High-level Models of Software-management Interactions and Tasks for Gradual Transition Towards Autonomic Computing

Edin Arnautovic, Hermann Kaindl, Jürgen Falb, Roman Popp
Institute of Computer Technology
Vienna University of Technology
Vienna, Austria
{arnautovic, kaindl, falb, popp}@ict.tuwien.ac.at

***Abstract*—For making software systems autonomic, it is important to understand and model software-management tasks. Each such task contains typically many interactions between the administrator and the managed software system.**

We propose to model software-management interactions and tasks in the form of discourses between the administrator and the software system. Such discourse models are based on insights from theories of human communication. This should make them "natural" for humans to define and understand. While it may be obvious that such discourse models cover software-management *interactions*, we found that they may also represent major parts of the related *tasks*. So, these well-defined models of interactions and tasks as well as their operationalization allow their execution and automation. Based on this modeling approach, we propose a specific architecture for autonomic systems. This architecture facilitates gradual transition from human-managed towards autonomic systems.

***Index Terms*—Self-managing systems; autonomic computing; interaction modeling**

I. Introduction

Today's software systems are usually distributed and very complex. They have a large amount of parameters and possible configurations and it is crucial to satisfy their quality requirements such as performance, availability and security. Management of these systems includes tasks required to control, measure, optimize, troubleshoot and configure software in a computing system.

In order to automate software systems' management tasks, it is important to understand and represent them in some more or less formal way. Any such task contains typically many interactions between the administrator and the managed software system. We found that modeling these interactions facilitates understanding and specifying tasks as well. In order to make interactions easy to understand and to specify by humans, their specification should be on a high level.

Thus, we propose to model the software systems' management tasks in the form of *discourses* between the administrator and the system. Such discourse models are based on insights from human communication theories and provide specifications for tasks and their interactions; for the basic approach see [1]. We elaborate on it here and extend significantly both our task and interaction specifications (task and discourse metamodel) as well as our approach to communication content representation (management domain content metamodel).

Although autonomic computing is a challenging vision, truly self-managed systems are hard to achieve and not (yet) in wide-spread use. The processes and means for the transition towards this vision are still not sufficiently investigated. In order to address these issues, this paper presents an approach to gradual transition towards autonomic systems (an earlier sketch of this proposal can be found in [2]).

The core idea of our approach is that the same interaction specification is used both for management by human administrators as well as for autonomic management. More precisely, the same discourse model is used for the automated generation of user interfaces for human management as well as for the specification of the interactions between the autonomic manager and the managed system in the case of autonomic management. Whenever a management task is sufficiently understood and a related implementation available in the autonomic manager, managing this task can be handed over to the autonomic manager without changing its interaction specification in the discourse model. As a consequence, a smooth and gradual transition towards self-managed software systems will be facilitated, where the portion managed by human administrators becomes smaller and smaller.

In contrast to the major body of research on autonomic systems, this approach does not focus on designing and developing autonomic managers per se. However, the transition towards autonomic systems and supporting means seem to be equally important for their acceptance. Our approach contributes to the latter and is, therefore, complementary to work on improving autonomic managers.

The remainder of this paper is organized in the following manner. First, we provide some background on the human communication theories that we build upon. Then we present our running example, that we use to explain our approach to representing interactions and tasks in the form of discourse models. As an important part for the operationalization of such discourse models, we define their procedural semantics. Then we specify both our high-level autonomic architecture and our transition process from human-managed towards autonomic systems. Two case studies indicate the feasibility of our approach. Finally, we discuss related work.

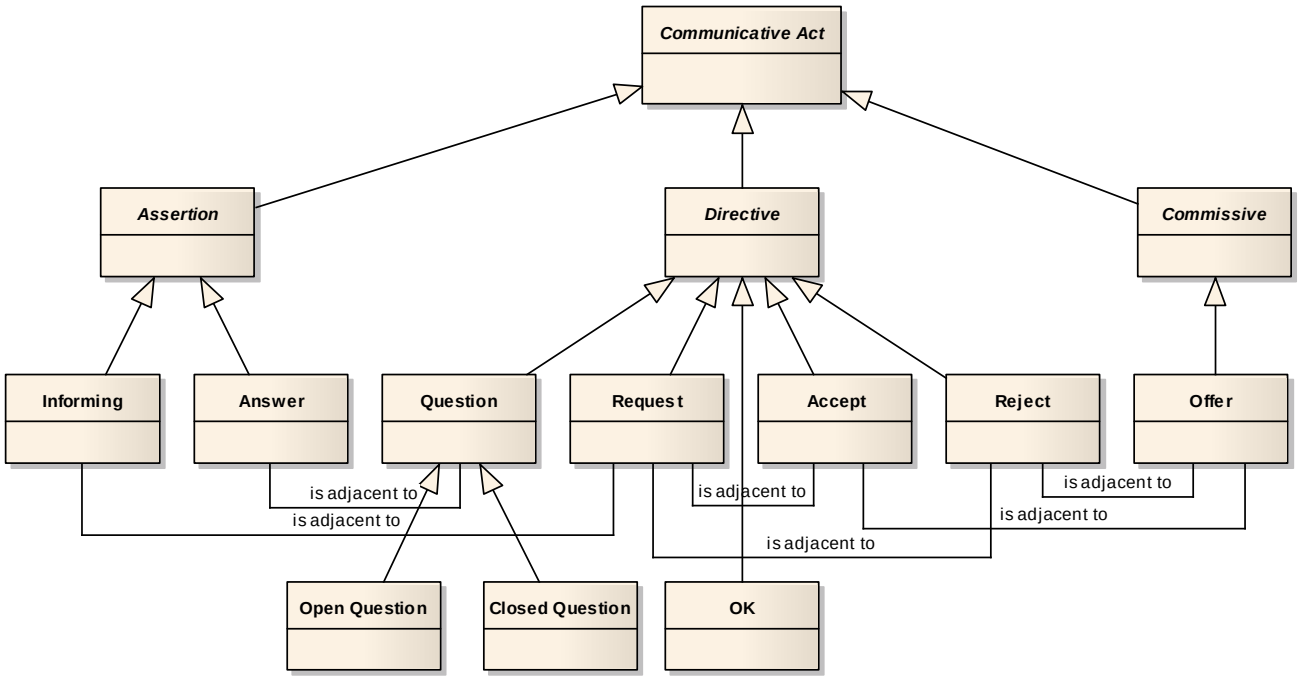

Fig. 1. Part of communicative acts hierarchy.

II. Human Communication Theories

Both tasks and their interactions can be specified in many ways. We strive for a uniform high-level approach to task and interaction representation based on the following human communication theories.

Communicative acts are derived from speech acts [3] and represent basic units of language communication. Thus, any communication can be seen as enacting of communicative acts, acts such as making statements, giving commands, asking questions and so on. *Communicative Acts* carry the intention of the interaction (e.g., asking a *Question*) and can be further classified into *Assertions*, *Directives* and *Commissives*. *Assertions* convey information without requiring receivers to act beside changing their beliefs (e.g., Informing and Answer). *Directives* (e.g., Question, Request, Accept) and *Commissives* (e.g., Offer) require an action by the receiver or sender for the advancement of the dialogue by further communicative acts. This classification is shown in Figure 1. The figure shows only a small selection from many communicative acts. Communicative acts have been successfully used in several applications: inter-agent communication in FIPA Agent Communication Language[1] (ACL), information systems [4] and high-level specifications of user interfaces [5].

Conversation Analysis. While communicative acts are useful concepts to account for intention in an isolated utterance, representing the relationship between utterances needs further theoretical devices. We have found inspiration in Conversation Analysis [6] for this purpose. Conversation analysis focuses on sequences of naturally-occurring talk "turns" to detect patterns that are specific to human oral communication, and such patterns can be regarded as familiar to the user. In our work we make use of patterns such as "adjacency pair" and "inserted sequence".

Rhetorical Structure Theory (RST) [7] is a linguistic theory focusing on the function of text, widely applied to the automated generation of natural language. It describes internal relationships among text portions and associated constraints and effects. The relationships in a text are organized in a tree structure, where the rhetorical relations are associated with non-leaf nodes, and text portions with leaf nodes. In our work we make use of RST for linking communicative acts and further structures made up of RST relations. Thus, they represent the structure of possible interactions between an administrator and the software system. We use two types of RST relations: symmetric, multi-nuclear (e.g., *Joint, Otherwise*) and asymmetric, nucleus-satellite (e.g., *Result, Condition, Elaboration*).

III. Running Example

We use a simplified online store as our running example. The online store application enables the customer to look at and browse through different catalogues and products, to create user profiles, create and manage lists of preferred and desired products. It also allows ordering, shipping management and credit card processing. For the design of this application it is very important to separate data storage and management from data processing and presentation.

Such an application is usually implemented using a multi-tier architecture, with three as a typical number of tiers:

[1]Foundation for Intelligent Physical Agents, Communicative Act Library Specification, www.fipa.org

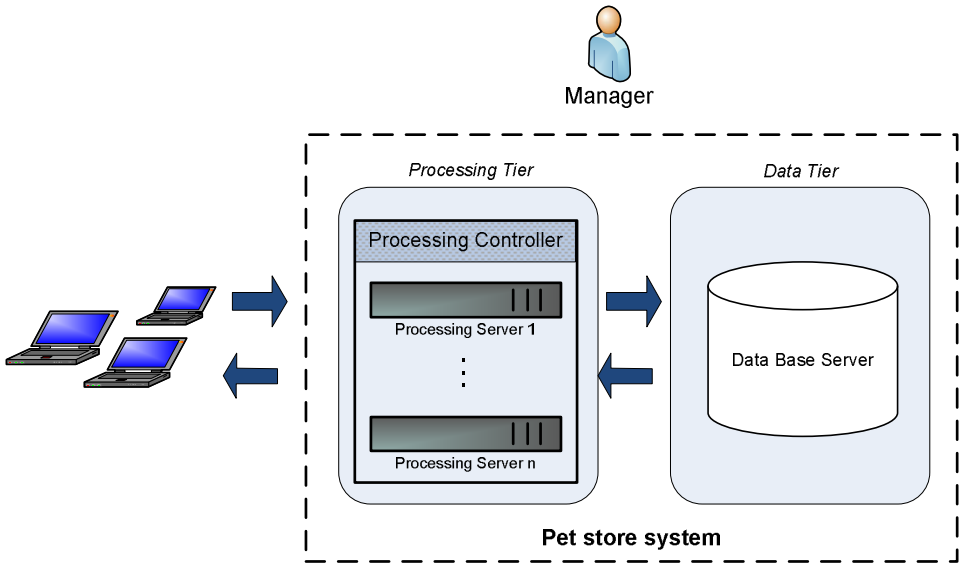

Fig. 2. Online store system architecture.

