Autonomous Software

Michael Rovatsos Gerhard Weiss

Institut für Informatik, Technische Universität München
85748 Garching, Germany
{rovatsos,weissg}@in.tum.de

Abstract

Industrial-strength software is reaching a level of inherent complexity which tends to make an effective development, deployment and administration impossible. This has led to a rapidly growing interest in the notion of autonomous software, that is, software which takes over, and encapsulates, action choice and responsibility from its users and operators so that it can handle its complexity on its own. A key condition for the broad acceptance of autonomous software is the availability of a clear notion of autonomy as a software property upon which precise specification schemes for autonomous software systems can be build. There are diverse approaches available in computer science that are useful in this respect. This chapter describes a generic autonomy specification framework which gives an integrated view of these approaches and of the state of the art in specifying autonomy as a software property.

1 Introduction

Advances in information technology and growing expectations on the functionality of computer-based information processing systems form the basis for a fundamental change in the software landscape. Characteristic to this change is the rapidly increasing importance of industrial, commercial and scientific software systems which operate and are tightly embedded in open, distributed, networked, dynamic, and hardly predictable socio-technical environments. Despite the impressive progress achieved in software engineering during the past decades, this kind of software systems tend to possess an extraordinarily high level of inherent complexity which makes it practically impossible to develop, administrate and deploy them effectively in terms of time and costs. This serious problem has led to the much-attended vision of autonomous software, that is, software being able to handle its complexity on its own. The spectrum of primary attributes associated with autonomous software is broad and ranges from self-diagnosing and self-structuring over self-managing and self-governing

to self-repairing and self-adapting.[1] The key idea underlying these attributes is to have software equipped with action and decision choice so that it can fulfil its tasks even under critical and unexpected circumstances (e.g., changes in the technological infrastructure or in the application-specific user demands) *without* requiring human support, feedback or intervention.

Autonomy orientation may be viewed as a natural next step in the evolution of generic software models [18, 35]. In the course of this evolution, the basic building blocks of software – monolithic programs, modules, procedures, objects and components – gained increasing degrees of localisation and encapsulation of data processing and state control. What is common to all traditional building blocks is that their invocation happens through *external* events, such as start directives by users and call statements or messages by other software entities. Autonomous software exceeds this limitation of "external-only invocation" by additionally encapsulating invocation control. In other words, autonomous software significantly differs from traditional software in that it takes over responsibility for deciding (in accordance with the demands of its users and administrators) when and under what conditions to become active and to react on external events. Because of this difference, the step toward autonomous software is also a highly challenging one which can not be realised casually by adding some lines of code, but one which deeply impacts all phases of software development, from early requirements capturing over implementation to integration with legacy systems.

Putting the vision of autonomous software into practice requires, first and foremost, a clear notion of software autonomy upon which precise schemes for autonomy specification can be built. In the computer science literature – especially in the literature on autonomic computing (e.g., [28]), computational autonomy (e.g., [24]), and agent-oriented software engineering (e.g., [38]) – diverse approaches have been proposed which are useful in this respect.[2] This chapter describes a generic, domain- and application-independent autonomy specification framework which brings these approaches and their key concepts together in a coherent whole. This framework comprises two parts: an autonomy matrix which gives a static view of software autonomy; and an autonomy transformation loop which captures the dynamic aspects of software autonomy. These two parts taken together provide the vocabulary necessary for talking about autonomous software. To our knowledge, they constitute the first comprehensive conceptual framework that allows for an *analysis* of computational

[1] Among these attributes, self-governing is most closely related to the original sense of the Greek term "auto+nomos" ("self+law"). As noted in [2], in European languages the word autonomous is commonly used to refer to something that is capable of self-government, while in American English its usage is stronger associated with self-directedness and independence from outside.

[2] Autonomic computing is a technological effort initiated by IBM that aims at building autonomous computing systems [28]. There are related efforts by other IT leaders, such as Sun's N1 initiative [33], HP's adaptive enterprise initiative [16] and Microsoft's dynamic systems initiative [22]. Computational autonomy is a research line that has its roots in the field of agent and multiagent technology and that explores autonomy from the perspective of computational intelligent systems.

autonomy that is crucial for engineering autonomous systems.

It should be stressed that autonomous software *is* still a vision, at least as far as the self-responsible carrying out of complex tasks without human intervention or feedback is concerned. With this respect, the framework presented in this paper should rather be understood as an "instruction manual" for dealing with autonomy issues that will arise in the future rather than a representative description of aspects of present-day software. Also, in many cases, full autonomy is not even desirable, as human designers or users want to be able to control the system at any point in time.

However, we will show that many *aspects* of autonomy already appear in existing applications (even if we cannot speak of "full" autonomy yet), and that the class of software applications in which autonomy plays a role is becoming increasingly important.

The chapter structure is as follows. First, section 2 introduces an exemplary system and application which is used to illustrate the various concepts relevant to autonomy specification. Next, sections 3 and 4 describe the autonomy matrix and the autonomy transformation loop, respectively. Finally, section 5 concludes with considerations on urgent open issues raised by autonomous software.

2 An Illustrative Example

Before we embark on a description of the key characteristics of autonomous software, it is useful to introduce an exemplary system that can be used to illustrate the concepts we suggest. The Link Exchange Simulation System LIESON [29] is a fully implemented system that is highly suitable for this purpose.

LIESON is a distributed, agent-based software simulator in which agents representing Web site owners manage the linkage (via hyperlinks) between their own site and others' on behalf of the Web site owners. These agents pursue two goals: Firstly, they seek to maximise the traffic attracted to their own(er's) site. Secondly, they want those sites to be most popular that express similar opinions as they do themselves, i.e. they aim at a link-based dissemination of their opinion. To further these goals, the agents negotiate with each other over linkage actions (such as laying a link, deleting it or labelling it with a positive/negative comment).

