# ON ADAPTABILITY IN GRID SYSTEMS

Artur Andrzejak, Alexander Reinefeld, Florian Schintke, and Thorsten Schütt
*Zuse Intitute Berlin*
*Berlin-Dahlem, Germany*
<surname>@zib.de

**Abstract** With the increasing size and complexity, adaptability is among the most badly needed properties in today's Grid systems. Adaptability refers to the degree to which adjustments in practices, processes, or structures of systems are possible to projected or actual changes of their environment.

In this paper, we review concepts, methods, algorithms, and implementations that are deemed useful for designing adaptable Grid systems, illustrating them with examples. Contrary to the existing literature, the portfolio of the proposed approaches includes unorthodox tools such as game theory. We also discusses methods which have not been fully exploited for purposes of adaptability, such as automated planning or time series analysis. Our inventory is done along the stages of the feedback loop known from control theory. These stages include monitoring, analyzing, predicting, planning, decision taking, and finally executing the plan.

Our discussion reveals that several of the problems paving the way to fully adaptable system are of fundamental nature, which makes a 'quantum leap' progress in this area unlikely.

**Keywords:** adaptability, non-functional properties, autonomic computing, decentralized service architecture

# 1. Introduction

During the last four decades, system architects have mainly focused on performance issues. Their quest for performance was overly successful, but only at the cost of an increased software complexity which makes computer systems difficult to operate and maintain. Vertical software integration like the popular Service Oriented Architecture (SOA) helped reducing the barriers for system use, but if something fails only experienced software experts are able to trace down through the many software layers to the source of failure.

This is especially true for networked computers which are operated in changing environments with variable user needs. Leslie Lamport's saying *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"* tells about the vulnerability of distributed environments. The aggregation of many independent heterogeneous subsystems to a well-functioning Grid causes much administration overhead. Since human operators are costly, slow and error-prone, advanced self-management properties are needed that are able to cope with resource variability, changing user needs and system faults.

This paper deals with adaptability in future Grid systems. Adaptable middleware is able to mask changes in the execution environment. Such changes may be caused by variations in the availability of processors, networks, storage. Adaptable middleware is also self-stabilizing [11], that is, it recovers after faults without global re-initialization. This is a permanent optimization process, which is executed in a closed feedback loop.

The paper is organized along the stages of a feedback loop. In the following section, we recall the basics of feedback loops and self-stabilization known from control theory. Thereafter, we investigate the stages in more detail: models, policies and goals in Section 3, analysis and prediction in Section 4, and planning and decision taking in Section 5. We conclude the paper with a discussion of challenges and limitations.

# 2. Feedback Loop

Adaptability refers to the degree to which adjustments in practices, processes, or structures of systems are possible to projected or actual changes of its environment. Adaptation can be spontaneous or planned, and be carried out in response to or in anticipation of changes in conditions.

**Open / Closed Loop.** Adaptable systems are in a state of continuous self-regulation [11, 16] through a *feedback loop*. Deviations of output from some ideal or desired state are fed back into the control unit, which then acts to minimize the discrepancy.

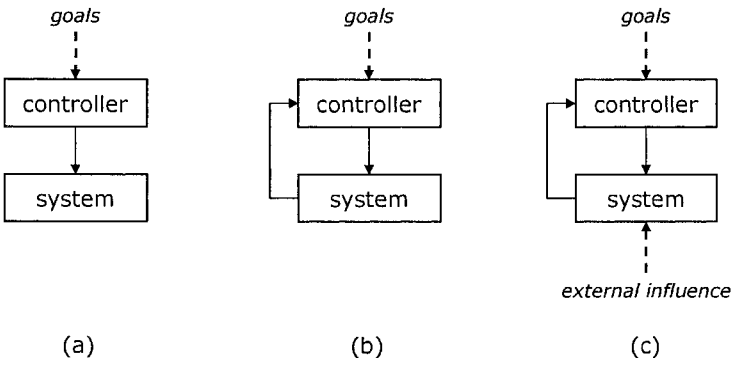



*Figure 1.* Three feedback loop configurations.

We distinguish three kinds of feedback loops [16] as illustrated in Figure 1: an open loop, a closed loop, and a closed loop with external input. Part (a) illustrates the static open loop configuration. The user defines the goal state and hands it to a controller, which drives the system. No feedback is involved. Part (b) shows a reactive configuration, where the controller is able to observe the impact of its actions via a feedback loop. Finally, part (c) illustrates a closed loop with additional impact from outside the system. The controller observes the system and indirectly reacts on external changes.

Open loops are commonly used when services act on static data or when they quickly return some status information. As an example, a service that submits a parallel job to multiple sites may retrieve a list of available CPUs, and assumes that all CPUs are still available after it checked which of the resources match the job's requirements. Closed loops, in contrast, are used for continuously running services, like the load-balancing of file availability in peer-to-peer systems [28]. In large Grid systems, closed loops with external input are perhaps the most important model. This is because local site administrators may change resources or services at any time without prior notice.