- presentation tier (e.g., implementing a Web interface),
- application logic tier (e.g., using Enterprise Java Beans), and
- data tier (e.g., using a relational database).

Yet, for our running example we integrate the presentational and logical functionality into one *processing tier*, and thus end up with a two-tier architecture for reasons of simplicity. The resulting online store architecture is shown in Figure 2. The data tier contains one database server. The processing tier contains a cluster with several processing servers, which are controlled by a *Processing Controller* server. A processing controller performs, for example, load balancing. The online store application is deployed in a hosting environment where each of the servers gets some amount of processing power assigned (e.g., using some virtualization technology). It is also possible to dynamically add additional servers and to integrate them into the processing tier. For each server in the architecture, the online shop owner has to pay a certain amount of money. The amount depends directly on the assigned processing power, which can be changed during runtime.

A major goal of the online shop owner is to achieve the best possible customer satisfaction and shopping experience. On the other hand, the owner wants to keep the operation costs as low as possible. One of the most important criteria for customer satisfaction in Web applications is the elapsed time from the page request until the requested page is fully displayed on the screen. This parameter is known as *response time*. Other parameters which influence user experience such as graphical design are not considered in our example. It is evident that the deployment architecture and the characteristics of the included components within this architecture directly influence the system response time. The manager of the shop application has the following possibilities to influence the runtime architecture and characteristics of the online shop:

- Increase or decrease assigned processing power of the database server.
- Increase or decrease assigned processing power of each of the processing servers in the processing tier.
- Add or remove a processing server in the processing tier.

The manager of the shop application can get the following information about the online store's state:

- average response time
- assigned processing power of the database server and of each processing server in the processing tier
- current utilization of the processing power of the database server and of each processing server in the processing tier
- average processing power utilization of the processing tier

The task of optimizing allocation of servers and their respective power to tiers in order to satisfy customers, and thus provide a high quality of service under peek loads and to keep the running costs low at the same time, is known under *provisioning*. Provisioning problems can be dealt with using complex mathematical models and architectures (e.g., according to [8]). Also some other information beyond the system itself can be required (e.g., current expense of the processing power per given unit). However, we do not go into more detail about such algorithms, since we are more interested in the communication which occurs within such management tasks.

In our running example, the manager of the shop application monitors the application's response time. If it happens to rise above a given limit, the manager tries to figure out more details about the cause, by acquiring information about the current runtime architecture, its structure and the properties of the system as a whole and its components. This includes, e.g., average processing power utilization in the processing tier, number of servers in this tier, assigned processing power and processing power utilization of each processing server

as well as assigned processing power and processing power utilization of the database server. After having collected all this information, the manager decides to take some action: either to increase the assigned processing power of one of the servers or to add additional servers to the processing tier. When the manager realizes that the response time falls below a given threshold value, he can remove one of the servers from the processing tier to save operating cost.

For our approach it is important that the control and monitoring features listed above represent the content of the communication between the manager of the online shop system and the system itself. These features serve as the subject-matter that the manager and system are "talking about".

IV. High-level Interaction and Task Specification

We have developed a metamodel based on human communication theories which defines what the structure of the interaction and task models should look like in our approach. We explain it using an interaction specification for a management task based on the running example.

We model the communication between a managed software system and its (human or autonomic) manager in the form of discourses and relate them to the corresponding management tasks. Our conceptual metamodel is shown in Figure 3, specified as a UML class diagram.[2] While related concepts have been used for the modeling of human-computer interaction (e.g., in [9]), we use discourse models additionally for communication within software systems, more precisely between a managed software system and its autonomic manager. In addition to the communication specification, it is necessary to represent the communication content.

The metamodel illustrated in Figure 3 consists of two main parts. The upper rounded box represents discourses and management tasks. A management task represents a typical task of software system management such as system optimization, recovering from errors, etc. Figure 3 shows that a management task is specified by a discourse. A discourse consists of communicative acts, adjacency pairs, discourse relations, and their hierarchical structure and represents the modeling of interactions. This part is also used for general human-machine communication modeling [9][10][11].

The lower part of Figure 3 consists of classes involved in the description of the content to communicate for the purpose of the system's management and, therefore, represents elements of the management domain. There are two different kinds of information to be exchanged between a software system and its manager. These are the current state of the system captured by properties and management actions to be executed by the system.

A. Task and Discourse Metamodel

Our approach to task and communication models in the form of discourses can be sketched as follows. In essence, it has communicative acts as its "atoms", from which "molecular" structures can be composed in two dimensions. First, adjacency pairs model typical sequences of communicative acts within a dialogue that include turn-taking like for *question – answer* or *request – accept*. Second, *Discourse Relations* relate *Nodes* which can be adjacency pairs or other discourse relations, thus building up the hierarchical structure of the discourse. Discourse relations are further specialized into *RST Relations* (from Rhetorical Structure Theory) and *Procedural Constructs*.

The *Communicative Act* class represents a single interaction and carries the intention of the communication, e.g., asking a *Question* about the response time. Communicative acts can have many different types according to their communication intention as shown in Figure 1. The importance of explicitly specifying the intention is twofold. Raising the level of abstraction in general contributes to more "natural" specification and thus more efficient design. In our case, we raise the level of abstraction from simple messages (as e.g., in UML sequence diagrams) to communicative acts: questions, requests, offers, etc. In addition, we use the intention encoded in the type of the communicative act for automatic user interface generation [12]. For example, the intention of a *Question* communicative act is information gathering. Thus, input widgets like text areas or input boxes which allow the user to provide information are generated. The intention of the *Informing* communicative act is to provide new unknown facts to the receiver of the communicative act, where the receiver does not need to act upon receiving the communicative act. This can result in text or an image. Automatic generation of user interfaces would be much more difficult with interaction specifications on a lower level of abstraction (e.g., UML sequence diagrams).

Figure 4 shows an example discourse for optimizing the response time of the online store. The interactions depicted in the rounded boxes at the bottom of Figure 4 are cast in terms of communicative acts, e.g., the left-most *Question* for the system response time. The communicative acts in Figure 4 can be viewed as a usage scenario for optimization which advances from left to right. Since there are many sequences of interactions possible, we could think of this example more generally as a *use case*. While use cases carry additional information to sequences of actions, they barely represent more complicated structures.

More importantly, neither scenarios nor use cases represent something like intentions of the various interactions. Since our discourse models are built on communicative acts, they specify the type of communicative act for each interaction. E.g., Figure 4 shows *Questions* and their *Responses*, as well as *Requests* with *Accepts*. This extra piece of information carries the intent of such an interaction.

In addition, according to Conversation Analysis there are frequently occurring pairs of communicative acts — adjacency pairs. E.g., a *Question* must have a related *Response*, and a *Request* must have a related *Accept* or a *Reject*. Adjacency pairs can contain embedded dialogues like clarification dialogues which may become necessary before a communication party

[2] At the time of this writing, the specification of UML is available at http://www.omg.org.

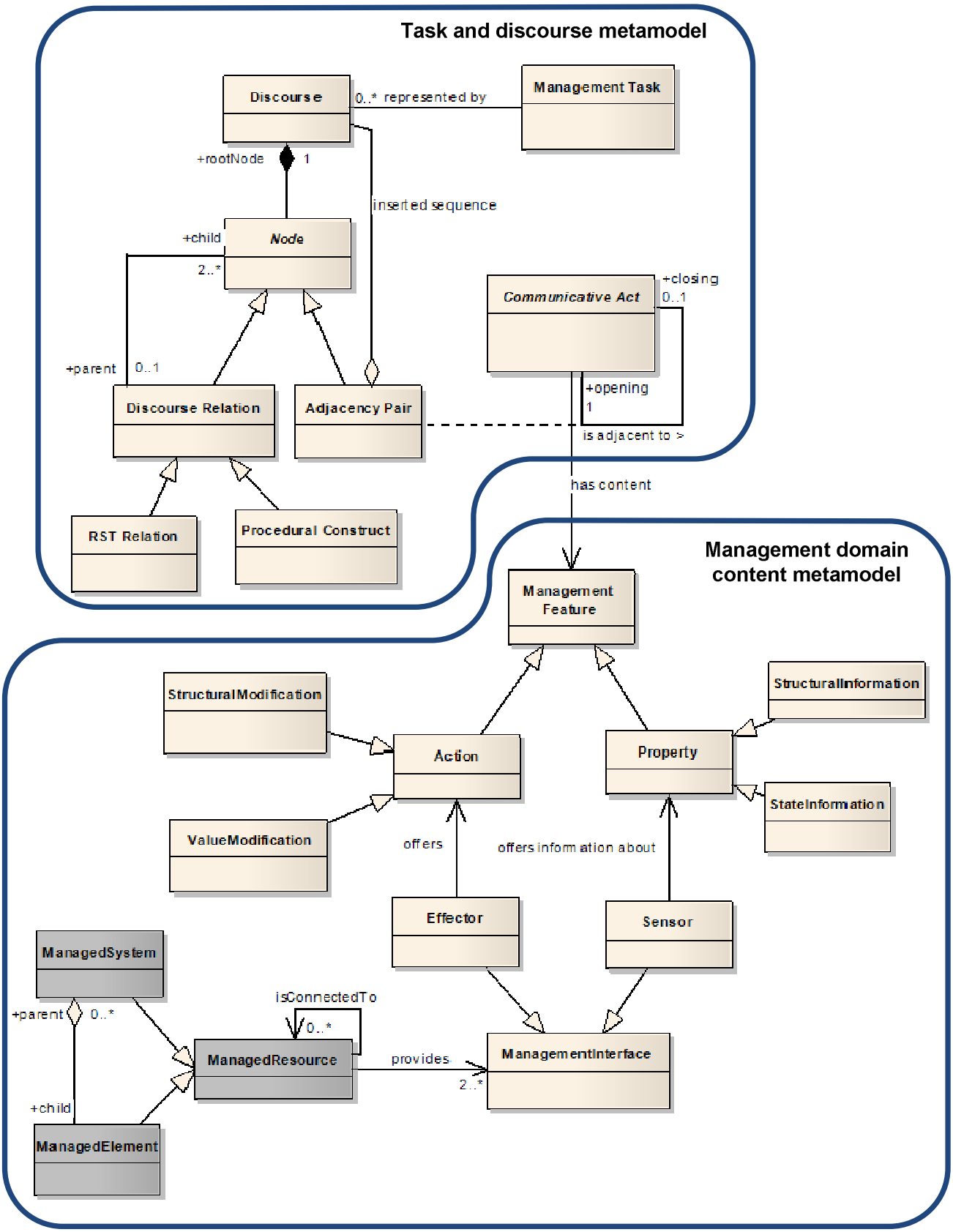

Fig. 3. Task and discourse metamodel connected with management domain content metamodel.