The system was primarily built as a simulation testbed for socially intelligent agents that are able to use a set of pre-defined interaction patterns when communicating with others in a goal-oriented way [30]. Apart from this social reasoning functionality, of course, they also have a rational, goal-oriented reasoning and decision-making apparatus that enables them to reason about the information they obtain about the current linkage network, to project future states, to assess the desirability of these states and to plan towards the achievement of these goals.

In the following sections, we will use LIESON as an example of a software system that is complex enough to incorporate the different kinds of autonomy we describe. More particularly, it is characterised by key features common

among those software applications in which autonomy can be seen to play a crucial role:

- *Spatial distribution of data and control:* The different Web site owners who deploy agents are stakeholders with potentially incompatible motives. Typically (i.e. unless a distributed system is deployed by a single stakeholder for the purpose of distributing a complex task among "strictly cooperative" units), different parts (here: agents) of the software follow their own design objectives and are not necessarily concerned with meeting global coherence.

- *Mutual observation:* Since agents have no access to each other's internal design, there is a need for monitoring others' behaviours and reacting appropriately to it. As we shall see below, this is very important in terms of the observer perspective that is assumed when modelling the autonomy of a piece of software.

- *Complex application environment:* The Web linkage domain is an application environment characterised by constant evolution and uncertainty regarding the current status of the global network of hyperlinks due to the impossibility of constantly monitoring all Web sites and their links. Typically, such environments require that agents self-responsibly decide to prioritise their "goals" and to reason about possible paths of computation. Quite obviously, if this entails that program behaviour may deviate from what was expected by its user or designer, then we are dealing with – at least partial – autonomy of the software.

While LIESON is a very particular application, we shall show below that many of these features are also present in other kinds of software. In particular, we will argue that they do not only occur in agent-based software systems.

3 Software Autonomy – The Static View

3.1 Autonomy and Agency

In the above discussion of LIESON as a typical application in which autonomy analysis makes sense, the concept of *agents* was used in accordance with common usage, to refer to a software unit that is able to pursue its design objectives autonomously, flexibly and in interaction with humans and other agents (e.g., [25]).

The design of LIESON heavily draws upon *agent* and *multiagent* technology [34] that have their origins in (Distributed) Artificial Intelligence research. In these areas, *autonomy* is considered one of the core defining elements of *agency* [37]. In fact, it is often only through the aspect of autonomy (in the sense of self-governance without external intervention) that agents can be distinguished from other software or hardware components, such as objects, modules, etc.

Therefore, it will sometimes be convenient in the following to refer to the software artefacts that represent Web site owners as "agents" and, more generally, to speak of agents whenever we mean autonomous software components that interact with each other and their environment. However, it should by no means be inferred from this that our discussion of autonomy only applies to agent-based approaches. Instead, we make use of the notion of agency as a *design metaphor* that emphasises autonomy (but also a certain amount of sophistication with respect to functionality, adaptiveness and the ability to operate in dynamic and/or uncertain environments, etc.) rather than in the sense of a set of technologies adopted from agent/multiagent system research (such as agent architectures, interaction protocols, agent communication languages, coordination mechanisms, etc.)

Also, what matters for our analysis when we picture a piece of autonomous software as an agent is the role of the *observer* perspective in autonomy modelling and analysis: By looking at an agent "from the outside", we put a great deal of emphasis on the process of *ascribing* autonomy qualities to software, and the agent notion supports this view (in contrast to concepts such as "unit", "program", "module" or "component").

Our analysis is based on the insight that this "ascribing" aspect of modelling autonomy is relevant for a variety of "conventional" software applications that are built without employing agent technology. As in the case of LIESON, this class of applications is characterised by distribution of data and control, mutual observation, and complexity of the application environment. The most suitable examples of such systems are applications in which different software (and hardware) components belong to different stakeholders and are not controlled by a central coordinating instance, such as

- peer-to-peer systems for distributed management and exchange of information,
- ad hoc networks for routing and communication between mobile devices,
- supply chain management systems which cater for flexible and loosely coupled B2B interactions,
- electronic marketplaces, auctions and trading platforms

and many others of a similar flavour. For this kind of systems that encapsulate components with a certain degree of autonomy – whether referred to as agents or not – we will next lay out a basic typology of different autonomy types.

3.2 The Autonomy Matrix

The autonomy matrix provides the basic vocabulary for analysing and modelling autonomy in terms of different autonomy types, and thus is foundational to our understanding of autonomy, at least as far as the *static* view of autonomy is concerned. This means that the autonomy matrix describes what kinds of

	perspective	
range	internal	external
performative	*capability*	*dependency*
deliberative	*motivation*	*control*
normative	*commitment*	*expectation*

Table 1: The autonomy matrix

autonomy a piece of software may possess, while disregarding the dynamics that may alter this autonomy status (these will be dealt with in section 4).

What this typology achieves is to break down the abstract notion of autonomy into a set of concepts the existence of which can be more easily identified and assessed in a concrete software system. This not only enables us to focus on the autonomy perspective of a system, but it also constitutes an important step towards *dealing* with the different kinds of autonomy that a system may exhibit.

The matrix itself is shown in table 1, and it distinguishes six types of autonomy and thus allows for a fine-grained specification of the autonomy status a software system might possess. The distinction made results from applying two different dimensions of autonomy: the *perspective of observation* and the *range* (or *scope*) of activity. In the following, first these two dimensions are explained, then the six autonomy types are described, and finally a basic autonomy specification pattern is presented.

3.3 The Perspective of Observation

The perspective of observation specifies the viewpoint of the observer who is describing an autonomous software component. Two kinds of perspectives can be distinguished, as they result in essentially different characterisations of the autonomy owned by a software system: the perspective of what can be called an internal observer, and the perspective of what is called here an external observer. An *internal* observer is an observer who is able to cross the boundary between the software and its environment and hence to obtain information about internal aspects of the agent that is not readily available when observing interaction with the environment. In particular, the autonomous software artefact itself is an internal observer of its own autonomy. An *external* observer, on the other hand, can only use what he perceives of the interaction of the software with its environment. Typical examples of external observers are users of a computer system or of a piece of software (who have not implemented the system themselves), market analysts who monitor the external activities of a company but are not provided with internals, etc.