**Autonomic Grids.** The described feedback loop is sometimes referred to as 'autonomic'—a term with a biological connotation. It indicates the unconscious self-regulation [22] within human bodies and economic systems.

Each autonomic element consists of one or more managed systems and a control cycle, as shown in Figure 2. Sensors (not shown in the figure) observe behavioral characteristics of the system and report them to a monitor. The *monitor* collects, aggregates and filters the data and logs them for further use. An *analyzer* provides functions and mechanisms for correlating complex situations. The *planner* constructs actions that are needed to achieve the user-specified goal from the current status. As there are often multiple ways to achieve the goal, the

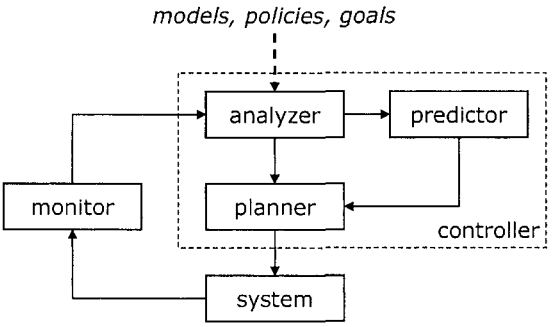
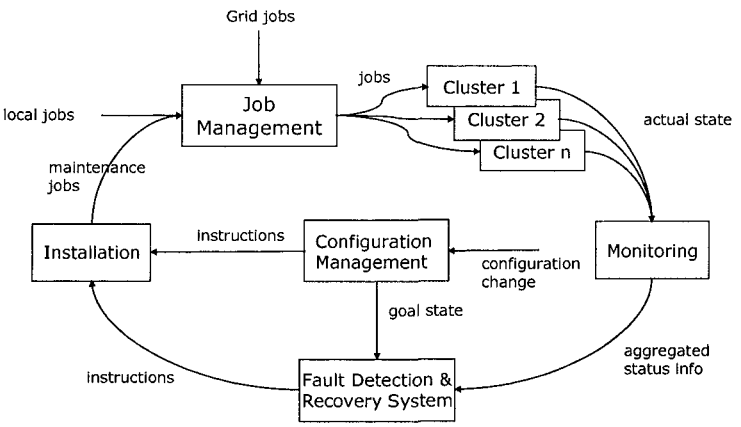



*Figure 2.* Feedback loop in more detail.

planner may be guided by the results of a *predictor* which determines forecasts based on time-series analysis. This allows the system to 'interpret' situations and predict future behavior. An execution unit (not shown) applies the actions to the system by means of one or more actuators. All components exchange information via appropriate protocols and some aggregated data is stored for later analysis and tracing. With few exceptions (see the DataGrid example below) autonomic elements in Grids are not yet present, both on the level of individual servers as well as on the level of clusters and virtual organizations. Thus, the above picture represents a vision which is likely to be found in the future generation of Grid systems, but not in today's systems.

**Example – DataGrid**

*In the course of the European DataGrid project, a feedback loop has been implemented for the autonomous management of PC clusters which are embedded into a worldwide Particle Physics Grid for the analysis of the LHC data.*




*The above figure illustrates the architecture of the cluster maintenance system [29]. A job manager accepts local and Grid jobs and assigns them to the*

*various clusters in a site. A monitor continuously checks the cluster, aggregates the system data and reports it to a fault detection and recovery system. The latter compares the data to the goal state given by the configuration manager and takes appropriate actions when both states diverge. If, for example, some specific software is not available on the monitored cluster (e.g. after adding some newly purchased PCs), the fault detection and recovery system injects a maintenance job for installing the software on the compute nodes. This is done without affecting other jobs.*

**Multivariable Feedback Loops.** In [16] several approaches are shown how multivariable and non-linear control loops can be implemented. The simplest approach of controlling each pair of input and output parameters independently as a single-input, single-output system may lead to undesirable oscillations, so-called thrashing effects. They can be avoided by using controllers with Linear Quadratic Gaussian optimal control theory (LQG) [16], which allows to optimize multi-input, multi-output systems by minimizing a quadratic cost functional and respecting the presence of Gaussian white noise disturbances in the system.

**Errors in Control Loops.** While control loops are made to cope with over/underloaded, unavailable or erroneous external systems, they are themselves often not free of errors. Errors may be transient or permanent. Both must be coped with. Depending on the stage at which an error occurs in the control loop, we distinguish three types: mistakes, lapses and slips. *Mistakes* occur in the planning phase due to insufficient information or lack of expertise. *Lapses* occur between the planning and the execution phase. Usually a plan is not applied immediately, but its steps must be memorized (stored) for later use. Any misses between the formulation of the intended actions and their execution are termed lapses. *Slips* are caused by insufficient skills in the execution phase. While these categories have been defined in the context of human errors [27], they analogously apply to distributed systems. Especially, mistakes and slips may occur in the planning and execution phase (ref. Figure 2).