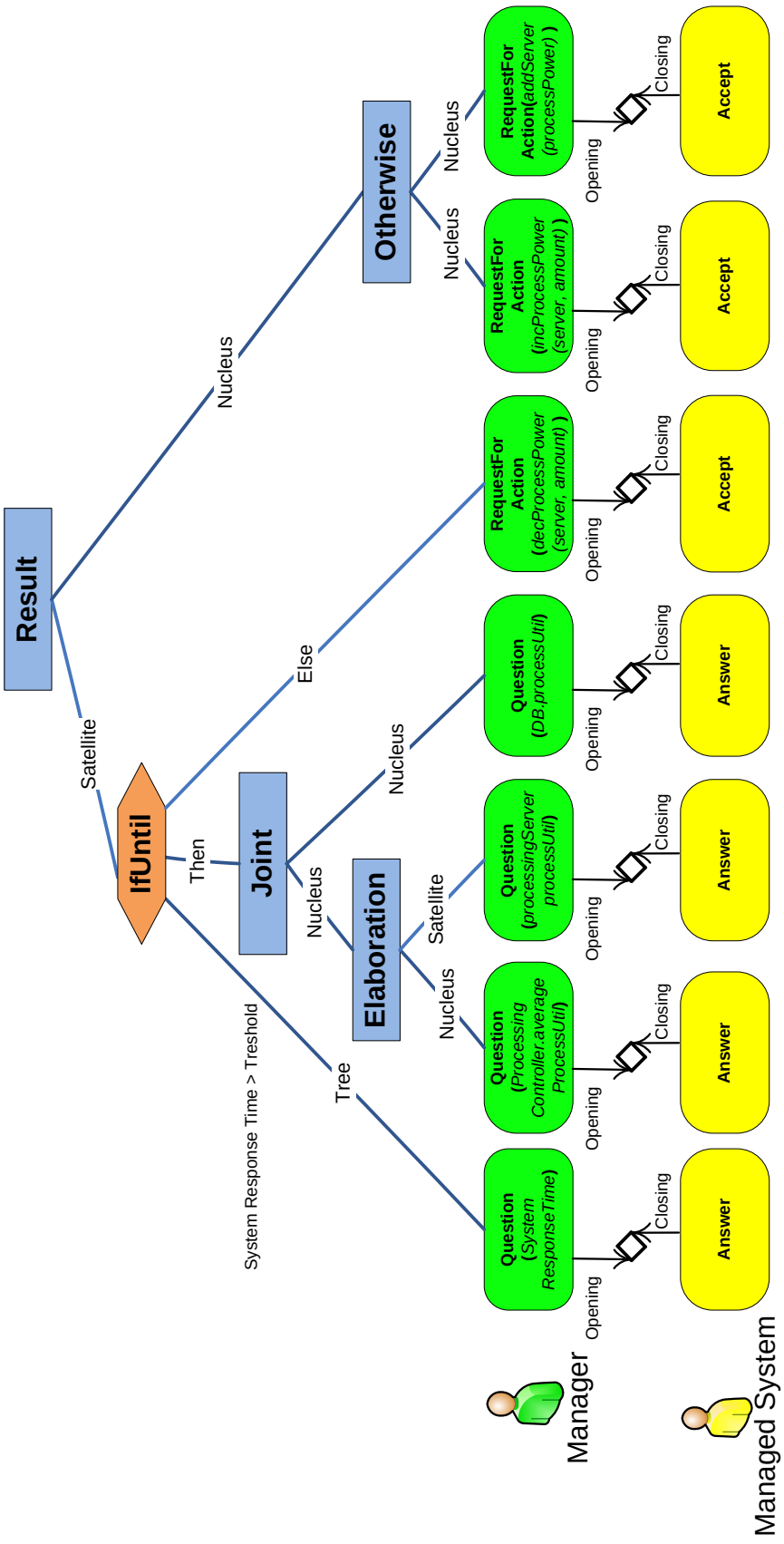

Fig. 4. Discourse for optimizing the response time.

is able to answer a question, for example. Thus, adjacency pairs are modeled in our metamodel as association classes. In our example, adjacency pairs are graphically represented by diamonds. They connect the Opening communicative act with the Closing one.

As stated above, *Discourse Relations* relate adjacency pairs and further structures made up of Discourse Relations. They are specialized into *RST Relations* and *Procedural Constructs*.

All *RST relation*s used in our approach describe a subject-matter relationship between the branches they relate. They usually do not determine any particular execution order but eventually may suggest one. We use two types of RST relations: symmetric (multi-nuclear) and asymmetric (nucleus-satellite) relations. Multi-nuclear relations like *Joint* link equal discourse trees. In our example, the *Joint* relation links the Elaboration with questions about the utilization in the processing tier and a question about the utilization in the database. Note that the order of these actions is not specified by the *Joint* relation; that is why in this particular example also another scenario would fit in, where, e.g., first the database and then the processing tier is asked for its utilization. If it is possible, these questions could be asked even concurrently.

Nucleus-satellite relations link a discourse tree that represents the main intention and a discourse tree that supports the nucleus. For example, the *Elaboration* states that the satellite branch elaborates the dialogue executed in the nucleus branch. In Figure 4, asking the questions about the utilization of each server within the processing tier is the elaboration of the question about the average utilization in the processing tier (controller).

In addition to RST Relations, it turned out to be useful to be able to prescribe particular sequences and repetitions eventually based on the evaluation of some conditions. RST relations are not sufficient for this purpose and, therefore, we have introduced *Procedural Constructs* into our tree structure.

Procedural Constructs provide means to express a particular order between branches of the discourse tree, to specify repetition of a branch and to specify conditional execution of different branches. Thus, our procedural constructs add control structures to our discourse trees that are more complex than usual *if-then-else* or *repeat-until* constructs in typical procedural programming languages. When operationalizing the discourse tree, these procedural constructs also determine which information cannot be presented together on one screen of a graphical user interface. One such construct is *IfUntil*. E.g., in our example the *IfUntil* relation requires information about the server response time, and the execution continues only if the server response time exceeds some defined threshold.

Figure 4 represents a discourse for optimizing the response time for our running example and depicts such a tree structure of the discourse. It illustrates how all interactions within the discourse conceptually belong together as a whole. This structure is composed from Discourse Relations where RST relations are shown in boxes and the *IfUntil* procedural construct in a hexagon.

In our example, there is the relation called *Result* at the top. It represents that the actions requesting the increase of the processing power of a server *or* adding a new server to the system, are a consequence of the situation *resulting* from the preceding interactions. These preceding interactions are subordinated to the *IfUntil* procedural construct. After the Question about the system response time has been asked, the manager decides if the *Tree* branch has to be repeated, or either the *Then* or the *Else* branch will be executed. As stated above, a *Joint* relation (here within the Then branch) does not prescribe the order of execution and allows concurrency. The *Elaboration* relation relates the communication about the general properties and their details. A more formal specification of the relations is given below in Section V.

Such a tree of Discourse Relations could be viewed as the design rationale of the interactions. Alternatively, it can be viewed as a "plan" structure of the discourse for arriving at some goal. In this view, it is actually a non-linear plan (see the *Joint* relation in this example), while the usage scenarios are related linear plans. It is important to note that, where the discourse model represents a generic set of possible discourses, the concrete discourse flow will be controlled by the (human or autonomic) manager.

B. Management Domain Content Metamodel

Besides representing the communication flow explained above, a complete communication representation includes the representation of the communication content as well. The lower part of Figure 3 shows the part of the conceptual metamodel which describes the discourse content for managing software systems.

There are two different kinds of information to be exchanged between a software system and its (autonomic or human) manager:

- the information about current system *properties* and
- the *actions* to be executed by the system as requested by the manager.

Each communicative act is associated with its content: system properties and actions. *Properties* and *Actions* are generalized into the *Management Feature* concept in our metamodel. We distinguish between two types of properties: *StateInformation* and *StructuralInformation*. *StateInformation* represents the managed resource as seen from outside by its parameters (black box). *StructuralInformation* carries the information about the runtime architecture and structure of the system. Analogously, two types of Actions exist: *StructuralModification* and *ValueModification* for modifying the runtime architecture of the system or some of its properties.

Properties and Actions make up the *Sensor* and *Effector* interfaces, respectively, and are generalized into the *ManagementInterface* concept. Each of the *Managed Resources* is related to such management interfaces. Usually, a managed resource will contain one *Effector* and one *Sensor* interface, each containing several system actions and properties. However, a managed resource could have several of them, if interfaces group properties and actions according to some criteria (e.g.,

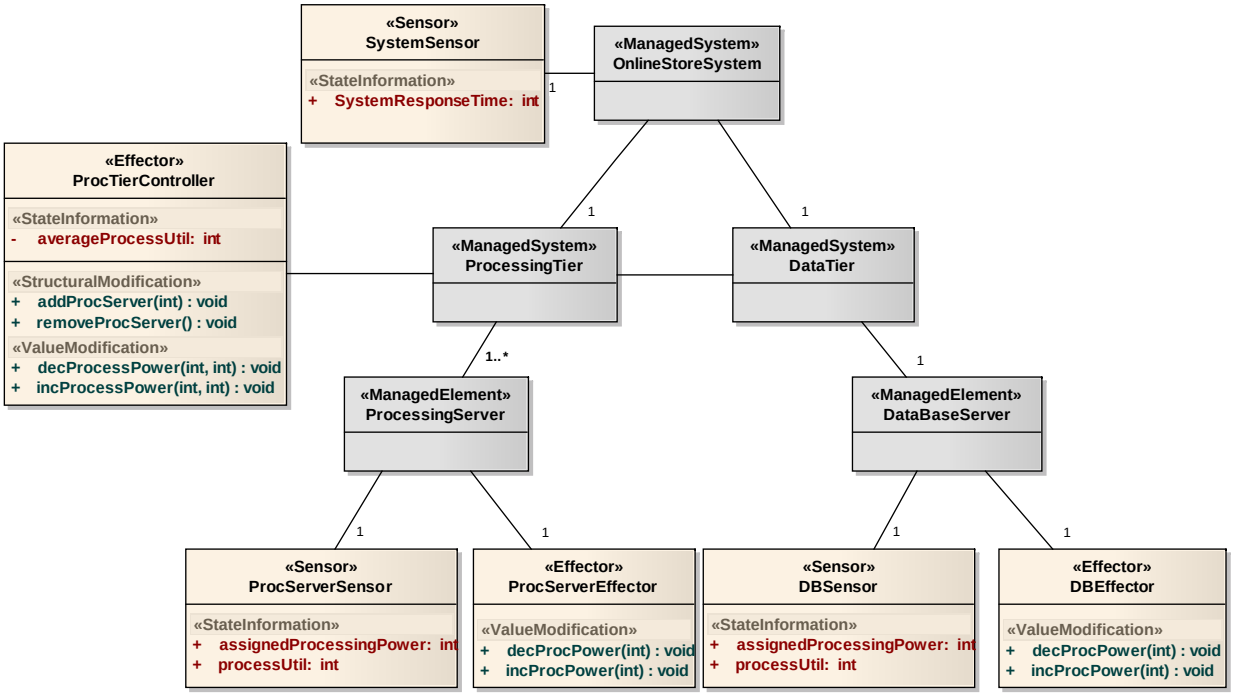

Fig. 5. Content representation for our running example.

one sensor containing only performance and another only security properties).