Note that for both kinds of observer perspectives, several additional factors come into play that determine how the observer assesses the autonomy of the software under analysis. These are, *inter alia*,

- the *accuracy* of information about the (internal and external) behaviour of a component or system,
- the *processability* of this information (e.g. if its amount exceeds the capacities of the observer), and
- the *precision* of the information (i.e. the level of detail that is provided).

Also, of course, mixed-perspective observation is possible (and very common in practice). For example, software components with a social ability are usually capable of assuming the role of other components when observing themselves. Or, a person who has implemented a software system has access to the internals on the system, but is also externally observing its behaviour when testing it.

3.4 The Range of Activity

The range of activity that must be determined to gain a full understanding of the autonomy of an software entity has to do with whether its actions make it autonomous in behaving as an *individual* or in relation to its *social context*, and also whether we are referring to autonomy in terms of *actions* or in terms of *internal state*. With this respect, three categories can be distinguished.

The *performative* range of autonomy defines what actions a piece of software *can* take in principle, i.e. how it can influence its environment and the standing of others by taking action. In specifying performative autonomy, we typically identify

- the "primitives" of action the software disposes of (i.e. what changes it can effect in its physical/computer/network environment),
- the reliability with which it can employ them (a program is not really "capable" of performing some action if this action fails regularly),
- by the resources it can access (data, CPU time, network bandwidth, reasoning resources (representations, heuristics, algorithms, etc.) etc.),
- and by the complexity of "agendas" it can pursue while acting (e.g. how long or conditionally branched the action sequences are that it can pursue once it has decided on them).

If the autonomous software is ascribed mental qualities [21], (which is often the case, for instance, in agent-based approaches), certain "mental" activities can be seen as belonging to this class, but only in a low-level sense, e.g. adding and retracting facts to and from one's beliefs.

The *deliberative* range of activity has to do with the motivations that guide the behaviour of a software component, it describes what it *wants* to do. Unlike the performative range, this level is not concerned with the "what" of software activity, but rather with the "why" that stands behind an observed behaviour. Deliberation explains the goals and needs of an entity rather than its direct actions (by which the environment is manipulated or it communicates with

other agents), i.e. it provides the reasons for a certain behaviour. The activities that deliberation is aimed at are, quite naturally, only mental actions, such as generating and revising goals, adopting intentions or commitments, [36] etc.

The *normative* range, finally, defines what the software is *supposed* to do. Of course, this largely depends on *who* expects the software to do something: If it has a model of e.g. its own anticipated behaviour, the agent *itself* might have such normative expectations. More commonly, though, designer, the users, and other autonomous components it interacts with have a picture of what the agent "should" do. The difference to the performative view is that autonomy here is not framed by ability, but by the reactions of some other entity – it does not constrain anything the software artefact might do in principle. For example, a digital assistant who purchases goods on behalf of its owner on the Internet may be expected to pay for them at a normative level, while it may be at the same time able to commit fraud at the performative level. The difference between normative and deliberative autonomy, on the other hand, is that expectations have nothing to do with the current motives. For example, an agent may commit itself to something it does not want, and then it will certainly restrict his own autonomy by normative means of influence exerted upon itself [10].

Obviously, it is not possible to identify the latter two ranges (deliberative and normative) in just about any kind of software system, and it requires assuming a *knowledge-level* [23] outlook on computational systems that allows for ascribing mental qualities to them [21]. However, it will become clear from the description of the suggested individual types of autonomy below that assuming such a perspective is possible and reasonable in many situations.

Again, mixtures of these three different ranges are possible and do often occur in practice. For example, all deliberative and normative activities must have a physical counterpart since we are talking about physical software components, and this physical counterpart is always a performative activity.

3.5 Different Types of Autonomy

We will now discuss the six types of autonomy in more detail.

Capability – internal performative autonomy Internal performative autonomy is defined by the capabilities of a software component, i.e. by the capacity it has to influence its environment. It is an internal view, because the software can only perform actions it knows of, and must choose from these actions. A strongly related notion is that of resources, i.e. of cognitive or environmental aspects used to effect changes on the environment. Key concepts that are related to this autonomy category are *skill*, *ability*, *competence*, *expertise*, *learning*, and *knowledgeability*.

Example. In LIESON, agents' capabilities are given by (i) the physical action options they have depending on their knowledge and their environment, and, (ii) at a more abstract level, by their reasoning resources.

At the level of physical actions they can freely modify the outgoing links of their owner's Web site, as long as they have the required knowledge to "think of" these actions. For example, they can only lay a link to a site if they have found the site on the Web before and if they have obtained feedback from their human owner as to how much he likes the respective site. Also, they can execute special actions to explore new regions of the Web and they can visit known sites to update their knowledge about the outgoing links of that site.

Their autonomy is also restricted by reasoning resources, such as the number of future linkage states they can anticipate in each iteration to consider as a goal, by the number of goals they can schedule for future attainment, by they fact that they can only entertain a single conversation at a time, and by the time they need to wait for a response during a communicative encounter. In this respect, it is also a significant limitation that they can never really "catch" up with changes to the linkage network – assuming that each agent can execute a single action (visiting a site *or* sending a message to another agent *or* modifying an outgoing link) in each iteration, the global link configuration can change much faster than the agent can perceive these changes by visiting sites.

Dependency – external performative autonomy From an external point of view, not knowing what the internal capabilities of a component are, we can only observe the *dependencies* between the software and its environment (this relational aspect is emphasised in [7, 9]). The degree to which we ascribe autonomy to a piece of software in this sense depends on how closely its behaviour is coupled to that of its environment. For example, if an agent always performs an action after some specific event in the environment, it is very probable that this agent somehow depends on the effects of the environment (not necessarily in a physical sense, though; in the case of two deliberative agents it may also be the case that the action of one agent is spawned by the actions of the other through, e.g. a goal that arises from it).