## 3. Models and Policies

The actors of a feedback loop operate in the context of managed Grids, and therefore require a model of the managed entities, their relationships and possible actions. Also the flow of information between the stages of the feedback loop and to/from the external interfaces requires a specification of an exchange format, both in syntax and in semantics. Such models and their description can be application dependent, for example having a form of C-like records or a file with a proprietary structure. However, for the sake of re-usability it is preferable to make them generic and to standardize them. Such a standardization facilitates the exchange of problem cases, best practices, or system descriptions between

different system administrators or even enterprises, significantly lowering the formalization effort. A universal set of models would ensure the exchangeability of feedback loop components such as analysis and decision algorithms without the need for rewriting or adapting the self-management framework.

## 3.1 Models

The mentioned model and its description ideally would be able to capture the architecture of the managed system, its state, the allowed management actions, desired target system states and the optimization goals. In other words, the standard must be able not only to capture the managed entities including their state and their relationships, but also the dynamic aspects of the management process. Currently, none of the existing specification frameworks adheres to all of these requirements, but the specification standards discussed below are likely candidates to be extended in this direction.

**CIM.** The *Common Information Model (CIM)* [10] is a conceptual model and language standard for describing computing and business entities in Grid, enterprise and service provider environments. It is an ongoing effort by the Distributed Management Task Force (DMTF), a non-profit collaborative body comprised of academic and industrial members that is leading the development of management standards for computing system environments.

CIM is comprised of a *schema* and a *specification*. The schema provides the actual model descriptions of the managed entities and their relationships in an object-oriented way. The CIM schema includes for example models for systems, applications, networks (LAN) and devices. The CIM specification is the language and methodology for describing management data, i.e. it covers the 'syntax' part of CIM and the way how the models are exchanged. As for the later aspect, the standard language used to define elements of CIM is *Managed Object Format (MOF)*, which is based on the Interface Definition Language (IDL). An XML-based description language, xmlCIM, has also been introduced as part of an HTML-based protocol for exchanging CIM information.

While CIM is surely the most widely adopted standard in the area of system management, it does not allow the specification of management actions to a degree required by an adaptive Grid infrastructure. The CIM schema policy model comes most closely to the description of management actions, yet it allows only constructs of the form

if <condition(s)> then <action(s)>.

This limited form does not admit e.g. declarative action specifications. However, a part of CIM is the CIM meta-model which allows for an elegant introduction of new features into this standard, thus rendering CIM as the most likely candidate to fulfill all requirements stated above.

**SDL.** The *Specification and Description Language (SDL)* [19] has been introduced by the Telecommunication and Standardization Sector of ITU as a means to describe behavior, data and structure of particularly larger systems. SDL was originally targeted for specifying telecommunications real time systems, for example call and connection processing in switching systems, maintenance and fault treatment in such systems, or data communication protocols. One unique feature of this standard is the graphical representation with text elements, which greatly enhances the readability of the specifications. The latest major revision of SDL, the *SDL-2000*, is completely based on object orientation.

In SDL the basic specification concept is an *agent*, which is an instance of an extended finite communicating state machine with its own signal input queue, life cycle, and reactive behavior specification [12]. This notion generalizes the former SDL concepts of a system, block and process. An agent is specified by its attributes (parameters, variables, procedures), behavior in the form of implicit or explicit state machine, and internal structure, i.e. contained agents and communication paths.

The last part indicates that all elements are specified in SDL as a hierarchy of simple and composite agents. It is possible to determine the scheduling semantics of an agent by choosing an appropriate container: a *block* indicates concurrency of the subelements, and a *process* an alternating execution. The communication between agents is specified by means of *channels*, *gates* (endpoints of channels), and *connections* (joining/splitting of channels at implicit gate) [12]. An essential part of SDL is the specification of the agent behavior through state machines consisting essentially of *states* and *transitions*.

**CIM versus SDL.** While the primary intention of the current CIM version is to describe system structures and component relationships, SDL focuses on the description of system behavior, resembling a high-level programming language. It lags behind CIM by not offering prepared 'templates' for commonly encountered components (covered in CIM by the CIM Schema Common Models) and by its limited extensibility. For example, a declarative specification of behavior would be hard to express in this standard. Also the exchange of the SDL-models is not easy due to the graphical nature of the language and a missing mapping to XML. On the other hand, SDL provides richer means to express behavior and data transfer than other standards.

