Requirements-Driven Design of Autonomic Application Software

Alexei Lapouchnian, Yijun Yu, Sotirios Liaskos, John Mylopoulos

Department of Computer Science, University of Toronto

{alexei, yijun, liaskos, jm}@cs.toronto.edu

Abstract

Autonomic computing systems reduce software maintenance costs and management complexity by taking on the responsibility for their configuration, optimization, healing, and protection. These tasks are accomplished by switching at runtime to a different system behaviour – the one that is more efficient, more secure, more stable, etc. – while still fulfilling the main purpose of the system. Thus, identifying the objectives of the system, analyzing alternative ways of how these objectives can be met, and designing a system that supports all or some of these alternative behaviours is a promising way to develop autonomic systems. This paper proposes the use of requirements goal models as a foundation for such software development process and demonstrates this on an example.

1 Introduction

As management complexity and maintenance cost of software systems keep spiraling upward, Autonomic Computing (AC) [6][11] promises to move most of this complexity from humans to the software itself and to reduce software maintenance costs, thereby drastically reducing the dominant cost factor in the software lifecycle. This reduction is expected to come about because autonomic software can self-configure at runtime to match changing operating environments; it can self-optimize to tune its performance or other software qualities; it can self-heal instead of crashing when its operating environment turns out to be inconsistent with its built-in design assumptions; and it can self-protect itself from malicious attacks.

There are three basic ways to make a system autonomic. The first is to design it so that it supports a space of possible behaviours. These are realized through an isomorphic space of possible system configurations. To make such designs possible, we need concepts for characterizing large spaces of alternative behaviours/configurations. Goal models in requirements engineering [1] and feature models in software product line design [5] offer such concepts. For example, the possible behaviours of an autonomic meeting scheduling system might be characterized by a goal model that indicates all possible ways of achieving the goal "Schedule Meeting." The second way of building an autonomic system is to endow it with planning capabilities and possibly social skills so that it can delegate tasks to external software components (agents), thereby augmenting its own capabilities [15]. Evolutionary approaches to autonomic systems [14], such as those found in biology, constitute a third way of building autonomic software. We only explore the first way in this paper.

The purpose of this paper is to show that requirements goal models can be used as a foundation for designing software that supports a space of behaviours, all delivering the same function, and that is able to select at runtime the best behaviour based on the current context. The advantages of this approach include the support for traceability of software design to requirements as well as for the exploration of alternatives and for their analysis with respect to quality concerns of stakeholders. We also sketch an autonomic systems architecture that can be derived from these goal models. We then illustrate how self-configuration and self-optimization behaviour can be achieved in our approach and how properly enriched goal models can serve as sources of knowledge for these activities.

The rest of the paper is structured as follows. We introduce goal-oriented requirements engineering – the foundation of our approach – in Section 2. There, we also discuss the use of goal models for capturing and analyzing alternatives as well as outline how design-level views can be created from goal models. In Section 3 we discuss the use of goal models for the design of Autonomic Computing systems, while Section 4 presents our approach in detail. Discussion and conclusion are in Sections 5 and 6 respectively.

2 Background

In this section, we introduce goal-oriented requirements engineering as well as some relevant work on using goal models for customizing and configuring software.

2.1 Goal-Oriented Requirements Engineering

A major breakthrough of the past decade in (Software) Requirements Engineering is the development of a framework for capturing and analyzing stakeholder intentions to generate functional and non-functional (hereafter quality) requirements [1][12][17]. In essence, this work has extended upstream the software development process by adding a new phase (*early* requirements analysis) that is also supported by engineering concepts, tools and techniques, like its downstream cousins. The fundamental concepts used to drive the new form of analysis are those of *goal* and *actor*. For example, a stakeholder goal for a library information system may be "Fulfill Every Book Request". This goal may be decomposed in different ways. One might consist of ensuring book availability by limiting the borrowing period and also by notifying users who requested a book that the book is available. This decomposition may lead (through intermediate steps) to functional requirements such as "Remind Borrower" and "Notify User". A different decomposition of the initial goal, however, may involve buying a book whenever a request can't be fulfilled[1]. Obviously, there are in general many ways to fulfill a stakeholder goal. Analyzing the space of alternatives makes the process of generating functional and quality requirements more systematic in the sense that the designer is exploring an *explicitly represented* space of alternatives. It also makes it more rational in that the designer can point to an explicit evaluation of these alternatives in terms of stakeholder criteria to justify his choice. An authoritative account of Goal-Oriented Requirements Engineering (GORE) can be found in [16].