These management interfaces are provided by the *Managed Resource*, which is specialized into *Managed Element* and *Managed System*. Managed systems can basically contain other managed resources, thus allowing one to build a hierarchical structure of a managed system. These three concepts (dark gray classes in the metamodel in Figure 3) enable the modeling of the system structure, so that the manager and the managed system can communicate about it. The runtime instantiation of this model can be also used by the autonomic manager for reasoning. Structural properties and actions for changing the system structure are only possible for managed systems and not for managed elements.

Figure 5 shows an instantiation of this part of the metamodel for our running example and represents the model of the system architecture. The figure shows the model as a UML class diagram, where classes, methods and attributes are assigned with stereotypes corresponding to the metamodel. The gray classes represent the system structure of the *OnlineStoreSystem*, including two tiers and servers within these tiers. The pink classes represent the management interfaces. For example, *SystemSensor* is a management interface for getting the server response time and *DBEffector* for increasing and decreasing the processing power of the database server. Runtime architecture would be an instantiation of this model, where we would, for example, have several instances of the processing server.

V. Procedural Semantics

An important issue is to operationalize our communication and discourse specifications and to make them executable within our architecture. In order to achieve this operationalization, we transform the communication specifications in the form of discourses into state machines. In this sense, the transformation to state machines defines procedural semantics of our discourse models. For this transformation, the intentions of the communicative acts are not used, but they are not lost, since they are used for other purposes.

In many real-world applications, predetermined discourses are well suited for communication between human and computer, since the user can anticipate the behavior of the system and can expect the same user interface whenever starting the discourse with the system again. This behavior can be achieved by using state machines for the discourse management. Many approaches directly utilize some kind of state machine to model communication. These include both human-machine communication and user interfaces required for it (e.g., in [12][13]), as well as machine-machine communication (e.g., in [14]). Contrary to these approaches, we do not use state machines for explicit communication modeling, but solely for the specification of their procedural semantics and for their execution. Importantly, we do not let the user model them.

Since, especially in user interfaces, the possible interactions are manifold, a flat state machine can become quite complicated. A solution for reducing the complexity of state machines are statecharts, which introduce hierarchies into state machines. Since our discourse models are hierarchical as well,

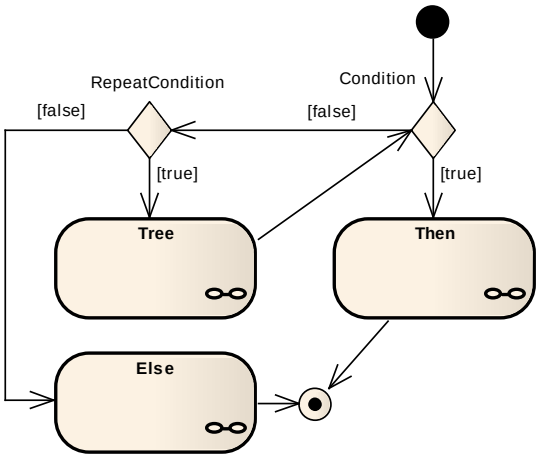

Fig. 6. Mapping of the *IfUntil* procedural construct.

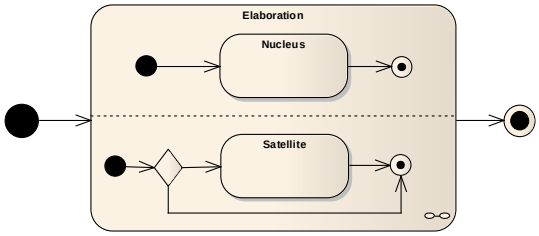

Fig. 7. Mapping of the *Elaboration* RST Relation.

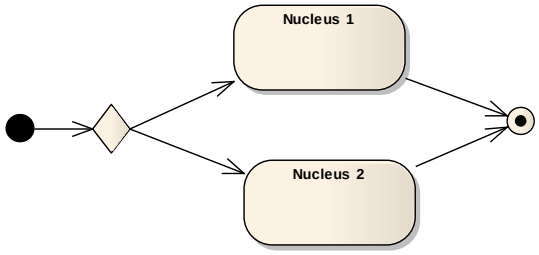

Fig. 8. Mapping of the *Otherwise* RST Relation.

we use statecharts to specify the procedural semantics of the discourse relations and thus the dynamics of the complete discourse.

Such statecharts have the following basic structure:

- Each state transition corresponds to exactly one communicative act and thus represents the advancement in the dialogue.
- State transitions are triggered by sending a communicative act by either the manager or by the managed system.
- Each state entry gets processed, resulting in system effects and in possible triggering of a new communicative act.
- Adjacency pairs are reflected in the statechart by a sequence of transitions. Thus, they constrain the set of potential communicative acts of outgoing transitions of the current and adjacent states.
- Discourse relations are mapped to state machine patterns forming a submachine state that can be included in higher-level discourse relation mappings.

In the following, we specify the procedural semantics of the discourse relations used in our running example.

The statechart of our *IfUntil* [11] relation is shown in Figure 6. *IfUntil* is a procedural construct that we found useful for defining a certain control structure in our task and discourse models. It is more complex than the usual procedural statement in a typical procedural programming language. In fact, it may be considered a combination of an if-statement and a conditional loop. If Condition is fulfilled in the statechart, the discourse continues in the Then branch.

Otherwise, there are two possibilities:

1) the Tree branch is performed if the Condition is not fulfilled. It can be performed again and again until Condition becomes fulfilled, or
2) the discourse can continue in the Else branch, if RepeatCondition is also not fulfilled.

This branching of the flow is modeled by the UML "choice" construct — graphically represented by a diamond. Checking, whether Condition and RepeatCondition are fulfilled is performed by the (autonomic or human) manager.

The *Elaboration* RST relation states that the satellite branch elaborates on the communication executed in the nucleus branch. The procedural semantics are defined in the statechart shown in Figure 7. Communication in the satellite branch is optional — the communication in the nucleus branch does not have to be elaborated. However, if the communication in the nucleus branch gets elaborated, it can even occur in parallel. This is decided usually by the manager for requesting additional information about some parameter. To complete the execution of the relation, both, nucleus and satellite have to be completed.

If the communication flow requires that in one particular moment either the one *or* the other branch of the discourse has to be performed, we use the *Otherwise* RST relation. Usually the manager decides which one of the nuclei (even choosing out of several ones) has to be performed. After one branch is started, it also has to be completed in order to complete the relation as a whole. The procedural semantics is shown in the statechart in Figure 8.

The *Result* RST relation represents that the communication in the nuclei is a consequence of the situation *resulting* from the communication taken place in the satellite. Its simple statechart is shown in Figure 9. It can be used for improving the rendering of the management user interface.[3]

The *Joint* RST relation is a multi-nuclei relation, which doesn't prescribe any order of the execution of the communication in its nuclei. The communication within nuclei can even be performed concurrently, and the relation is completed after all nuclei are completed. This possible concurrency is shown in its statechart in Figure 10 by two compartments separated by a dashed line.

[3]We have slightly changed its semantics with respect to [1].

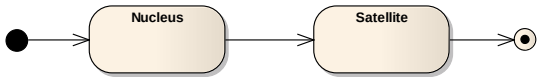

Fig. 9. Mapping of the *Result* RST Relation.

For operationalizing the complete discourse model, the statecharts of each relation have to be combined into one hierarchical statechart according to the discourse relation hierarchy in the discourse model. This is done by traversing the tree structure recursively and applying statechart mappings of the corresponding discourse relations. Typically, the statechart of one discourse relation is included as a submachine state in the statechart of the higher-level statechart. Therefore, the hierarchy of the overall statechart corresponds to the hierarchy of the discourse tree. The result of this traversing for our running example is presented in Figure 11. It completely defines the set of possible discourse flows.

Starting at the top of our example discourse in Figure 4, we have the *Result* RST relation. It basically defines two subsequent statecharts of the relations *IfUntil* and *Otherwise*. The *IfUntil* relation contains three branches: Tree, Then and Else. The Tree and Else branches don't contain any further discourse relations. Both of them contain one adjacency pair each, which are also mapped to the (simple) statecharts. The Tree branch contains the adjacency pair for acquiring system response time. The adjacency pair is mapped to the statechart containing two states: S1 and S2. The transitions within this statechart are induced by the communicative acts: "M: Question (System Response Time)" uttered by the Manager (**M**:) and "S: Answer" uttered by the System (**S**:). The Then branch contains the Joint relation, which again contains an Elaboration relation in one of its nuclei. The *Otherwise* relation doesn't contain any further discourse relations in its nuclei and the mapping defined above is applied, where each nucleus contains one adjacency pair. Prior to each state transition, either the manager or the managed system is supposed to fill in the content of the communicative act. For the decision points (e.g., in *IfUntil* or *Otherwise* relations) the manager is involved and it controls the further discourse flow. Such a complete discourse statechart is interpreted by the Task Execution Engine of our architecture.

VI. Autonomic Architecture and Transition Process

Figure 12 illustrates our proposed autonomic architecture designed to execute and automate modeled tasks as presented above. It is based on the generic autonomic architecture [15], separating the autonomic manager from the managed system.

The *Task Execution Engine* interprets the task and its associated discourse according to the procedural semantics of the discourse relations presented above and manages the flow of the communication. It utilizes also the intention encoded in the type of the communicative acts. For example, for a *Question* it would invoke some information gathering facility of the underlying managed system and for a *Request* it would call corresponding action. The *Human Administrator* or the *Autonomic Manager* controls the autonomic administration process by deciding which set of actions (set of communicative acts) has to be performed next.