## 3.2 Policies

A policy is a *definite goal, course or method of action to guide and determine present and future decisions* [20]. Traditionally, the term *policy* is used to describe parts of the system configuration that controls system behaviors such as in security policies or quality-of-service policies. This covers for example rules specifying event-triggered actions as well as goals to be achieved by the

management actions. Policies have been most successfully used in the area of networks, where they specify traffic priority issues, access control, quality of service and other aspects. There are many approaches to describe policies, including logic-based languages or Role-Based Access Control (RBAC) specification in the security area, and CIM, Policy Description Language (PDL), or event-trigger-rules in the system management area [8]. The most sophisticated policy description language is perhaps *Ponder* [9] which can be used for specifying management and security policies. It is a declarative, object-oriented language that supports a variety of features, for example various access control mechanisms for firewalls, operating systems, databases and Java, and event triggered condition-action rules for management of networks and distributed systems.

## 4. Analysis and Prediction

Modeling and predicting the demand of individual servers or their clusters is one of the key supporting techniques for the automated management and scheduling of computing resources. In Grid environments, modeling and prediction of the future application demand facilitates the performance tuning, anomaly detection, scheduling of jobs, sharing of resources, capacity planning, or discovering interdependencies between applications.

The usefulness of modeling and prediction for self-management depends not only on the accuracy, but requires some additional, scenario-specific features:

- *human-readable models* – these allow for plausibility checks and help improving methods and results
- *assessment of the prediction confidence* – knowing how much we can trust a prediction facilitates the choice of a management strategy, e.g. an application with less predictable demand might receive a dedicated server, while predictable applications might be admitted for resource sharing
- *exploitation of data correlations* – by taking into account the compute demand traces from the whole cluster, a model might be more accurate than one based on a single trace
- *computational efficiency* – to ensure a wide acceptance of the technique, the overhead caused by modeling should be negligible compared to the computational cost of the application itself
- *long-term accuracy* – at least some of the methods should be long-term accurate to facilitate capacity planning.

We take a closer look at three methods which at least partially fulfill the above list of features: (1) classical ARIMA/Kalman filter timeseries modeling,

| | ARIMA/Kalman | Classification | Sequence Mining |
|---|---|---|---|
| **Human-readable** | no | partially | yes |
| **Confidence Assessment** | partially | partially | partially |
| **Exploiting Correlations** | yes | yes | no |
| **Computational Efficiency** | yes | yes | yes |
| **Long-term Accuracy** | no | partially | yes |

*Table 1.* Feature matrix of the considered prediction methods.

(2) classification based on data mining methods, and (3) mining of repetitive patterns in the demand by means of sequence mining. Table 1 shows the fulfilment of the required features by these methods.

**ARIMA.** This is a classical timeseries decomposition method used in econometrics [23]. It assumes that a future sample value can be estimated as a linear combination of past sample values (the *autoregressive*, 'AR' part of 'ARIMA'), and a linear combination of past prediction errors (the *moving average*, 'MA' part). To remove trends and effects of seasonal fluctuation, the series might be differenced (the *integrated*, 'I' part) prior to the decomposition. The disadvantages of the method are its high computational cost in estimating the coefficients of the linear combinations, and the rapidly decreasing accuracy if the coefficients are not recomputed on new data (i.e. the method admits only a small forward leg).

As a partial remedy, the computationally efficient Kalman filter [23] can be applied after an ARIMA model has been obtained. This so-called *state space* approach updates with each sample an internal 'black box' model (initially given by ARIMA) to minimize the prediction errors. To take advantage of the potential correlations in the input data, the multivariate versions of both approaches can be used.

**Data Mining and Classification.** Data mining originally focused on finding regularities in consumer behavior, yet it quickly found applications in other fields such as bioinformatics, health care, or system management. A typical application in the latter field is the detection of anomalies caused e.g. by denial-of-service attacks, or failure of system components. It also plays a major role in a field called *recovery oriented computing (ROC)* [6], a research initiative devoted to the problem of fast and non-intrusive recovery from failures as opposed to avoiding them. Here data mining tools allow to identify a critical system state which requires a rejuvenation from a normal state or a transient error. Data mining provides several tools for performing prediction, the most important two being classifiers and algorithms for mining association rules.

*Classifiers* can be seen as functions which take as arguments certain easily obtainable system characteristics (*attributes*), e.g. load in the last 15 minutes, number of processes, disk activity in bytes etc. and return a value representing a likely system state (*class value*), e.g. 'all ok', 'probable denial-of-service attack', 'major system malfunction', etc. The key advantage is that a classifier learns from provided examples (tuples of attribute values and the class value) and can be regarded as a black box after the training. Simple yet fast and accurate classifiers include decision trees, Bayesian methods, or $k$-nearest search. More elaborate methods such as neural networks or support vector machines might provide increased accuracy at the higher cost of training. Classifiers allow for exploitation of data correlations, and as a by-product, the data preparation stage might identify the dependencies between the demand of different applications.