Here, central concepts are *influence*, *distribution of scarce resources*, *compatibility* and *complementarity* of activities. The dependencies perspective of analysis is often assumed in coordination science, which sees coordination itself as the process of *managing* dependencies [20].

Example. LIESON agents mainly depend on each other by virtue of (i) the prescriptive force of communication patterns and (ii) correlations between agent capabilities and their utility functions.

As for (i), agents have to comply with the admissible message sequences prescribed by existing protocols, so that the range of potential responses is limited by communication patterns. This entails that any utterance will be followed by a fixed set of possible responses. Note that, although in principle any communicative message can be uttered in any situation the design of LIESON does not allow agents to dispose of certain communicative options at the deliberative level in this case, so that it is performative autonomy that is actually restricted here.

With (ii), the most basic autonomy constraint is that agents can only lay links towards existing sites, i.e. appearance or deletion of a site affects their action capabilities. With respect to utility scores, agents depend on the links others lay toward them, in the sense that they are not free to assign themselves any desired utility. (At a more abstract level, of course, this is a dependency stemming from the designer who defined the utility function and not from the actions of other agents.)

Motivation – internal deliberative autonomy Deliberative autonomy is strongly related to "freedom of will". Unlike capabilities who merely described what the software can do, motivation explains what it wants to do. A highly autonomous software component is self-motivated, and it is able to derive the justification for his actions through inference from first principles at any point in time. Adopting others' instructions, "commands", valuations or preferences, on the other hand, severely restricts this kind of autonomy.

We shall not ponder more deeply on the issue of whether perfect motivational autonomy is possible, since this is a fundamental philosophical problem that is not of much relevance to practical software engineering (the reader may have noticed that it sounds strange to speak of what the software "wants"). For the purposes of our analysis, we will always assume that there are some underlying explicitly implemented first principles that the software component follows and that set the scene for its range of activity. We will speak of autonomy in terms of motivation if, based on these principles, the autonomous software entity has the freedom to prioritise goals, to decide on whether these are fulfilled or unachievable, and to generate plans for achieving these goals.

Key issues that arise in this type of autonomy are related to questions of *goal selection* and *goal revision*, *intention*, *preference*, and *initiative*. One area in which these phenomena are studied is that of Belief-Desire-Intention agency [27], which deals with rational (i.e. goal-oriented) practical reasoning models for computational agents.

Example. The LIESON system allows agents to deliberate regarding linkage configurations they want to achieve. In each reasoning cycle, they randomly generate possible future states of the link network, make utility predictions for these states and schedule them according to these predictions in a rank-based "goal queue" (that is bounded in size) for future achievement. Then, the top option from this queue is selected as a current goal and de-queued, a plan is generated for its achievement and this plan is subsequently executed. Thereby, goals that cannot be achieved anymore or have been already achieved are simply skipped. The range of envisioned future states depends on the agent's knowledge of the linkage environment. Thus, an agent who knows more about existing sites and links can generate more future options and hence has a wider choice of possible goals.

If a plan that has to be executed to achieve a goal involves others' actions, communication processes are initiated to persuade the respective actions to

contribute to the plan. Likewise, the agent itself might be asked to participate in a joint plan, and LIESON agents are implemented in a way that forces them to at least consider cooperating with others. This limits motivational autonomy in a sense since it forces agents to treat potential future states suggested by other agents the same way as they would had they "thought" of these possible goals themselves. However, this is not too severe a limitation, since agreement to adopt such a goal still ultimately depends on whether it serves the private goals of the agents. But at least, he will have to deal with the suggestion, which may delay other decision-making steps.

Control – external deliberative autonomy From an external point of view, self-motivation is self-control, and loosing the autonomy of which goals to choose is perceived (if it is perceived at all) in the form of external control. The crucial difference to dependency is that dependency needs not be "realised", it is just the statement of a relationship between several entities. Control, on the other hand, must be *exerted* to become visible. Of course, the two notions are connected, though, since larger dependency can lead to higher controllability. Note, however, that these two kinds of autonomy need not be unidirectional: from a dependency point of view, a manager is often more dependent on his staff than the other way round, if we measure dependence by "the number of people that have to support you" (in a simplified world, the manager needs the support of the majority of the staff, while each staff member needs only the manager's support). On the other hand, the staff exerts much less control on the manager than the other way round, because they are (more or less) following his instructions (and the background of company objectives/culture) in making decisions, while he is supposed to make decisions himself.

The central concepts here are *power* [5], *opponent modelling* [4], and *regulation* of behaviour.

Example. In LIESON, the utility function that agents use to project the expected payoff of a possible future linkage situation is contingent on the current linkage situation (or, to be more precise, on agents' incomplete and possibly incorrect knowledge thereof). So if, for example, an agent with high popularity influences the utility values of others to a great degree, he has implicit power in the agent society, since his actions will entail a series of reactions on the side of "weaker" agents via changes to their utility scores. Note that this kind of control is different from (performative) dependencies, because the powerful agent affects the way in which agents select their *goals* rather than their capabilities to perform actions: agents still have the same action repertoire, but the set of actions that they consider *relevant* has changed.

From an external perspective, the autonomy restriction imposed on agents by the obligation to process others' requests is not observable. However, external observers can verify whether actions that have been requested of an agent are actually performed by that agent. So (although this need not hold in LIESON in general) situations may occur in which agents obey others' "commands" with

such regularity that a relationship of control manifests itself.

Commitment – internal normative autonomy For an autonomous entity to exhibit a certain behaviour that is not arbitrary with respect to its capabilities, it must *commit* itself to doing something particular. Thus, from the internal viewpoint, it sees its autonomy (possibly only temporarily) constrained by commitments it makes. The simplest kind of such commitments are intentions [11], which simply "shift" certain actions from the core reasoning components to the environment, so that they almost become part of the environment themselves (they will be executed without further reasoning, unless intention revision – which can be seen as an almost "external" interaction with those intentions – makes them undone). Much more interesting, however, are *social commitments* [6, 31, 32], which stem from interaction with others and constrain this kind of autonomy. It is important to note that these (internal) commitments are very different from the societal view of commitments because they only infringe autonomy if *adopted* by the individual [12, 10].