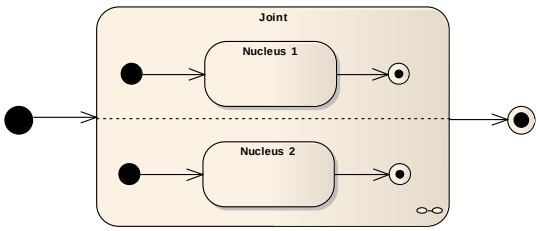

Fig. 10. Mapping of the *Joint* RST Relation.

The *UI Generator* generates and controls the user interface for the case of human management. It utilizes the procedural semantics of the subject-matter relationships of the RST Relations (e.g., Elaboration, Otherwise, etc.) as presented above. In addition, it takes into account the intentions of the communicative acts as mentioned in Section IV-A. The *Communication Adapter* encapsulates low-level interaction interfaces of the managed system.

This architecture enables the whole spectrum of management possibilities. Without an Autonomic Manager first, it provides for high-level communication with the human administrator only, through a generated user interface. In Figure 12(a), the human administrator is responsible for the management Tasks I and II. For example, management Task I could be the performance optimization as in the running example, and Task II could be a (self-)healing task.

After performing the management by humans for some time according to the defined discourse, it is expected that some insights into system behavior will have been gained. With these insights, the Autonomic Manager has to be designed and implemented. Since the discourse constrains possible interactions, this is easier to do than implementing the autonomic manager from scratch. Some of the relations include conditions for their execution as defined above, which have to be evaluated by the Autonomic Manager. It "decides" how the discourse proceeds by initiating the sending of corresponding communicative acts. Depending on the discourse complexity as well as the number and variety of relevant parameters and possible actions, the manager would be more or less complex. However, the design and deployment of autonomic managers is out of the scope of this paper.

VII. Case Study: System Simulation with SimSys

In the first cases study, we simulated an IT system with a tool and modeled one typical management task for it. We concentrated on human management using graphical user interfaces. The used tool has been developed in the course of a research project at IBM [16] and was kindly provided to us for our research. It has been used at IBM for the evaluation of policy-based software system management. This tool can simulate IT systems with different configurations containing

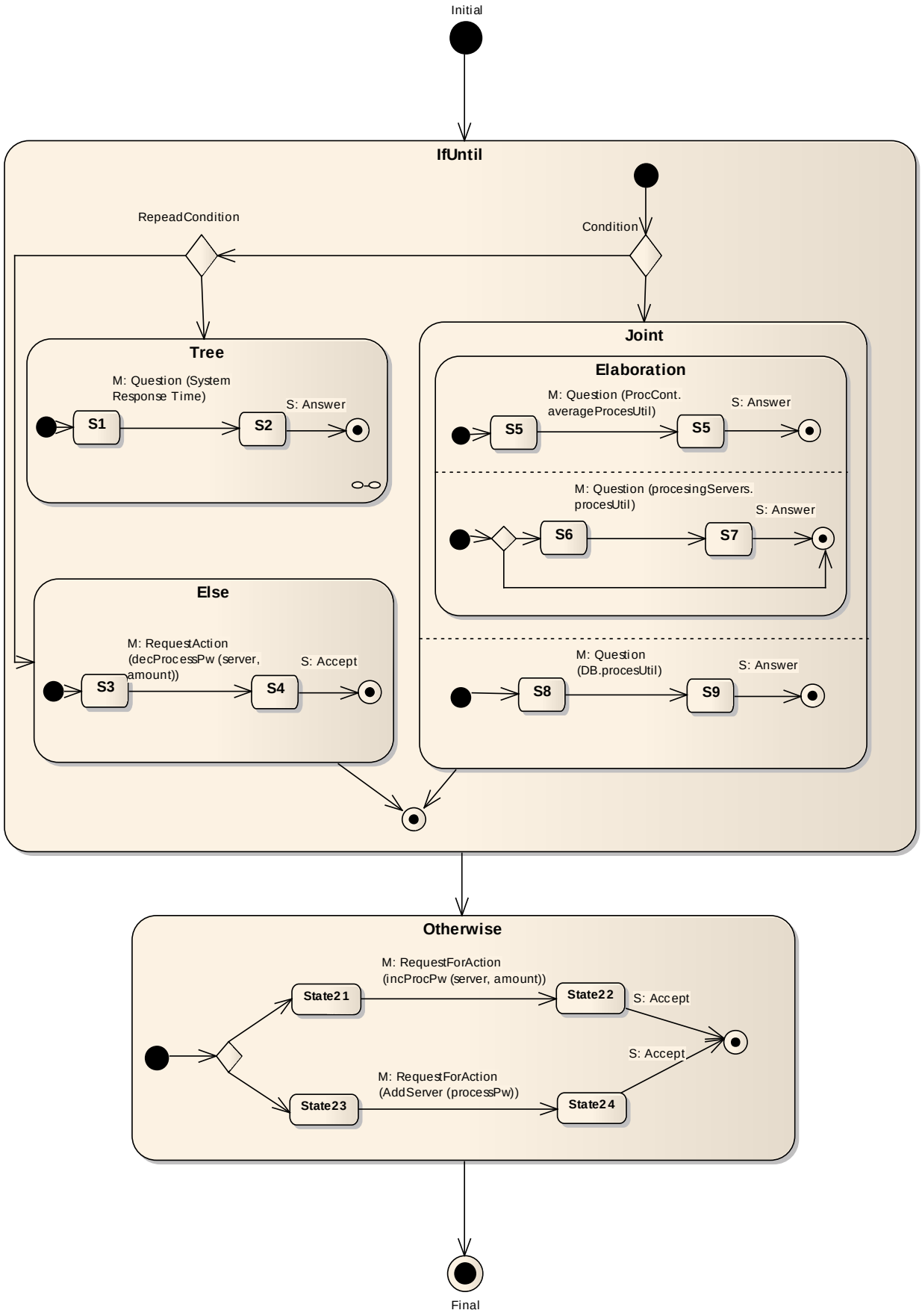

Fig. 11. Complete Discourse Statechart.

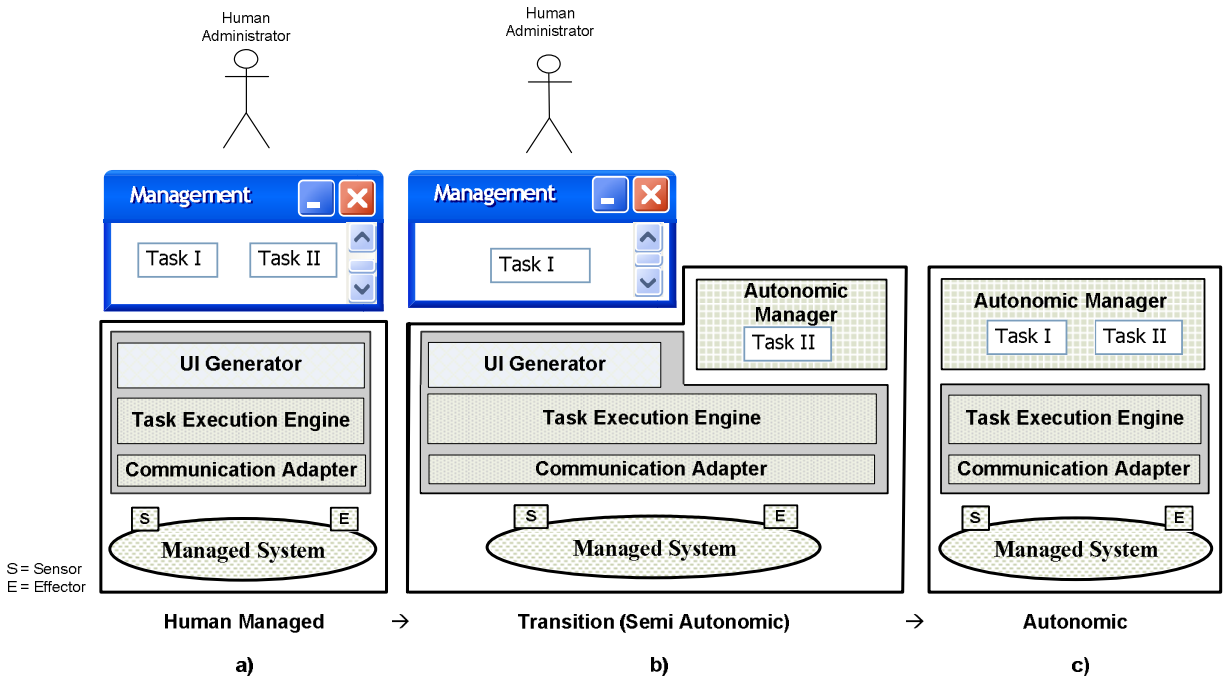

Fig. 12. Autonomic Architecture and Transition Process.

different kinds of servers. It also simulates the workload on such systems. It defines a system as a set of processes in which each process has a set of defined properties and operations. For example, one HTTP server would be represented by one running process. This process would have properties such as CPU power or disk space and would provide operations to change these properties. These processes can also have (incoming or outgoing) connections to others processes. A process can execute the operations on connected processes over these connections. A typical example is the *sendRequest* operation between two servers (processes), which forwards a user request to connected servers.

The simulated system in our case study represents a typical Web store. The simulation tool offers a graphical representation of the simulated system architecture as illustrated in Figure 13. The Web store contains the IT-Infrastructure, which calculated parameters, e.g., response times, dropped requests, etc. Within the IT-Infrastructure, there are three server clusters in charge for distributing HTTP requests, application server requests, and database requests, as common in three-tier architectures. Within each server cluster, servers are responsible for request processing and, if needed, their forwarding. The Shopper process created a sample workload on the system generating *shopper* objects, where each shopper created series of requests to the system. When a request entered the site, the SimSys simulation system time-stamped it for the response time recording and sent it to the HTTP cluster, which forwarded the request to the first ready HTTP server and further via the application cluster to the database. We attached our communication platform to the simulated system via Java method calls.

Let us explain the management discourse shown in Figure 14. Much as in our previous examples, the most significant parameter is the response time (in the SimSys system identified as *latency*). So, the manager can ask a question about the system latency. If the manager considers the latency too large, the manager can ask for the current user activity. As a *result*, the power of the servers can be increased (e.g., if only the system latency has risen) or additional servers can be added (e.g., if both the system latency and the user activity have risen).