*Association rules mining* attempts to retrieve additional information in form of explicit rules involving attribute and class values. The user can specify the interestingness criteria by which the rules are selected. In classical data mining applications such as modeling of consumer behavior the rule frequency is a common criterium [13]. In the domain of system adaptability we will be also searching for rare but important events, such as system overload, component failure, or a deadlock. A subsequent analysis (currently still performed manually) is required to distinguish whether a discovered rule represents an important relationship or is due to a pure chance only.

**Sequence Mining.** Sequence mining methods allow for discovery of repetitive patterns in time series, such as increased server load at certain times during the week. An approach which takes into account the possibility of changes of the patterns over time is presented in [1]. In this method, the results of the mining phase can be applied to make long-term predictions, provided the patterns are not likely to change significantly in the forecast horizon. The results are also human-readable and might help to identify inefficient resource usage or false configurations. Unfortunately, the correlations between different application traces cannot be taken into account.

## 5. Planning and Decision Taking

**Automated Planning.** A plan is a partially ordered collection of actions for performing some task or achieving a certain goal. There are many elaborate programs supporting human planners, such as project management tools or automatic schedule generators. However, *automated planning* is much more difficult, with many research prototypes and few practical systems. While the research has still to cope with a host of difficult practical problems in this domain, mostly involving computational complexity, there are several successful cases such as the NASA DS1 mission [4].

Automated planning arose as a field of artificial intelligence due to the need for affordable and efficient planning tools. The application domains of such tools include complex and changing tasks which require a high degree of safety or efficiency, or autonomy and self-control capabilities in artifacts such as robots or spacecrafts. In the field of Grid systems, automated planning can be used to automate complex tasks involving heterogeneous resources and having a lot of interdependent steps. Examples of such tasks are the migration of distributed jobs including data transfer and software installation, or the construction of complex resources such as virtualised server farms on demand. Given the specifications of possible actions in such an environment, an automatically generated plan can be translated into a standardized workflow and distributedly executed [3].

The major problem of the research on automated planning is the size of the search space. Of course, the planning problems are not trivial—the complexity of the problems PLAN-EXISTENCE and PLAN-LENGTH range from NP-complete to EXPSPACE-complete in all but few trivial cases [15]. However, many practical cases can be solved more efficiently, for example by incorporating domain-specific knowledge in form of specialized algorithms. Currently, general-purpose automated planning can solve smaller problem instances but requires substantial tuning and adjusting for larger problem sizes.

The progress in the research on automated planning is recorded in the yearly International Planning Competitions [14], which provide a chance to compare prototypical general-purpose and specialized planners on a set of realistic problem cases. In the course of the competitions, a common standard for problem and goal description has been established—the *Planning Domain Definition Language (PDDL)*. This declarative language allows for specification of predicates, functions, actions and other objects in a Lisp-like notation, acting as a widely adapted front-end to most of the research and production planners. Recent versions of PDDL have been enriched with means of temporal planning, which encompass relations between time intervals, precedence and 'deadlines' of action executions, and others.

Some of the major methods deployed in classical planning include planning-graphs, propositional satisfiability techniques, and constraint satisfaction techniques [15]. The increase of efficiency is usually addressed by heuristics such as guided forward-search, and control rules.

**Optimization.** Optimization plays a central role in the tuning of system parameters, e.g. for increasing performance or allocating resources. In its most general form, an optimization problem consists of a set of variables for which value assignments are sought, a set of constraints imposing relationships between these variables, and possibly one or more objective functions whose values should be maximized or minimized. Note that if we drop the objective

function(s) and just seek an assignment of values to variables which satisfy the constraints, than this definition also covers satisfiability and constraint satisfaction problems.

If the variables take real values, and both the constraints and the objective function is linear in the variables, such an optimization problem is called a *linear program.* Such problems belong to a benign class of problems for which it is not only theoretically possible to find a provably optimal solution in polynomial time, but which also can be solved fast in practice. Unfortunately, already small variations of the problem such as discrete variable domains or higher-order constraints make the problems hard to solve optimally both in theory and in practice. As to annoy the computer scientists striving for adaptability of systems, *real* optimization problems in system management are rarely representable as linear programs.

**Example – Job Assignment**

*Consider the problem of assigning jobs to servers in a server-consolidation scenario of a large cluster [2]. Here we model the situation that a job $j$ is hosted on a server $s$ by setting a variable $x_{s,j}$ to* 1*, and setting it to* 0 *otherwise. Using such binary variables renders this problem* NP*-complete and makes it costly to solve optimally in practice, even if all constraints and the objective function remain linear. If we want to further minimize the total communication cost between the processes, the objective function becomes quadratic, additionally increasing the problem's complexity.*

However, not every problem must be solved optimally—in most cases a good or even any solution will do, and here heuristics come into play. While in many instances heuristics such as simulated annealing or genetic algorithms work much faster than the exact algorithms, there is a trade-off: heuristics do not guarantee an optimal solution, and usually it remains unknown by how much the found solution is worse than the optimum. Moreover, if a solution has not been found by the heuristics, it does not mean that none exists. Finally, some problems are even too hard for the generic heuristics, and require a time-consuming design of more efficient problem-specific approaches. This particularly applies to the domain of scheduling, e.g. job-shop problems, or the replica placement problems [21].