Frequently used concepts of this category are *(dis)obedience*, *submission*, and *benevolence*.

Example. We observe both social and non-social commitment-relevant autonomy restrictions in LIESON.

The main social autonomy restriction is that agents have to adhere to a set of pre-defined communication patterns during inter-agent dialogues. Therefore, once they choose a "path" in the set of possible response strategies, they commit themselves to the restrictions imposed by the hard-coded patterns. So, for example, if they agree to execute an action they actually internally commit to executing it. However, they are able to learn from experience which of the given patterns to apply in a given situation in a strategic (i.e. utility-maximising) way. This ensures that such social commitments are not generated if they contradict the agent's private interests, and, in fact compensates them for the above loss of utility by providing a mechanism of choice at the next level.

At the non-social level, agents commit to the goals they have selected, and will maintain these goals unless they have been achieved or become inachievable. This commitment is different from a motivation, because it does not *spawn* a certain kind of activity. Instead, it serves to *maintain* such activity without further rational, goal-oriented reasoning.

Expectation – external normative autonomy The final category is that of *expectations*. Expectations are formed externally and concern interaction of the agent with its environment. They are normative in the sense that they express the rights, duties, obligations an agent is subject to, and possibly also the sanctions it has to face if it breaks these expectations. However, such expectations need not be *a priori* deontic claims made by the designer of the system, a social system, etc. They can also express the "image" of an agent that has been formed through its past behaviour by expecting a similar behaviour

from it in the future [3].

Typical concepts in this category are *norm*, *convention* [13, 17], role [26, 19], *obligation* [31], *institution* [15, 1], and social order [8, 14].

Example. For this last category, the internal commitments of agents do not matter, since they cannot be observed. Still, we can infer the existence of expectations in LIESON from the fact that when agents plan their strategic communication behaviour, they employ decision-theoretic utility maximisation over the set of admissible continuations of the dialogue. This is only possible because they can expect the agent to adhere to the given protocols. For example, if there are only three alternative responses A, B and C that the other agent can use and probabilities for these have been derived from previous experience, the expected utility of the whole conversation can be easily computed. If, on the other hand, the other agent were allowed to produce *any* reply, the utility calculation would be very imprecise.

Additionally, the fact that the agents use previous experience to infer the future behaviour of other constitutes an expectation-based strategy by itself: each agent expects that others will behave in a similar way in the future as they did in the past.

3.6 A Pattern for Autonomy Status Descriptions

Based on the six types of autonomy, we can present a semi-formal pattern for *autonomy status descriptions* that specify the autonomy situation of an agent (at this point, we do not deal with groups of agents but only with single agents – most of the concepts carry over, however, if groups are viewed as collective actors):

> Component A is T-autonomous with respect to the influence of B in context C from the perspective of an observer O because it acts according to model M' rather than according to the model M that O expected.

A set of such status descriptions characterises which autonomy situation A is in.

For example, saying that

> *Agent 1* is control-autonomous with respect to the influence of *Agent 4* in the context of *adding a link from Agent 1 to Agent 2* from the perspective of *Agent 4* because it *adds the link to Agent 2 regardless of Agent 4's plea to refrain from this action*

allows us to make a rather precise statement regarding a specific type of autonomy in the behaviour of a particular LIESON agent.

It is important to note that this kind of description focuses on the relationship between *autonomy* and *determination*, i.e. that autonomy is always identified by observing a deviation from a behaviour that would have resulted

from adhering to some assumed model. In other words, M is an assumption regarding the factors B that determine A's behaviour in situation C, and after observing the actual behaviour of A (either from the inside or from the outside), we have to overthrow M in favour of some different model M'. In the following sections, we will turn to the dynamic nature of autonomy, where this modelling activity plays an important role in reasoning about autonomy in a complex system composed of autonomous software artefacts.

4 A Dynamic View of Software Autonomy – The Autonomy Transformation Loop

Specifying autonomy with the above categories helps to define precisely the capabilities, motivations and social comportment of an autonomous software component with respect to a particular environment (that may contain other such components). In particular, the autonomy of an entity is always a relative notion that refers to a second entity: whenever there is a lack of *auto*nomy, there must be *hetero*nomy, i.e. someone else must "hold" the autonomy that the first component lacks. Such autonomy specifications provide a rather *structural* specification a system that is made up of several autonomous components, a "snapshot" of who can, wants and has committed himself to doing what at some specific point in time. What is amiss is a *dynamic* view that explains how autonomy owned by a computational entity such as a software system evolves, that is, how it transformed from one status into another.

4.1 Transformation through Interaction and Communication

From work on adjustable computational autonomy it is known that *interaction* is essential as a process that can change the status of autonomy. This implies that there is a basic feedback loop between autonomy and interaction, as shown in figure 1. Here, the concept of interaction is understood in the most encompassing sense, i.e. as interaction with the environment, the user, the designer, etc. The general intuition is that while interaction is unfolding, the range of

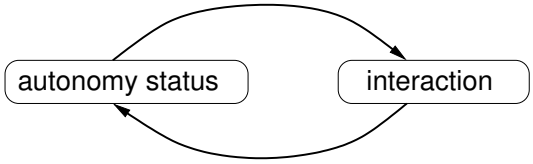

Figure 1: Basic autonomy transformation loop

possible behaviours undergoes constant changes. For example, when LIESON agents explore the environment, they may discover a new site which alters their

capability repertoire since new linkage actions become relevant. At the same time, their link configurations may change because of the actions of others, so that different *goals* appear promising or achievable, while others may have to be disregarded. Or, a promise to maintain a link to someone else may have caused a *commitment* that they must now be held or at least considered. From an external point of view, the changes in the above example may result in autonomy status changes, depending on who is observing the respective autonomous entity and how he is modelling it.

