototype of an advanced management task, we have almost completed the implementation of a fault recognition and recovery mechanism for the Portable Batch Scheduler (PBS), a set of network daemons controlling high-performance clusters. PBS has been found quite brittle. As not just the hardware, but also the OpenPBS software itself can fail at any time, we have built background health sensors, together with event-handlers for known error conditions. Errors and their respective recovery mechanisms were identified through traditional knowledge engineering techniques, such as interviews. Consequently they are direct copies of the administrator's manual actions (e.g., logfile inspection, job cancellation and server rebooting).

For example, the state of each of the compute nodes is periodically checked by a background task. Each check fires off an event signaling the status of that node. Arbitrary tasks can be connected to events. We currently support sending notification emails, restarting `init.d` processes, printing log messages and updating plots. The last is connected not to the status events, but to a background timer to periodically update the Tivoli-like web interface.

BetaGIS's ability to use standard shell tools helped reduce development time and program complexity for this scenario. Its use of recurring tasks, events, and a graphical front-end make it unsuitable for scripting. A professional monitoring framework written in, say, C could do the same. But then it would probably use the same external model as BetaGIS.

5.4 BetaGIS internals

The quintessential test for a piece of control software is whether it can monitor and control itself. BetaGIS uses its task-handling code for many of its internal actions. The `configure` task discussed in

Section 4.2 is one. Another, the web front-end, relies on on-demand graph construction through external tools such as `dot` and GNU `graph` for its visualization methods. Image requests are automatically forwarded to these applications. When multiple locations carry the application an optimization technique is applied, such as load-balancing. Similar mechanisms exist for remotely executing tasks in general (e.g., through `sh`, `ssh`, `snmp`, `http`).

6 related work

Composition of interdependent processes is central to the task-handling framework. Perry and Wolf introduced this architecture [6], which lends itself well to modification. The advantages of *automatically* modifying code have been observed before [17]. Even task-oriented adaptation [18] has been suggested previously. But so far adaptation is limited to optimization of quantified values, which are often hard to define correctly. The same problem also plagues approaches based on Bayesian reasoning [19].

Inference-based approaches, including aforementioned Bayesian methods, have as advantage that they can be applied to a wider range of problems than numerical optimizers. Georgiadis *et al.* have taken an approach very similar to ours [20]. But they chose a decidedly more bottom-up method for constructing the overall architecture. The end-user specifies the architecture in terms of constraints and the software system assembles itself from the available components. Components are simple processing elements connected through reliable pipes. It is questionable, however, how well this elegant design translates to the real world. Another very similar approach uses autonomous agents as principal reasoning objects [21]. Here the agents are also rule-based and capable of interacting, but the solution is geared towards agent environments. Both projects lack our focus on supporting the existing infrastructure and thus also BetaGIS's ability to leverage off-the-shelf tools.

Systems-oriented approaches have so far been mostly limited to parametric optimizations. Delphoi [22] gathers low-level data and relates it to higher-level queries. It is especially interesting because it not only returns simple numerical data, but also a handful of higher level queries that correlate simple statements. These queries are hardwired at compile-time, however, and limited in scope and number. Another practical application, MDS-2 [23], implements a distributed knowledge exchange from off-the-shelf parts. It lacks inference. Furthermore, its hierarchical nature makes it hard to set-up in a global space with untrustworthy partners. Our method is also related to, and partly stems from, the Application Private Networks [24] initiative. The differences should be searched for mostly in

application domain. Apnets targets network protocol optimization, not general resource management.

Virtual Private Grid [25] is a practical solution focusing on leveraging existing tools. Following the UNIX philosophy of doing one thing well, VPG is a shell extension that removes the burden of dealing with network topology and connection setup. Remote job submission is hardly more difficult than local submission. VPG does not take into account higher level strategies, e.g., regarding parallelization or optimal job placement.

Our method to achieving adaptation brushes against more esoteric programming techniques. Called *meta-programming*, *meta-compilation*, *generative programming* or simply code generation, these technologies extend existing templates (e.g., the C++ template system [26]) to generate optimized applications in a just-in-time fashion. More well-known, and more limited, programs of this type are YACC and the C preprocessor. But all these tools and languages are geared towards programmers and have no runtime knowledge-base.