**Example – Genetic Algorithms**

*Genetic algorithms [17] are particularly popular heuristics in the optimization. They have proved to be robust and efficient for a multitude of problems, including complex multi-criteria optimization problems such as design of an airplane propeller profile. The basic idea of a genetic algorithm is very simple. At each optimization phase, a pool (*generation*) of potential problem solutions (*genes*) is maintained in memory. A transition between generations is achieved by tweaking the genes (by* mutation *and* cross-over*), and the subsequent evaluation and selection of the best among them. After a fixed number of transitions or other stopping criteria the best gene becomes the problem solution. The quality of the solution increases with growing number of genes and number of transitions.*

*While some effort is necessary to encode a potential solution as a gene and to design the mutation and cross-over operators, the approach is very flexible - the gene can contain both discrete and real-valued variables at the same time, and the objective function(s) might include rules and algorithms, not only expressions. Furthermore, genetic algorithms can be easily parallelized with high speed-up factors.*

**Expert systems.** Another approach for planning and decision taking are traditional expert systems from the field of artificial intelligence, studied since the 1970s. An expert system consist of a model of the problem space and an inference engine which analyses a given state and either proposes actions to improve the state or derives new facts about the current state (diagnosis). Often they maintain a decision tree as part of their internal data structure and take paths in the tree depending on the monitored data. If a leaf in the tree is reached, it is annotated with an appropriate action for this combination of parameters. Such systems can be used both for the analysis and the planning inside adaptable systems. If decisions were taken wrongly, the tree may be updated automatically or manually. In the longterm, the necessary knowledge to properly react on environmental changes is collected and can be automatically applied when similar situations occur.

**Example – Prolog**

*Prolog is a logical programming language, which is mainly used in natural language processing and expert systems. Its first use goes back to 1972 when Colmerauer et al. [7] initiated a research project to create a natural language processing system, where they implemented the language processor in Prolog. Shortly after this project, also a symbolic computation system and a problem solving system were developed in Prolog.*

*Internally Prolog is able to derive variable assignments that satisfy so called 'facts' and 'rules' by a built-in backtracking mechanism. Using these facts and rules, the goals and policies and actions of a system can be expressed without the need to know which of the actions have to be applied to achieve the goals. Prolog allows even during runtime to add new facts and rules to the knowledge base, which makes Prolog systems powerful and flexible.*

*Applied to replica management, facts may describe the current location of replicas and their sizes as well as the available disks, network topology, and replica availability, while rules describe the conditions for a replica migration. Then Prolog would be able to answer basic questions like: 'Can all replicas be stored without violating storage constrains?', 'How can replicas from sites A, B and C be transmitted to a target site D with minimal overall network traffic?'. Additionally constraints, which are a common extension to Prolog systems, may be added to specify disk capacities for example.*

**Game Theory.** Often, multiple goals must be fulfilled, resulting in an optimization problem with multiple feedback loops. These may be either (1) disjunct, (2) partly overlapping, or (3) contradicting in their goals.

The first case, i.e. disjunct goals, is the easiest to handle, because the actions do not affect each other in their optimization process. The optimization in each loop may independently strive for its optimum without compromising any other loops' goals.

The third case, i.e. contradicting goals, corresponds to a zero-sum-game with $n$ non-cooperative players ($n$ loops). As first proved for adversary games ($n = 2$) by John von Neumann in his less known German contribution from the year 1928 [26] and again published in the much later, but epoch-making monograph 'Theory of Games and Economic Behavior' [24], there exists always a solution to this problem.

More difficult is the second case with partly overlapping goals. It may be solved by John Nash's generalization of von Neumann's minimax theorem, the *Nash equilibrium* [25]. One popular example is the *prisoner's dilemma*, where a selfish maximization strategy does not lead to a joint optimum, because each prisoner chooses to defect although the joint payoff of the players would be higher by cooperating—hence the dilemma. Adaptive Grid systems are, however, in a continuous optimization process which corresponds to the *iterated prisoner's dilemma*. Here the prisoners game is played repeatedly, thereby giving each player the opportunity to punish its opponent for previous non-cooperative play, which eventually leads to a superior cooperative outcome after several rounds of learning.