In this example, we show a case where a human manager is responsible for the management and communicates with the system using the generated user interface. Figure 15 shows screenshots of this simple management user interface. First, the manager has the possibility to ask a question about the system latency using a button from Screen 1. This represents the *Tree* branch of the *IfUntil* relation. The answer to this question is presented in Screen 2. In addition, it is possible to ask about the user activity (by selecting the button on the right in Screen 2). In this case, the human manager "evaluates'" the condition of the IfUntil relation. If the manager decides that the latency is too high (in our case 6 seconds), the condition in the IfUntil relation is fulfilled and the manager should be able to continue with the communication (asking the second question by pressing the *getActivity* button). Screen 3 shows the answer to this question and two buttons, *addProcessingPower* and *addServer* representing the possibilities to issue corresponding requests.

This example shows a simple management scenario for the human management case. It demonstrates the feasibility of our approach to handle the required communication (model

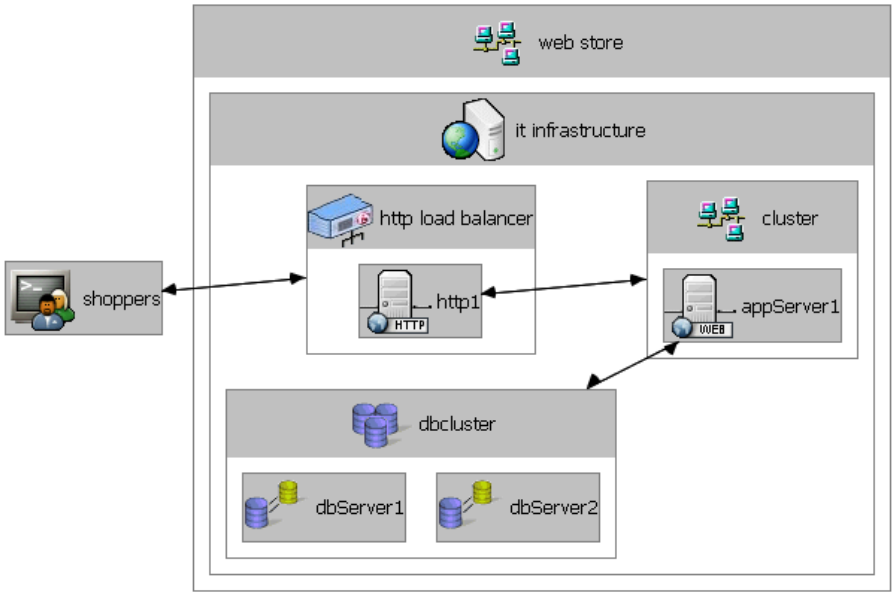

Fig. 13. Simulated System.

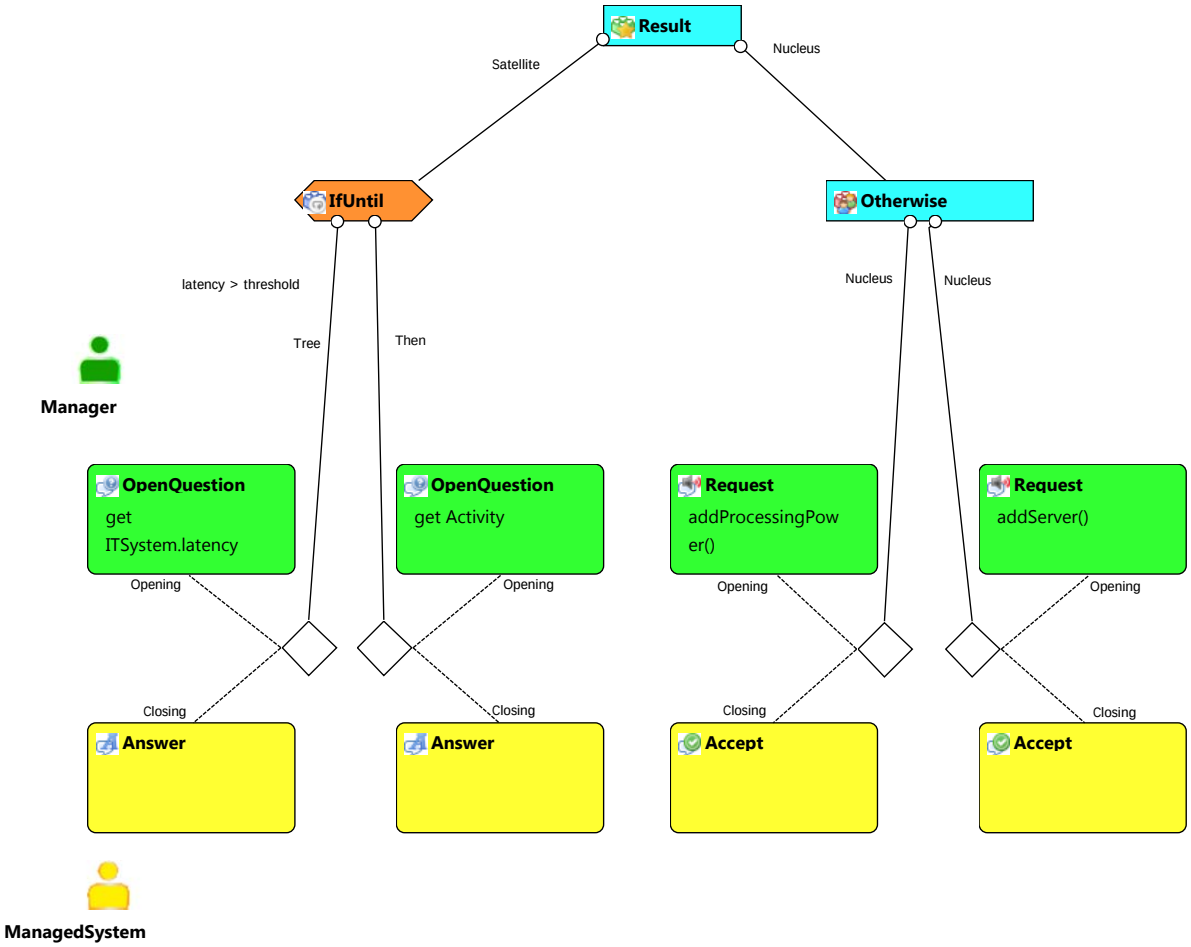

Fig. 14. System Optimizing Discourse Specification.

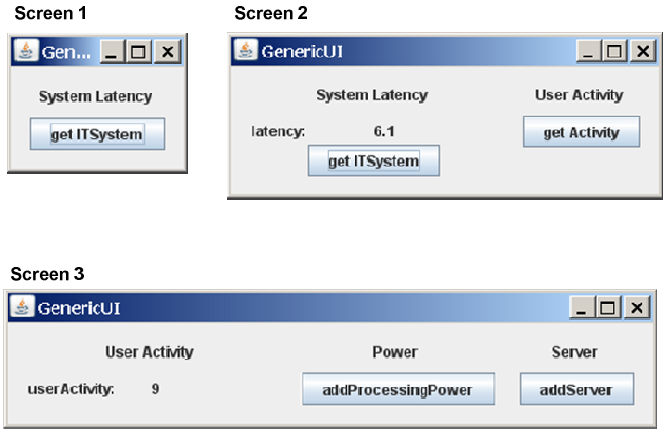

Fig. 15. Screenshots of the Management User Interface.

and execution) between the system to be managed (in this case a simulated three-tier business application) and the human manager.

VIII. Case Study: Application Server JBoss

This case study demonstrates our approach on the management of a Java application server. In this second case study, we utilized the Java application server JBoss and implemented two management tasks in an autonomic manner.

An application server is a software system which usually takes the role of the "business logic" in the multi-tier architecture. Its main purpose is to encapsulate the access to the data in the database and to simplify data manipulation along the lines of the business process. Usually, an application server utilizes some component model and enables the development of component-based applications. It acts also as a container for these components. An application server offers different features to simplify application development. These include a programming model as a set of APIs, resource handling and pooling, support for distributed computing, authentication and authorization, messaging, transactions, monitoring and control, etc.

In our case study we use the JBoss server, which implements the J2EE industry standard[4] and enables the development of Java components — Enterprise Java Beans (EJBs). For extending JBoss, new components (services) were developed according to the Managed Beans (MBeans) service specification. To be able to perform the case study, especially to be able to monitor required parameters, these were the following extensions:

- a CPU and Memory Monitor, which collects data from the underlying Java Virtual Machine (JVM), and
- a Response Time Recorder. This extension is concerned with gathering information about the response times for client requests.

[4]http://java.sun.com/j2ee/docs.html

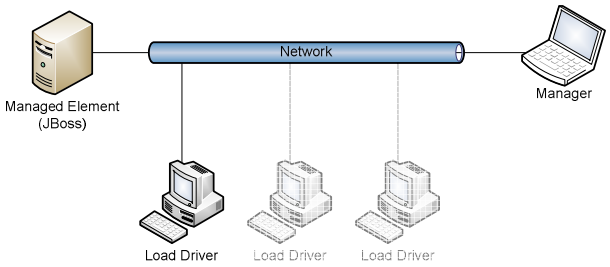

Fig. 16. Case Study Setup.

A. Case Study Setup

The system setup for the case study is shown in Figure 16. It is a distributed setup, with the JBoss application server, the autonomic manager and the load generators installed on different machines. For achieving a more realistic load for the application server, we have used the benchmark tool ECperf [17]. It is designed to measure performance and scalability of J2EE application servers and provides a typical J2EE business application as well as load generators to simulate the workload for such an application.

Since JBoss is a complex software system, which has many parameters that have to be configured, and since the configuration is tedious and error-prone, a self-configuration capability would seem desirable. Therefore, we use the configuration scenario as one example for our case study.

Our second example is about optimization. The optimizing capability is also of interest for a system like JBoss. An optimization in this case would imply the reduction of the response times of client requests sent to the application server. This requires the response times to be measurable and system parameters to be accessible and changeable.

B. Communication Content Model

Prior to creating the management discourses we have to figure out, which parameters are of interest for the monitoring of the JBoss status, as well as to define possible corrective actions which have impact on the behavior of the JBoss server. In essence, we have to define what the manager and the managed server (JBoss) will communicate about.

The most significant JBoss properties for this case study are the following:

- available memory,
- the size of the database connection pool,
- the thread pool size for the EJB invocation processing threads, and
- the server's response time.