A flexible method of modeling software architectures - and arguably a more refined one than our current task-handling code - is using specialized languages. Coordination languages [15] are designed to coexist with languages expressing computations. They express the interactions between the processing components in a formal manner. In their goal they are similar to Occam [14], but in power more expressive. Therefore they are also more expressive than our task-handling framework - and a candidate for adoption.

7 conclusion

Reducing human intervention can make distributed systems more robust. Automation based on formal encoding of best practices plus knowledge-based adaptation can help close the loop. A key advantage of the chosen approach is its support for the existing hard- and software infrastructure.

Validating our method, a prototype expert system was shown to be able to handle example problems. We plan to extend our work by (1) incorporating resource bounds and access restrictions into the framework and (2) scaling the solution to wide-area networks, as discussed in Section 3.2. At the same time, we will extend and formalize our language and undertake more elaborate testing.

Acknowledgments

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ).

References

- [1] James Mortleman. Humans to blame for security breaches. <http://www.vnunet.com/news/1154153>. "April 7th, 2004".
- [2] Jonathan D. Moffett and Morris S. Sloman. Policy hierarchies for distributed system management. *IEEE JSAC*, 11(9), 11 1993.
- [3] K. Yang, A. Galis, and C. Todd. A policy-based active grid management architecture. In *IEEE ICON'02*, August 2002.
- [4] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5-47, 1998.
- [5] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02*, pages 27-32. ACM Press, 2002.
- [6] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT*, 17(4):40-52, 1992.
- [7] Adrian A. Hopgood. *Intelligent Systems for Engineers and Scientists*. CRC Press, 2 edition, 2000.
- [8] David D. Clark, Craig Partridge, J. Christopher Ramming, and John T. Wroclawski. A knowledge plane for the internet. In *SIGCOMM03*. ACM Press, 2003.
- [9] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 5 2001.
- [10] Mario Cannataro and Domenico Talia. Semantics and knowledge grids: Building the next-generation grid. *IEEE Intelligent Systems*, 19(1):56-63, 2004.
- [11] Nicholas R. Jennings David De Roure and Nigel R. Shadbolt. The semantic grid: Past, present and future. *IEEE Proceedings*, 3 2005.
- [12] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *LNCS*, 2150, 2001.
- [13] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. Ffpf: Fairly fast packet filters. In *Proceedings of OSDI'04*, 2004.
- [14] INMOS. *The occam 2 Reference Manual*. Prentice-Hall, 1988.

- [15] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *761*, page 55. CWI, ISSN 1386-369X, 31 1998.
- [16] Thilo Kielmann. prom: A flexible, prolog-based make tool. Technical Report TI-4/91, Technical University Darmstadt, 1991.
- [17] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.
- [18] D. Garlan, V. Poladian, B. Schmerl, and J. Sousa. Task-based self-adaptation. In *WOSS'04*, 2004.
- [19] John Mark Agosta and Simon Crosby. Network integrity by inference in distributed systems. In *NIPS03*, 2003.
- [20] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *WOSS'02*, 2002.
- [21] J. Dietrich, A Kozlenkov, M Schroeder, and G Wagner. Rule-based agents for the semantic web. *Journal on Electronic Commerce Research and Applications*, 2(4):323–38, 2003.
- [22] Jason Maassen, Rob van Nieuwpoort, Thilo Kielmann, and Kees Verstoep. Middleware adaptation with the delphoi service. In *agridm2003*, 2003.
- [23] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *HPDC01*, 2001.
- [24] Dekai Li and Jonathan M Smith. Towards application-private networks (apnets), 2003.
- [25] Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. Virtual private grid: a command shell for utilizing hundreds of machines efficiently. *Future Gener. Comput. Syst.*, 19(4):563–573, 2003.
- [26] E. Willink. *Meta-compilation for C++*. PhD thesis, University of Surrey, 1999.