Looking at the process of autonomy transformation through interaction more closely, the simple feedback loop of figure 1 can be further refined, if we make the assumption that the observer or participating agent is *reasoning about autonomy* himself (i.e., the interesting case). Under this assumption, the *decision-making* process of interacting parties is preceeded by *modelling* and *decision-making* processes that take the autonomy status into account.

Modelling autonomy consists of identifying dependencies, analysing the distribution of resources and the control flow between components and possibly also building models of others' modelling processes.

This modelling activity only makes sense if it has the potential to influence the decision-making process of the actor. Of course, decision-making procedures also depend on other aspects such as objectives, general domain knowledge, etc. The less predictable the behaviours of other parts of the system are, the more important the aspect of autonomy becomes.

As shown in figure 2, *communication* plays a special role in mediating between modelling and decision making. This insight follows as a natural consequence once we discriminate between "physical" actions that can be taken and which modify the environment and purely communicative action. This is because communication allows for obtaining information about the current autonomy status before making decisions. This information feeds into the autonomy modelling processes of interacting parties, and, even more importantly, it can be used to anticipate and plan future autonomy configurations in a system. For example, if a component is informed of something that contradicts prior belief and decides to revise its belief, it is increasing its dependency on the knowledge of the informer. The informer, on the other hand, is submitting himself to the expectation that he tells the truth, and unless he states something like "I am not sure" or "I say X but won't be held responsible if it is not accurate" he implicitly agrees to be labelled as "liar" if others find out he did not tell the truth. In other words, he is restricting his own normative autonomy by committing himself to truthfulness. As another example, consider a boss who is assigning a task to his employee and tells him "if you can't cope with this, you're fired". Through this statement, he is increasing the employee's dependency on him by threatening with loss of employment (rather than just with criticism or conflict). At the same time, delegation of the task restricts the deliberative autonomy of the employee; he must now "want" to complete the task more than anything else (at least at his workplace). If the boss also says, "Smith and Miller are going to help you in this, they are at your command", he is increasing the employee's performative autonomy, because the employee is

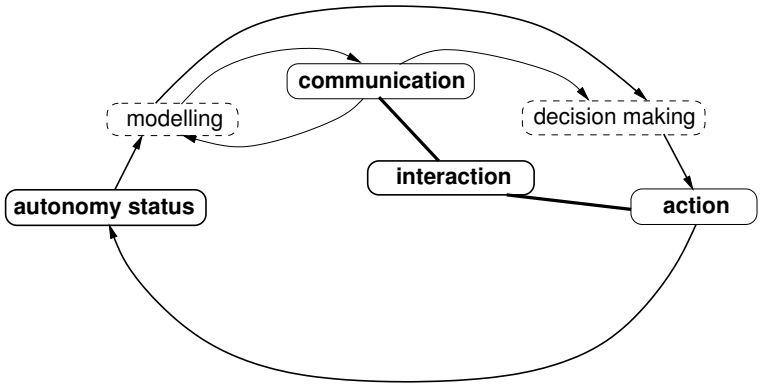

Figure 2: Extended autonomy transformation loop

endowed with additional resources that may increase his productivity.

It is the capacity of allowing autonomous entities to predict future autonomy states that makes communication highly suitable for *coordinating* future behaviours. In a way, engaging in communication means seeking to reduce the contingencies inherent to the behaviour of other autonomous components in the system.

4.2 Examples of Autonomy Transformation

Theoretically, any interaction can increase or decrease any of the six autonomy types in our autonomy matrix at the same time. Usually, though, interactions hardly affect more than two autonomy range types at the same time, and the effects are analysed from both an internal and an external perspective at the same time.

Let us look at some examples in the LIESON application domain, where we are going to discuss how the autonomy status of certain parts of a system can be influenced, viewed from three perspectives: (1) the designer of the overall system, (2) a human user that is using a LIESON agent to manage linkage from and to his Web site, and (3) a LIESON agent seeking to maximise the dissemination of his opinion. In these examples, we are going to mark each autonomy transformation by a combination of pairs of letters (e.g. **pi** for performative-internal (capability), **ne** for normative-external (expectation), etc.) and a + or **-** symbol for increase and decrease in the respective autonomy type, respectively (such that, for example **+pi-ne** stands for "increase in capabilities, increase in others' expectations (decrease in "expectation-related autonomy")"). If different aspects of a single autonomy category are affected by a transformation (or if we want to express the autonomy changes of different parties at the same time), of course, this would have to be accounted for by using a more elaborate syntax.

System designer perspective The system designer has access to the internals of the agent design, so that he can, in theory, explain whatever is happening in the deployed system, if he is given information about the preferences of the human users, and if we assume that these users would not manipulate the code of the LIESON application.

So, for example, if the designer knows the knowledge base contents of a LIESON agent, he can derive the capabilities of this agent, and the transformations **+pi** and **-pi** that occur when an agent obtains new information about sites or links (knowing the internal design, **pi** autonomy is identical to **pe** autonomy). However, the motivation of agents, although in principle regulated by a specific goal prioritisation scheme, is not entirely predictable. This is because agents randomly generate possible future linkage states in each round that they will consider in determining their goals. Here, the random generator is a source of uncertainty, so that the designer cannot tell whether a new, interesting alternative has been found in a specific reasoning iteration that would modify (**+di**/**-di**) the agents autonomy in choosing a goal. However, the designer may certain **+de** and **-de** changes, e.g. when a new site is created that makes it theoretically possible for the agent to have optimal linkage with this site as a goal or when that site disappears again. As for normative autonomy, the designer knows that, by its design, the agent can only use the prescribed communication protocols, which implies that other agents can expect it to answer in a fairly predictable way. So if the designer observes, for example, that the agent is committing himself a certain linkage action in the process of communication, this reduces both its internal and external normative autonomy (**-ni-ne**).