The most relevant actions are:

- setting the thread pool size, and
- clearing the EJB cache

Restrictions to the Java Virtual Machine (JVM) heap memory size would also have a deteriorating impact on system performance. Unfortunately, the maximum heap size cannot be changed during runtime, and minimum and maximum values are specified as startup parameters. A modification of these

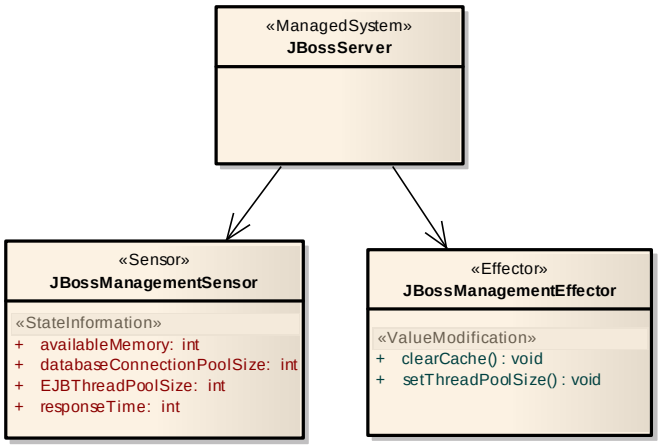

Fig. 17. JBoss Server model for the domain of discourse.

values would thus require a restart of the JVM and the system would be completely out of service for some time, including all applications running within that JVM instance. Therefore, this parameter has not been used in our case study. Also some other parameters have not been included in the case study (SQL statement cache, EJB cache maximum size, etc.).

To use JBoss parameters for the content of communicative acts, we have modeled them as shown in Figure 17. Since we do not model the internal structure of JBoss and are more interested in illustrating our approach to communication, the model is rather simple and contains only one class representing the JBoss server and associated Sensor and Effector interfaces.

C. Configuration Task of the JBoss Server

This example shows the configuration task for the JBoss application server. For a better understanding of this example, let us briefly explain the thread pooling in the JBoss server.

Figure 18 shows the interworking of thread pools. Every request that arrives at JBoss via the Tomcat[5] — embedded server for servlets — has to wait for a thread from the HTTP thread pool to be available in order to be processed. If the client request involves calls to an EJB on the server, the request also has to acquire a thread from the EJB thread pool. And if the EJB call during its execution needs to make a request to the database, a connection from the database connection pool has to be retrieved. The amount of threads and connections in the different pools limits the number of client requests that can be processed concurrently. Having more connections in the database connection pool than threads in the EJB pool is a configuration that is not useful and unnecessarily increases resource usage, because the number of database connections that can be used at the same time is limited by the number of threads in the EJB pool. Thus, it is not desirable to have more connections in the DB connection pool than threads in the EJB thread pool. If the ratio of EJB threads to database connections becomes too small, many client requests will not get a database connection within the configured timeout. For

[5]tomcat.apache.org

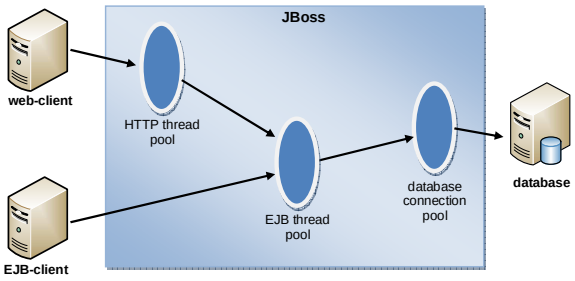

Fig. 18. JBoss threads pooling.

our scenario, we keep the number of database connections constant. The goal is to maintain the number of pooled EJB threads above the number of available database connections. Another goal of this scenario is to prevent an extensive use of memory. The amount of available memory is monitored and when that amount drops below the configured minimum threshold, the memory is freed by:

- ordering JBoss to clear up all EJB caches, and by
- reducing the amount of concurrency in request processing by reducing the number of EJB worker threads.

The discourse for this configuration task is shown in Figure 19. Starting from the top of the diagram, the communication for the configuration of the thread pool sizes and the communication for the memory usage managing can be performed in parallel — jointly — as defined by the RST Relation *Joint*. For the configuration of the thread pool sizes the Manager issues *Questions* about the *databaseConnectionPoolSize* and *EJBThreadPoolSize*. If the ratio is not below a given threshold, the manager *requests* the action for setting up the appropriate threadPoolSize. For managing of memory usage, the manager issues the *Question* about the *availableMemory*, and when it drops below some threshold it tries to free it by issuing the *Requests* to *clearCache* and to *reduceThreadPoolSize* of the worker threads.

For the evaluation of the transition towards autonomic systems, an Autonomic Manager has been implemented. It periodically executes the task and discourse in order to perform management functionality. Regarding the thread pooling, it corrects the parameters using the procedure described above. For the memory usage management, we had to limit the maximum amount of memory used by the Java Virtual Machine to 100MB for getting observable effects. As stated previously, the ECperf benchmarking tool was used for load creation. The Autonomic Manager checks the memory status according to the discourse and executes corrective actions whenever needed. Under the same load conditions, JBoss crashed due to *OutOfMemory* exception when the Autonomic Manager has been put out of function.

D. Optimization Task of the JBoss Server

Our second example is the response time optimization task. The most significant parameter having an effect on the response time is EJB thread pool size [18]. The manager asks the *Question* about the current response time and *Requests* the

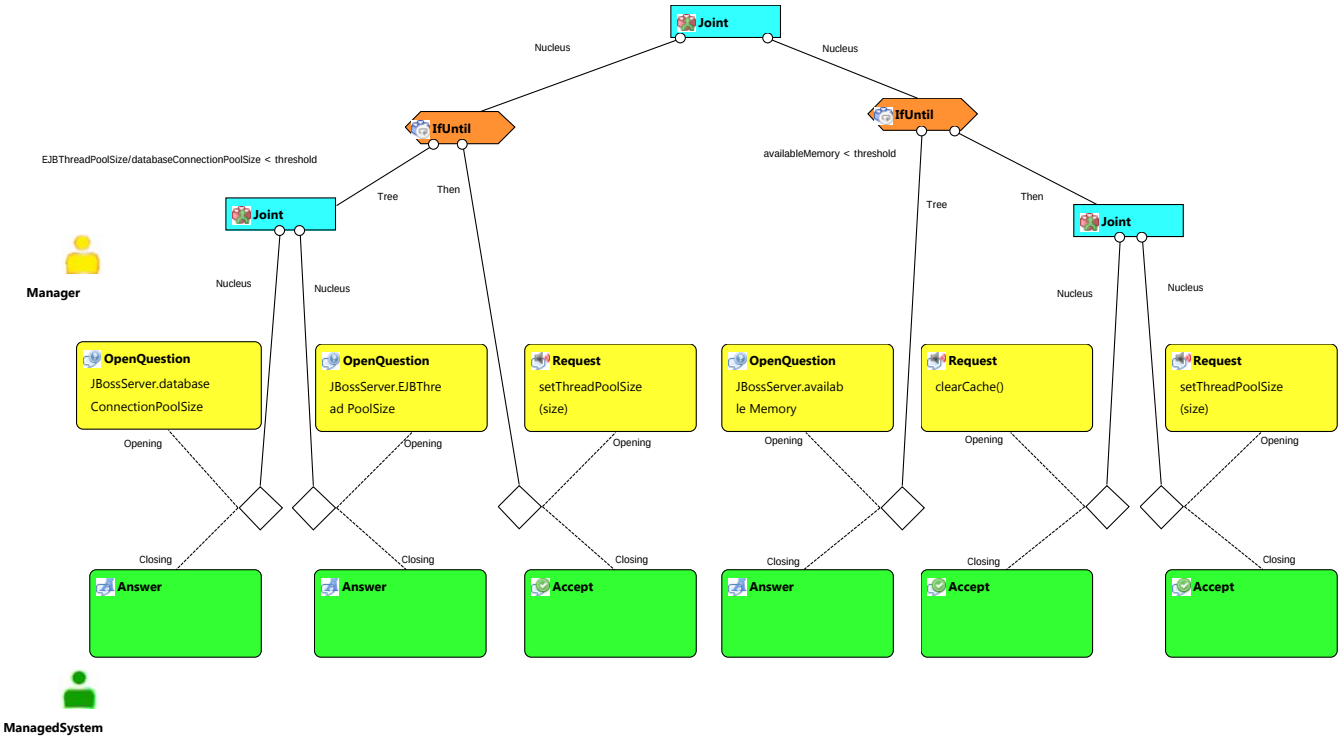

Fig. 19. JBoss configuration discourse.

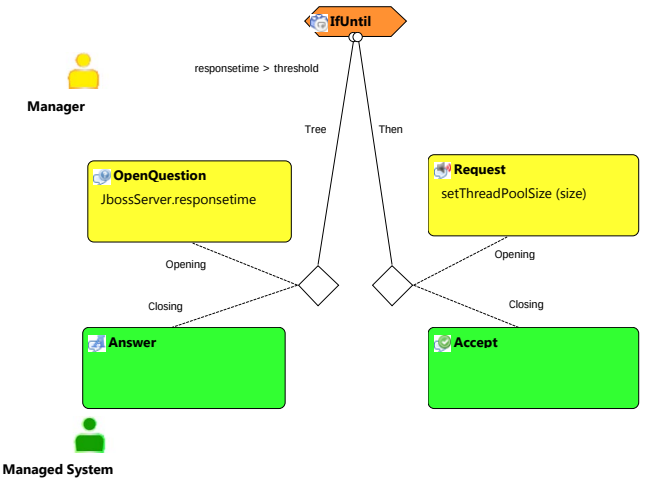

Fig. 20. JBoss optimization discourse.

action for changing the thread pool size, when the response time is below a given limit. Figure 20 shows the corresponding discourse. The communication in this discourse is simple and so are the resulting user interfaces. However, for showing the transition towards autonomic systems, an Autonomic Manager has been implemented as well.

The Autonomic Manager monitors the response time and makes small changes to the thread pool size according to the discourse. Once the response time has improved, it takes the new values of both the thread pool size and response time as a condition for the next execution of the optimization discourse. If and when the modifications have increased the response time, the autonomic manager tries to adjust the parameter in the opposite direction for the next execution of the optimization discourse. We have clearly seen effects of increasing the thread pool size for reducing the response times. However, since more threads use also more memory, this value cannot be increased indefinitely. In our case it settled around 80. We also observed that clearing the EJB caches temporarily increased the CPU load and thus the response times.

IX. Related Work

Our work relates both to the field of interaction modeling (between humans and computers as well as between computers) and to the field of autonomic and self-managed software systems.

Modeling interaction design is mostly done through techniques from task analysis and cognitive science. Techniques based on Hierarchical Task Analysis [19] or GOMS [20] model activities on various levels of detail in a hierarchical way to achieve a particular goal, and (e.g., temporal) relationships between tasks on the same level. On the more detailed levels, task models specify only the type of tasks (e.g., user, system or interaction task) or operators (click, select ...), but not their intention in the sense of asking, requesting, etc.