## 6. Challenges and Limits

**Service Semantics.** In dynamic Grids, services should be exchangeable. This is currently done using catalogs where all services are registered with their interface and name. When two services have the same interface and service name they are assumed exchangeable. Of course this is not necessarily the case, because the semantics of the services may differ. Among other problems, this has great impact on the reliability of Grid systems. If it is possible to register services with the same interface and name, that may be chosen but do *not* what the application expects them to do, denial of service attacks become possible. If the semantics would be formally specified as part of the service description, one would have to check whether the service semantics match the required semantics. Taking this to the extreme we could search for some composable services that—in combination—provide the desired semantics. This, however, is equivalent to an automated theorem prover, which is known to be very compute intensive.

**The Formalization Challenge.** We have discussed above the limits of self-managing solutions due to the inherent complexity problems. However, another factor is likely to become a key bottleneck towards self-management: the *formalization challenge.*

Consider a domain which bears strong similarities with the core of autonomic computing - creation of software. The purpose of software can be in essence interpreted as the automation of 'tasks'. Compared to the rapid and sustainable progress in hardware development, the gain of efficiency in software development lags behind on the orders of magnitude. This well-known fact is usually attributed to the effort of formalization. Simplifying, the formalization process has to do with the interaction between 'what we want' (what we expect from the program) and how to force the machine to behave in this way [5]. Of course, the formalization problem is more manifold, e.g. due to the variable specifications caused by the fact that the expectations of users change when they start working with the software.

The programming problem certainly generalizes the autonomic computing problem, since in all by few exceptions the means to attain the self-managing functionality is software. Does it mean that the effort of formalization for self-management is similarly high as in the programming problem? This is not necessarily the case, since in the domain of self-management the required solutions are simpler (and more similar to each other) than in the field of programming, and so the benefits of domain-specific solutions can be exploited.

A further step to reduce the effort of formalisation would be the usage of machine learning to automatically extract common rules and action chains from such descriptions [3]. Other tools are also possible, including graphical development environments (e.g. for workflow development), declarative specification of management actions used in conjunction with automatic planning, or domain-specific languages, which speed-up the solution programming.

**Complete fault-tolerance is neither possible nor beneficial.** One goal of autonomic computing is to hide faults from the user and to first try to handle such situations inside the system. Some faults cannot be detected, like whether an acknowledgement or calculation just takes a very long time, or was lost during data transmission. This is also known as halting problem [30] which states that no program can decide whether another program contains an endless loop or not.

Also in some cases it would be not a good idea to try to hide errors. If, for example, the user specified a non existing file as input data, the system should immediately report this back to the user and should not try to hide this error by waiting until such a file may be created sometimes in the future.

Finally, automation of management tasks does not come without cost, and in some cases this effort does not pay off, e.g. if such a task is very rare. Measuring or even estimating the cost of automation can help to decide whether it is cheaper to leave some scenarios not automated.

## 7. Conclusion

In this paper we have discussed requirements, features, and possible approaches of the adaptive Grid systems in context of the feedback loop. Such a loop is inherent in self-managing systems, and includes as the essential stages monitoring, analysis/modeling, decision taking and execution. The processing of information in such a framework requires a definition of models of the managed systems and the management actions, as well as goals and policies governing the decision process. A suitable specification language is still to be found, yet CIM, possibly in conjunction with SDL seems to be a appropriate candidate.

The analysis part of the feedback loop instantiates the models and extracts knowledge from information provided by the monitoring system. Our selection of the methods in this domain included the field of demand and event modeling and their prediction, data mining approaches, and the contribution of game theory to tackle the multiple loop problem.

The decision taking stage of the feedback loop evaluates the preprocessed input from the analysis part, and plans management decisions in accordance with the specified goals and policies. Automated planning, still not a very developed field of AI, might become a fundamental technique to tackle complex management scenarios involving many interdependent steps. Another essential contribution to decision making comes from optimization techniques, especially heuristics such as genetic algorithms. This is due to the fact that in the self-management scenario a good solution which can be found efficiently is superior over optimal solution which requires extensive computation.

The self-management challenge has created a lot of research activities, yet real 'breakthroughs' seem to remain elusive. Our work illustrates that reaching this goal requires a lot of effort and improvements in a multitude of fields, which makes singular 'quantum leaps' unlikely. A partial reason for this is the fact that many problems to be solved are not new, but of a more fundamental nature, for example in the field of automated planning. This does not exclude the fact that qualitatively new possibilities in systems management will arise once a certain threshold of the progress has been crossed. A good example is the development of computer hardware, where steady progress over decades has led to application areas unthinkable of on the offset of the journey.

## 8. Acknowledgements

This research work is carried out in part under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

## References


[1] A. Andrzejak and M. Ceyran. *Characterizing and Predicting Resource Demand by Periodicity Mining*. Journal of Network and System Management, special issue on Self-Managing Systems and Networks, Vol. 13, No. 1, Mar 2005.