At the very heart of this new phase of Software Engineering are goal models that represent stakeholder intentions and their refinements using formally defined relationships. Functional stakeholder goals are modeled in terms of hard goals (or simply goals, when there is no ambiguity). For example, "Schedule Meeting" and "Fulfill Every Book Request" are functional goals that are either fulfilled (satisfied) or not fulfilled (denied). Other stakeholder goals are qualitative and are hard to define formally. For instance, "Have Productive Meeting" and "Have Satisfied Library Users" are qualitative goals and they are modeled in terms of *softgoals*. A softgoal by its very nature doesn't have a clear-cut criterion for its fulfillment, and may be fully or partially satisfied or denied.

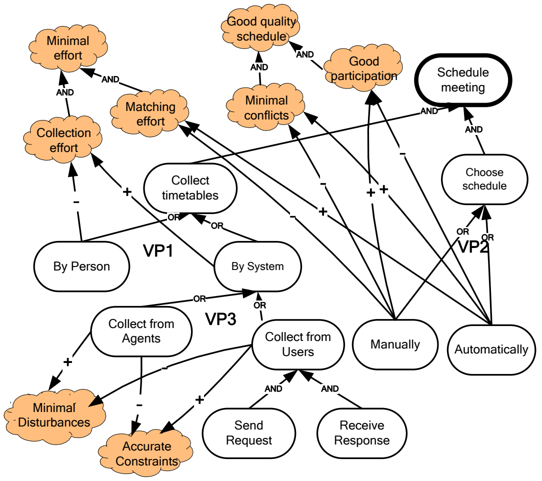

Figure 1: A goal model showing interdependencies among goals and qualities.

Goals and/or softgoals may be related through AND/OR relationships that have the obvious semantics that AND-decomposed subgoals must all

[1] Admittedly not a very practical one!

be attained for their parent goal to be achieved and at least one OR-decomposed subgoal needs to be achieved for achieving its parent goal. In addition, goals/softgoals can be related to softgoals through help (+), hurt (–), make (++), or break (--) relationships. This simple language is sufficient for modeling and analyzing goals during early requirements, covering both functional and quality requirements, which in this framework are treated as first-class citizens.

To illustrate what goal models are, and what they can do for the design of autonomic software, let’s suppose that the task is to design a system that supports the scheduling of meetings (Figure 1). Clearly, several stakeholders here (managers, engineers, admin staff, etc.) share the goal “Schedule Meeting”, which can be AND-decomposed into “Collect Timetables” and “Choose Schedules”. Each of the subgoals has two alternative solutions: it can either be done “By Person” (“Manually”) or “By System” (“Automatically”). A system can collect a timetable “From Agents” for each potential meeting participant (e.g., from his secretary) or directly from participants (“From Users”); the latter goal is further AND-decomposed into “Send Request” and “Receive Response” (regarding timetables).

Quality attributes are represented as softgoals (cloudy shapes in the figure). For our example, four top-level desired qualities are “Minimal (scheduling) Effort”, “Good Quality Schedule”, “Minimal Disturbance” and “Accurate (timetable) Constraints”. These can be decomposed into sub-softgoals. For example, “Minimal Effort” can be fulfilled by minimizing “Collection Effort” and “(human) Matching Effort”. Similarly, “Good Quality Schedule” is fulfilled by having “Minimal Conflicts” and “Good Participation”. Clearly, collecting timetables manually is a tedious task. Thus, it hurts the softgoal “(minimize) Collection Effort”. As shown in Figure 1, such partial contributions are explicitly expressed in the goal model. In order not to clutter the figure, we don’t show all partial contributions. For instance, when timetables are collected by a person, they tend to be more accurate. Thus, there should be a positive contribution from the “By Person” goal to the “Minimal Conflicts” softgoal.

In all, the goal model of Figure 1 shows six alternative ways for fulfilling the goal “Schedule Meeting”. It is easy to verify that generally the number of alternatives represented by a typical goal model depends exponentially on the number of OR decompositions (labelled as variation points “VP1” through “VP3” in Figure 1) present in the goal model (assuming a “normalized” goal model where AND and OR decompositions are interleaved). As such, goal models make it possible to capture during requirements analysis – in stakeholder-oriented terms – all the different ways of fulfilling top-level goals. A systematic approach for thoroughly analyzing the variability in the problem domain with the help of high-variability goal models is discussed in [10]. The paper proposes a taxonomy of *variability concerns* as well as the method for making sure these concerns are properly addressed during the goal model elicitation process. Now, if one were designing an autonomic software system, it would make sense to ensure that the system is designed to accommodate most/all ways of fulfilling top-level goals (i.e., delivering the desired functionality), rather than just some.

Another feature of goal models is that alternatives can be ranked with respect to the qualities modeled in the figure. Assigning to the system the responsibility for collecting timetables and generating a schedule is in general less time-consuming (for people), but results more often in sub-optimal schedules, since the system doesn’t take into account personal/political/social considerations. So, the model of Figure 1 represents a space of alternative behaviours that can lead to the fulfillment of top-level goals, and also captures how these alternatives stack up with respect to desired stakeholder qualities.