Formal interaction modeling is important for interactions between agents. Most approaches for modeling inter-agent communication utilize some form of finite-state machinery. E.g., Labrou and Finin [21] deal with interactions between

agents based on KQML, where *conversation policies* are proposed for the description of conversations between agents. Conversation policies represent simple conversations between agents in terms of possible sequences of KQML messages. Our discourse models can represent more complex interactions and should be easier to design by humans.

Management tasks are nowadays performed by well-trained professionals which are responsible for *configuring* the system so that the users can get their jobs done and for *maintaining* the system against both internal failures and internal or external attacks [22]. They interact with the system using command-line interfaces, graphical interfaces or Web-based management tools [23].

Modeling and specifying management tasks and user interfaces for performing those tasks has been neglected in general [24]. However, operators and administrators of software systems are constantly trying to automate administrative tasks and to reduce unnecessary interactions with the system. They use their own executable scripts to automate monitoring of system health, to perform operations on a large number of systems, and to try to eliminate errors on common tasks that take many steps [25]. Our approach also strives for the automation of administrative tasks but concentrates on interactions within such tasks and provides a well-defined way for their modeling.

A typical approach to define automation of software management tasks for autonomic computing is in terms of *policies*. Policies represent instructions to determine the most appropriate activity in a given situation. One way to specify policies has been defined in [26]. They represent policies in the form "`IF condition THEN action`" where *condition* contains a particular state of the system and the *action* represents the actions to be performed if such a state occurs. They also define how to manage and execute such policies. Kephart and Walsh [27] define three types of policies: Action, Goal and Utility Function policies. Action policies are on the lowest level and take also the `if-then` form. Goal policies specify a single desired state and the system should generate behavior itself from the policy. Utility Function policies generalize Goal policies where a desired state is computed by selecting from the present collection of feasible states the one that has the highest utility. The Accord framework [28] defines so-called operational interfaces. This offers the possibility to formulate, inject, and manage rules that are used to manage the runtime behaviors of the autonomic system. Rules incorporate also typical if–then expressions, i.e., "IF condition THEN actions". Similarly to our approach, Cheng et al [29] consider that *"the capturing and representation of human expertise in a form executable by a computer"* is crucial for the automation of management tasks. They have developed a new language for adaptation where the concepts used in the language are derived from system administration tasks. The basic concept in the language is a *tactic*, which embodies a small sequence of actions to fix a specific problem in a localized part of the system. A tactic contains the conditions of applicability, a sequence of actions, and a set of intended effects after execution.

Contrary to this work on policies, our approach focuses on and formalizes interactions between an administrator and a software system. We believe that the interactions are important both for the task execution and for understanding the task. It seems also non-trivial to reduce sometimes complex management tasks to single policies. We believe that our task models should be easier to create by humans since they are based on human communication theories.

X. Discussion

In order to utilize our approach, the system designers and developers would first have to attach our communication platform to the system — to integrate it into the managed system. If the system is designed from scratch, special interfaces would have to be developed. If it is a legacy system, the designers would have to understand and expose the interfaces. This is usually not trivial for such systems. In any case, some additional effort would have to be provided. This may seem to not pay off, especially if the transition towards autonomic operation is far away. However, we believe that our approach can bring some advantages even in the case of (only) human management. By performing task and discourse modeling, the designers become familiar with the system's behavior. Very often, the automation of management tasks is becoming possible only through such an improved understanding.

Our approach adds also some additional overhead to the communication through the need to convert the low-level communication messages into communicative acts. This is more significant for the autonomic management case, where the autonomic manager reacts on the system's events and enacts the corrective actions (possibly after some additional communication with the managed system for problem investigation). Our approach is more directed towards business applications and information systems, where usually the "best-effort" for correcting the problem is sufficient. Anyhow, even this would be much faster than the human reaction. However, due to this overhead, our approach is not well investigated to be used for real-time or embedded systems.

The IT industry has neglected tools and systems used by operators for configuration, monitoring, diagnosis, and repair, and the need for improved user interfaces for operators is large [30]. We believe that the discourse-based communication modeling of administrative (management) tasks, as well as systems modeling for the communication content can contribute to a better understanding of management procedures in general.

XI. Conclusion

In essence, we propose to model software-management interactions and tasks in the form of discourses between the administrator and the software system. In addition to the interaction and task models, we have developed a metamodel for the modeling of the management domain for such tasks. For the execution of these tasks we have defined their procedural semantics, and from these models, we are able to generate user interfaces.

Case studies showed that our approach can be used both in the case of human management as well as in the case of autonomic management. However, since two case studies involved two different managed systems (simulation and real system), management discourses were slightly different. For one managed system and one particular management task, we would have the *same* discourse and therefore the same communication specification both for human and autonomic management. This utilizes to the gradual transition towards autonomic systems.

XII. Acknowledgments

The extension of the JBoss server as well as autonomic managers have been programmed by the diploma student Christoph Bernhard Schwarz, who worked together with the authors of this paper. We also thank Eser Kandogan and IBM for providing us with the simulation tool SimSys.

References

[1] E. Arnautovic, H. Kaindl, J. Falb, and R. Popp, "High-level modeling of software-management interactions and tasks for autonomic computing," in *Autonomic and Autonomous Systems, 2008. ICAS 2008. Fourth International Conference on*. Washington, DC, USA: IEEE Computer Society, March 2008, pp. 212–218.

[2] E. Arnautovic, H. Kaindl, J. Falb, R. Popp, and A. Szép, "Gradual Transition towards Autonomic Software Systems based on High-level Communication Specification," in *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing, Autonomic Computing Track*. New York, NY, USA: ACM Press, 2007, pp. 84–89.

[3] J. R. Searle, *Speech Acts: An Essay in the Philosophy of Language*. Cambridge, England: Cambridge University Press, 1969.

[4] M. Nowostawski, D. Carter, S. Cranefield, and M. Purvis, "Communicative acts and interaction protocols in a distributed information system," in *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*. New York, NY, USA: ACM Press, 2003, pp. 1082–1083.

[5] J. Falb, R. Popp, T. Röck, H. Jelinek, E. Arnautovic, and H. Kaindl, "Using communicative acts in high-level specifications of user interfaces for their automated synthesis," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. New York, NY, USA: ACM Press, 2005, pp. 429–430, tool demo paper.

[6] P. Luff, D. Frohlich, and N. Gilbert, *Computers and Conversation*. London, UK: Academic Press, January 1990.

[7] W. C. Mann and S. Thompson, "Rhetorical Structure Theory: Toward a functional theory of text organization," *Text*, vol. 8, no. 3, pp. 243–281, 1988.

[8] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, pp. 1–39, 2008.

[9] J. Falb, H. Kaindl, H. Horacek, C. Bogdan, R. Popp, and E. Arnautovic, "A discourse model for interaction design based on theories of human communication," in *CHI '06 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA: ACM Press, 2006, pp. 754–759.

[10] C. Bogdan, J. Falb, H. Kaindl, S. Kavaldjian, R. Popp, H. Horacek, E. Arnautovic, and A. Szep, "Generating an abstract user interface from a discourse model inspired by human communication," in *Proceedings of the 41th Annual Hawaii International Conference on System Sciences (HICSS-41)*. Piscataway, NJ, USA: IEEE Computer Society Press, January 2008.

[11] R. Popp, J. Falb, E. Arnautovic, H. Kaindl, S. Kavaldjian, D. Ertl, H. Horacek, and C. Bogdan, "Automatic generation of the behavior of a user interface from a high-level discourse model," in *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences (HICSS-42)*. Piscataway, NJ, USA: IEEE Computer Society Press, 2009.

[12] J. Falb, R. Popp, T. Röck, H. Jelinek, E. Arnautovic, and H. Kaindl, "Fully-automatic generation of user interfaces for multiple devices from a high-level model based on communicative acts," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS-40)*. Piscataway, NJ, USA: IEEE Computer Society Press, Jan 2007.

[13] I. Horrocks, *Constructing the User Interface with Statecharts*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] M. McKinlay and Z. Tari, "Dynwes - a dynamic and interoperable protocol for web services," 2002, pp. 74–83.

[15] *An architectural blueprint for autonomic computing*, 3rd ed., IBM Corporation, June 2005, white Paper.

[16] E. Kandogan, C. Campbell, P. Khooshabeh, J. Bailey, and P. Maglio, "Policy-based management of an e-commerce business simulation: An experimental study," *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pp. 33–42, 13-16 June 2006.

[17] S. M. Inc., "Ecperf (tm) specification," Sun Microsystems Inc., 2002.

[18] Y. Zhang, W. Qu, and A. Liu, "Adaptive self-configuration architecture for j2ee-based middleware systems," *System Sciences, 2006. HICSS '06. Proceedings of the 39th Annual Hawaii International Conference on*, vol. 9, pp. 213a–213a, Jan. 2006.

[19] Q. Limbourg and J. Vanderdonckt, "Comparing task models for user interface design," in *The Handbook of Task Analysis for Human-Computer Interaction*, D. Diaper and N. Stanton, Eds. Mahwah, NJ, USA: Lawrence Erlbaum Associates, 2003, ch. 6.

[20] B. E. John and D. E. Kieras, "Using GOMS for user interface design and evaluation: Which technique?" *ACM Trans. Comput.-Hum. Interact.*, vol. 3, no. 4, pp. 287–319, 1996.

[21] Y. Labrou and T. Finin, "Semantics and conversations for an agent communication language," in *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, 1997, pp. 584–591.

[22] E. A. Anderson, "Researching system administration," Ph.D. dissertation, University of California, Berkeley, 2002.

[23] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker, "Field studies of computer system administrators: analysis of system management tools and practices," in *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*. New York, NY, USA: ACM Press, 2004, pp. 388–395.

[24] R. Barrett, Y.-Y. M. Chen, and P. P. Maglio, "System administrators are users, too: designing workspaces for managing internet-scale systems," in *CHI '03: CHI '03 extended abstracts on Human factors in computing systems*. New York, NY, USA: ACM Press, 2003, pp. 1068–1069.

[25] E. Kandogan and J. Bailey, "Usable Autonomic Computing Systems: The Administrator's Perspective," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 18–26.

[26] R. M. Bahati, M. A. Bauer, and E. M. Vieira, "Mapping Policies into Autonomic Management Actions," in *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, 2006, p. 38.

[27] J. O. Kephart and W. E. Walsh, "An Artificial Intelligence Perspective on Autonomic Computing Policies," in *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2004, p. 3.

[28] H. Liu and M. Parashar, "Accord: A Programming Framework for Autonomic Applications," *Systems, Man and Cybernetics, Part C, IEEE Transactions on*, vol. 36, no. 3, pp. 341–352, May 2006.

[29] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based Self-adaptation in the Presence of Multiple Objectives," in *ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Shanghai, China, 21-22 May 2006.

[30] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1.