As a final example, let us assume the designer is observing the system in operation and finds out that agents are acting in an overtly aggressive and deceptive fashion, so that linkage is highly sub-optimal from a global perspective. In the system as it is, the only measures the designer could take would be to re-implement the system or to insert additional agents into the running system that are programmed in a more cooperative fashion in the hope that LIESON agents (who are adaptive in the sense that they learn from communication experience) would adopt the more cooperative interaction behaviour exhibited by new, "friendly" agents.

This nicely illustrates the potential dangers associated with autonomous systems, and the importance of taking appropriate precautions at design time.

Human user perspective Typically, a human Web site owner is only given a rough description of the internal design of his LIESON agent, i.e. he knows something about the general reasoning and communication mechanisms employed by the agent but not much about the details of goal prioritisation, communication learning, future utility estimation etc.

For the human user, agent autonomy is largely dependent on the feedback the user provides regarding his preferences over others' Web sites. The more complex the preference profile, the fewer profitable future link constellations will the LIESON agent be able to identify, and the smaller will his goal repertoire

be (**-di**), but not knowing the internal design of the agent it is questionable whether the user would be aware of a corresponding **-de** transformation.

Also, the human user has the ability to inspect the linkage network himself, so he will notice **-de** and **-ne** changes in his agent's autonomy caused by some other agent becoming very powerful, once these transformations become manifest in agent's actions (e.g. striving to obtain a link from the powerful site or obeying the requests of this powerful site).

Obviously, the human designer is also able to check what the action options of his agent are, i.e. **+pe** and **-pe** changes can also be observed, although they need not coincide with the respective **+pi** and **-pi** transformations, because the agent may have incomplete and/or incorrect knowledge of the network.

LIESON agent perspective From an autonomy standpoint, an agent is only an external observer of another agent, so most of the above observations carry over to agent observers. What is really interesting from an agent perspective are transformations that concern normative autonomy and that are caused by communication. In fact, it is *only* by communicating that agents can influence each other's autonomy status deliberately, as they do not have direct access to the capabilities and motivations of others.

Depending on the nature of the communication protocols that are used and on the physical action consequences that result from them, there exists a variety of possible autonomy transformations. Examples include:

- A promise to execute some action or a complex series of actions in the context of a distributed plan. This causes the expectation (**-ne**) that the agent will adhere to the promise, unless the promise is cancelled. On the side of the *debtor* of this commitment, such a promise constitutes an increase in capability (**+pi**), since the agent is capable of evoking an action he cannot perform itself.

- A reciprocal long-term agreement, for example making a contract to pay a fixed amount of money every month to "rent" an ingoing link from someone else. The agent who is "renting" the link is loosing the capability of disposing of a certain amount of money, while he gains the possibility of achieving goals with a higher utility, and he is expected to make regular payments (**-pi+di-ne**).

- A threat to sanction a certain behaviour of the other. This imposes normative expectations (to refrain from the behaviour and to implement the sanction, respectively) on both sides, but, if successful, the threatening agent will have a larger autonomy in predicting future states of the linkage network, as it can rule out certain behaviours of others (**-pi-ne**).

These examples nicely illustrate that a thorough planning of the communication and social reasoning functionality is essential in the process of designing distributed systems that are able to handle autonomy issues effectively.

5 Conclusion

Since its inception in the 60s, software engineering has targeted at the development of software whose functionality and structure is, directly or indirectly (via other software entities), under full control of its users and administrators. With the advent of autonomous software, this situation changes fundamentally: taking autonomy as a software property serious means to hand action choice and invocation control over to the software itself. In the ultimate vision, autonomous software is able to take on any responsibility needed for successfully running an application under self-control and, thus, to fully hide its complexity and the complexity of its environment from users and administrators. This obviously is a long-term vision, although today software products (especially in the area of data and server management) are already available which indicate that this vision is much closer to reality than skeptics might think. We envisage that over the years software systems will appear which exhibit increasing levels of autonomy. This is not to say that autonomy orientation will replace other software models such as object orientation and component orientation; what autonomy orientation does is to provide a qualitatively new level of abstraction on top of existing models such as object orientation and component orientation.

In the light of the state of the art in developing autonomous software, currently the most urgent issue is to provide developers with engineering techniques – methods, formalisms, tools, and so forth – which enable them to appropriately tailor the type and extent of autonomy a software system should own. Thereby "appropriately" means that software autonomy is neither unnecessarily cut down (as this would result in software being not remarkably distinct from traditional software), nor unnecessarily admitted (as this would result in an increased risk of undesirable or even chaotic system behaviour). To tailor software autonomy appropriately without any supportive techniques is a highly complicated task even in the case of relatively simple applications such as LIESON (the LIESON case studies provided throughout this chapter prove this). The specification framework introduced in this chapter can serve as a guideline for devising such techniques. In particular, the framework presented in this chapter shows that it is possible to break down the abstract notion of autonomy into a set of concepts such as dependency and expectation which are sufficiently concrete to be processable at the level of software engineering. Once such techniques are available, other important questions can be addressed precisely, for instance: How to identify the need for autonomous software in a particular application? How to validate and verify autonomy as a software property? How to implement autonomy? How to make sure that software adapts its autonomous behaviour appropriately during run time? Questions like these require considerable research efforts to be answered, but these efforts are worthwhile in the light of the tremendous benefits autonomous software offers.

References

[1] Wolfgang Balzer and Raimo Tuomela. Social Institutions, Norms, and Practices. In Rosaria Conte and Chrysanthos Dellarocas, editors, *Social Order in Multiagent Systems*, pages 161–180. Kluwer Academic Publishers, Norwell, MA, Amsterdam, The Netherlands, 2001.

[2] J.M. Bradshaw et al. Adjustable autonomy and human-agent teamwork in practice: an interim report on space applications. In H. Hexmoor, C. Castelfranchi, and R. Falcone, editors, *Agent autonomy*, pages 243–280. Kluwer Academic Pub., 2003.