[2] A. Andrzejak, J. Rolia, and M. Arlitt. *Bounding the Resource Savings of Several Utility Computing Models for a Data Center*. HPL Technical Report HPL-2002-339, Hewlett-Packard Laboratories Palo Alto, December 2002.

[3] A. Andrzejak, U. Hermann, and A. Sahai. *Feedbackflow - An Adaptive Workflow Generator for System Management*, 2nd IEEE International Conference on Autonomic Computing (ICAC-05), 2005.

[4] D. Bernard, E. Gamble, N. Rouquette, B. Smith, Y. Tung, N. Muscetola, G. Dorias, B. Kanefsky, J. Kurien, W. Millar, P. Nayak, and K. Rajan, *Remote Agent Experiment. DS1 Technology Validation Report.* NASA Ames and JPL report, 1998.

[5] M. Broy and R. Steinbrüggen. *Modellbildung in der Informatik*. Springer-Verlag, Berlin, 2004, ISBN 3-540-44292-8.

[6] G. Candea, A.B. Brown, A. Fox, and D. Patterson. *Recovery-oriented computing: Building multitier dependability.* IEEE Computer, Nov. 2004, pp. 60–67.

[7] A. Colmerauer and P. Roussel, *The Birth of Prolog*. 2. SIGPLAN conference on History of Programming Languages, 1993, pp 37–52.

[8] N. Damianou, A. K. Bandara, M. Sloman, and E. C. Lupu. *A Survey of Policy Specification Approaches.*, April 2002.

[9] N. Damianou, N. Dulay, et al. *The Ponder Policy Specification Language.* Policy 2001: Workshop on Policies for Distributed Systems and Networks, Bristol, UK, Springer-Verlag, 2001.

[10] Distributed Management Task Force (DMTF). *DMTF CIM Concepts White Paper.* http://www.dmtf.org/standards/published_documents.php

[11] S. Dolev. *Self-Stabilization.* MIT Press, Cambridge MA, 2000.

[12] J. Fischer and E. Holz. *SDL-2000 Tutorial.* SAM 2000 Workshop Grenoble, 2000.

[13] P. A. Flach and N. Lachiche. *Confirmation-Guided Discovery of first-order rules with Tertius.* Machine Learning, 42, 1999, pp. 61-95.

[14] M. Fox and D. Long, *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains.* Journal of Artificial Intelligence Research, vol. 20, 2003, pp. 61-124.

[15] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning - theory and practice.* Morgan Kaufmann Publishers, 2004, ISBN 1-55860- 856-7.

[16] T. Glad and L. Ljung. *Control Theory: Multivariable and Nonlinear Methods.* CRC Press, June 2000.

[17] D. A. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

[18] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, 2001.

[19] International Telecommunication Union (ITU). *Specification and description language (SDL).* TU-T Recommendation Z.100, August 2002.

[20] The Internet Society. *RFC 3198 - Terminology for Policy-Based Management.* 2001.

[21] M. Karlsson and C. Karamanolis. *Choosing Replica Placement Heuristics for Wide-Area Systems*. Int. Conf. on Distributed Computing Systems (ICDCS), March 2004, Tokyo, Japan, pp. 350 -359.

[22] J.O. Kephart and D.M. Chess. *The vision of autonomic computing*. IEEE Computer, Jan. 2003, pp. 41–50.

[23] S. Makridakis, S. C. Wheelwright, and R. J. Hyndman. *Forecasting - Methods and Applications*. 3rd edition, John Wiley & Sons, Inc., 1999.

[24] O. Morgenstern and J. v. Neumann. *The Theory of Games and Economic Behaviour*. 1944.

[25] J. Nash. *Equilibrium Points in N-Person Games*. Procs. of the National Academy of Sciences, 36, 1950, 48–49.

[26] J. v. Neumann. *Zur Theorie der Gesellschaftsspiele*. Mathematische Annalen, vol. 100, 295–320, 1928.

[27] J. Reason. *Human Error*. Cambridge University Press, 1990.

[28] A. Reinefeld, F. Schintke, and T. Schütt. *Scalable and Self-Optimizing Data Grids*. Chapter 2 (pp. 30 - 60) in: Yuen Chung Kwong (ed.), Annual Review of Scalable Computing, vol. 6, June 2004.

[29] T. Röblitz et al. *Autonomic Management of Large Clusters and their Integration into the Grid*. J. of Grid Computing, 2(3):247–260, September 2004.

[30] A. Turing. *On Computable Numbers, with an application to the Entscheidungsproblem*. Proceedings London Mathematical Society (series 2) vol 42, 1936, pp.230-265.

http://www.springer.com/978-0-387-27935-0

Future Generation Grids
Getov, V.; Laforenza, D.; Reinefeld, A. (Eds.)
2006, XVIII, 308 p. 30 illus., Hardcover
ISBN: 978-0-387-27935-0