2.2 Reasoning with Goal Models

While goal models are a useful notation for modeling and communicating requirements, we are interested in the automated analysis of these models. To this end, Sebastiani et al. [13] present a sound and complete satisfaction label propagation algorithm that given a goal model with a number of alternative ways to satisfy its goals and a number of softgoals representing important quality concerns, can be used to find the alternative that achieves the top-level goal of the model while addressing these quality constraints. For instance, one can specify (see Figure 1) that the goal “Schedule Meeting” has to be achieved together with the non-functional constraint “Minimal Effort”. The algorithm will determine that the alternative where the collection of timetables from users and the selection of the meeting schedule are done automatically is the best option.

Additionally, given a goal model with a set of labels (i.e., satisfied, partially satisfied, etc.), the algorithm in [3] propagates these labels up towards the root goals using the semantics of AND/OR decompositions and contribution links. Thus, this algorithm can be used to determine how the satisfaction/denial of lower-level goals affects the satisfaction of higher-level ones. For example, the failure of the goal "Choose Schedule" in Figure 1 will deny the satisfaction of the goal "Schedule Meeting" even if its sibling goal "Collect Timetables" is satisfied.

2.3 Goal Model-based Customization and Configuration

There has been interest in trying to apply goal models in practice to configure and customize complex software systems. In [4], goal models were used in the context of "personal software" (e.g., an email system) specifically to capture alternative ways of achieving user goals as a basis for creating highly customizable systems that can be fine-tuned for each particular user. The Goals-Skills-Preferences approach for ranking alternatives is also presented in [4]. The approach takes into consideration the user's preferences (the desired quality attributes) as well as the user's physical and mental *skills* to find the best option for achieving the user's goals. This is done by comparing the skills profile of the user to the skills requirements of various system configuration alternatives. For example, for the user who has difficulty using the computer keyboard, the configurator system will reject the alternatives that require typing in favour of voice input.

In a generic version of the above approach, *capabilities* of the system's environment (e.g., the budget the customer allocated for the project or the current hardware/software environment in a customer organization) are used to prune the space of alternatives for achieving goals by removing infeasible ones, while *preferences* will be used to rank the remaining alternatives.

Goal models can also be used for configuring complex software systems based on high-level user goals and quality concerns [9][19]. Liaskos et. al [9] propose a systematic way of eliciting goal models that appropriately explain the intentions behind existing systems. In [19], Yu et. al show how such models can be used to automatically configure relevant aspects of a complex system without accessing its source code. A configurator system that accepts a goal model and a user preference profile (in XML) and outputs a configuration for the target system is presented. The tool can have a GUI front-end and was used to configure Mozilla Firefox and Eclipse IDE.

2.4 From Goal Models to High-Variability Software Designs

We use goal models to represent variability in the way high-level stakeholder objectives can be met by the system-to-be together with its environment. Thus, goal models capture variability in the problem domain. However, properly augmented goal models can be used to create models that represent variability in the solution domain. We use textual *annotations* to add the necessary details to goal models. For example, the sequence annotation ("**;**") can be added to the appropriate AND goal decomposition to indicate that the subgoals are to be achieved in sequence from left to right. Sequence annotations are useful to model data dependencies or precedence constraints among subgoals. For instance, it is easy to see that the goal "Collect Timetables" must be achieved before achieving the goal "Choose Schedule" (see Figure 1). The absence of any dependency among subgoals in an AND decomposition can be indicated by a concurrency ("||") annotation. It is important to note that the above-mentioned annotations capture properties of the problem domain in more detail and are not used to capture design choices, so they are requirements-level annotations. However, annotations that can be applied to OR decompositions are usually more solution-oriented and indicate how (e.g., in parallel to save time or in sequence to conserve resources) the alternatives are to be attempted. We do not use this kind of annotations in this paper. Conditional annotations can also be added to specify that certain goals are to be achieved only under some specific circumstances. Lapouchnian and Lespérance [8] discuss more types of annotations. The choice of annotations to be used with goal models is influenced by the kinds of analysis or model transformations that one would like to carry out on goal models.

In [18], we described how one can gradually enrich basic goal models with appropriate information and produce the several types of models that preserve the variability captured in the goal models. Among the models produced are *feature models* and *statecharts*. These can serve as a starting point in the development of a design for a software system that can deliver the desired functionality in multiple ways.