[3] W. Brauer, M. Nickles, M. Rovatsos, G. Weiß, and K. F. Lorentzen. Expectation-Oriented Analysis and Design. In *Proceedings of the 2nd Workshop on Agent-Oriented Software Engineering (AOSE-2001) at the Autonomous Agents 2001 Conference*, volume 2222 of *LNAI*, Montreal, Canada, May 29 2001. Springer-Verlag, Berlin.

[4] D. Carmel and S. Markovitch. Learning models of intelligent agents. In *Thirteenth National Conference on Artificial Intelligence*, pages 62–67, Menlo Park, CA, 1996. AAAI Press/MIT Press.

[5] C. Castelfranchi. Social power. a Point Missed in Multi-Agent, DAI and HCI. In Y. Demazeau and J. P. Muller, editors, *Decentralized A.I.*, pages 49–62. Elsevier Science Publishers, 1990.

[6] C. Castelfranchi. Commitments: from individual intentions to groups and organizations. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 41–48, 1995.

[7] C. Castelfranchi. Guarantees for Autonomy in Cognitive Agent Architecture. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Proceedings of the First International Workshop on Agent Theories, Architectures and Languages (ATAL-94)*, pages 56–70. Springer-Verlag, 1995.

[8] C. Castelfranchi. Engineering social order. In *Working Notes of the First International Workshop on Engineering Societies in the Agents' World (ESAW-00)*, 2000.

[9] C. Castelfranchi. Founding Agent's "Autonomy" on Dependence Theory. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 353–357, 2000.

[10] C. Castelfranchi, F. Dignum, C. M. Jonker, and J. Treur. Deliberate normative agents: Principles and architecture. In *Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Orlando, FL, 1999.

[11] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[12] P. R. Cohen and H. J. Levesque. Teamwork. *Noûs*, 35:487–512, 1991.

[13] R. Conte and C. Castelfranchi. From conventions to prescriptions: Toward an integrated theory of norms. In *Proceedings of the ModelAge-96 Workshop*, Sesimbra, Italy, January 1996.

[14] Rosaria Conte and Chrysanthos Dellarocas, editors. *Social Order in Multiagent Systems*. Kluwer Academic Publishers, Norwell, MA, Amsterdam, The Netherlands, 2001.

[15] Sue E. S. Crawford and Elinor Ostrom. A grammar of institutions. *American Political Science Review*, 89(3):582–599, 1995.

[16] Hewlett-Packard. Adaptive enterprise initiative, http://www.hp.com/large/globalsolutions/ai.html?jumpid=go/adaptive, 2003.

[17] N. R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, 8(3):223–250, 1993.

[18] N.R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.

[19] E.A. Kendall. Agent roles and role models: New abstractions for multiagent system analysis and design. In *International Workshop on Intelligent Agents in Information and Process Management*, 1998.

[20] T. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[21] J. McCarthy. Ascribing mental qualities to machines. In V. Lifschitz, editor, *Formalizing Common Sense: Papers by John McCarthy*. Ablex Publishing Corporation, Norwood, NJ, 1990.

[22] Microsoft. Dynamic systems initiative, http://www.microsoft.com/presspass/press/2003/mar03/03-18dynamicsystemspr.asp, 2003.

[23] A. Newell. The Knowledge Level. *Artificial Intelligence*, 18(1), 1982.

[24] M. Nickles, M. Rovatsos, and G. Weiss, editors. *Computational Autonomy. First International Workshop (AUTONOMY-2003), July 14, 2003, Melbourne, Australia*, volume 2969 of *LNCS*, Berlin, Germany, in preparation, 2004. Springer-Verlag.

[25] H.S. Nwana. Software agents: An overview. *The Knowledge Engineering Review*, 11(3):205–244, 1996.

[26] O. Pacheco and J. Carmo. A role based model for the normative specification of organized collective agency and agents interaction. *Journal of Autonomous Agents and Multi-Agent Systems*, 6(2):145–184, 2003.

[27] A. S. Rao and M.P. Georgeff. An abstract architecture for rational agents. In W. Swartout C. Rich and B. Nebel, editors, *Proceedings of Knowledge Represenation and Reasoning (KR&R-92)*, 1992.

[28] IBM Research. Autonomic computing, http://www.research.ibm.com/autonomic/, 2003.

[29] M. Rovatsos. *LIESON – User's Manual and Developer's Guide*. http://www7.in.tum.de/∼rovatsos/lieson/users-manual.pdf, 2002–2003.

[30] M. Rovatsos, G. Weiß, and M. Wolf. An Approach to the Analysis and Design of Multiagent Systems based on Interaction Frames. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-02)*, Bologna, Italy, 2002. ACM Press.

[31] M.P. Singh. Multiagent systems as spheres of commitment. In *ICMAS-96 Workshop on Norms, Obligations, and Conventions*, 1996.

[32] M.P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.

[33] Sun. N1 initiative, http://www.sun.com/software/solutions/n1/index.html, 2003.

[34] G. Weiß, editor. *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Cambridge, MA, 1999.

[35] G. Weiß. Agent orientation in software engineering. *Knowledge Engineering Review*, 16(4):349–373, 2002.

[36] M. J. Wooldridge, editor. *Reasoning About Rational Agents*. The MIT Press, Cambridge, MA, 2000.

[37] M. J. Wooldridge and N.R. Jennings. Agent theories, architectures, and languages: A survey. In M. J. Wooldridge and N.R. Jennings, editors, *Intelligent Agents*, Lecture Notes in Artificial in Artificial Intelligence, vol. 890, pages 1–39. Springer-Verlag, Berlin et al., 1995.

[38] M. J. Wooldridge, G. Weiss, and P. Ciancarini, editors. *Agent-Oriented Software Engineering II. Second International Workshop, AOSE 2001, Montreal, Canada, May 29, 2001*, volume 2222 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin et al., 2002.