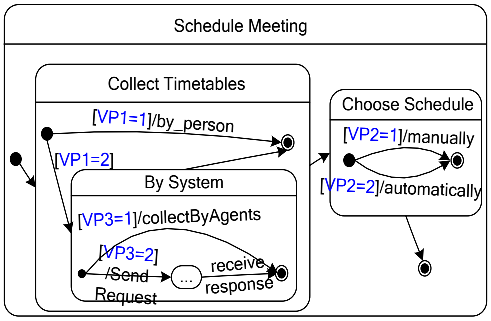

Figure 2: A fragment of the statechart generated from the goal model in Figure 1.

For example, to generate an initial statechart view (Figure 2) that models the behavioural variability of the system-to-be, for each goal the software system is responsible for a state that represents that goal being achieved by the system is introduced. We use super-/substates for organizing the states into a hierarchy that is isomorphic to the goal hierarchy from the source goal model. The generation of statecharts is based on a set of patterns that take into account goal decompositions and the temporal annotations that were used to enrich the original goal models. Here, the behaviour of the system depends on the selected process alternative in the corresponding goal model. Note that the conditions on some state transitions refer to the choices made for the variation points of the goal model (e.g., "VP1 = 2"). These conditions make sure that the choices in the goal model are reflected in the system behaviour. So, the variability of the goal model is preserved in the statechart. Note also, that in this approach, the selection of alternative system behaviours is externalized and should be handled by a specialized component (e.g., the configurator system as in [19]).

Thus, having a goal model representing the requirements for the system-to-be and the appropriate process-level enrichments, it is possible to generate initial design views that preserve the variability in the way the system-to-be can meet its objectives.

Overall, the approach of [18] is systematic and requirements-driven. It allows for the gradual increase of the level of detail of the goal models through the use of annotations. This process maps requirements goal models into solution domain models that can either be utilized as high-level design specifications or used to generate other, more elaborate design-level models of the system. In this approach, requirements traceability is supported through the tight mapping between notations.

Alternatively, a script can be generated from a goal model (e.g., as done in [8] for agent-based systems) that can be used for integration/orchestration of components designed to achieve leaf-level goals. In this case, the variability will be preserved in the script rather than implemented by the components of the system.

3 Towards Goal-Driven Autonomic Computing

In this section, we describe how goal models can be helpful in designing autonomic application software, outline the architecture for AC systems that can be easily derived from goal models, and describe our requirements-driven approach for developing autonomic systems

3.1 The Role of Goal Models

The building blocks of autonomic computing are architectural components called *Autonomic Elements* (AEs). An autonomic element is responsible for providing resources, delivering services, etc. Its behaviour and its relationships with other AEs are "driven by goals that its designer embedded in it" [6]. An AE typically consists of an autonomic manager and a set of managed elements, such as resources, components, etc. The manager must be able to monitor and control the managed elements.

An autonomic element manages itself to deliver its service in the best possible way. In order to achieve this, its autonomic manager must be armed with tools for monitoring its managed elements and the environment, for analyzing the collected data to determine whether the AE is performing as expected, for planning a new course of action if a problem is detected, and for executing these plans by, for example, tuning the parameters of its managed elements. Most importantly, these activities require the knowledge about the goal of the autonomic element, the configurations and capabilities of its managed elements, the environment of the AE, etc.

We believe that goal models can be useful in the design of autonomic computing systems in several ways. First, goal models are used to capture and refine requirements for autonomic systems. A goal model provides the starting point for the development of such a system by analyzing the environment for the system-to-be and by identifying the problems that exist in this environment as well as the needs that the system under development has to address. Thus, requirements goal models can be used as a baseline for validating software systems.

Second, goal models provide a means to represent many ways in which the objectives of the system can be met and analyze/rank these alternatives with respect to stakeholder quality concerns and other constraints, as described above. This allows for exploration and analysis of alternative system behaviours at design time, which leads to more predictable and trusted autonomic systems. It also means that if the alternatives that are initially delivered with the system perform well, there is no need for complex social interactions among autonomic elements (e.g., as implied in [6], where AEs are viewed as socially-capable intelligent agents). Of course, not all alternatives can be identified at design time. In an open and dynamic environment, new and better alternatives may present themselves and some of the identified and implemented alternatives may become impractical. Thus, in certain situations, new alternatives will have to be discovered and implemented by the system at runtime. However, the process of discovery, analysis, and implementation of new alternatives at runtime is complex and error-prone. By exploring the space of alternative process specifications at design time, we are minimizing the need for that difficult task.

Third, goal models provide the traceability mechanism from AC system designs to stakeholder requirements. When a change in stakeholder requirements is detected at runtime (e.g., by using the approach in [2]), goal models can be used to re-evaluate the system behaviour alternatives with respect to the new requirements and to determine if system reconfiguration is needed. For instance, if a change in stakeholder requirements affected a particular goal in the model, it is easy to see how this goal is decomposed and which components/autonomic elements implementing the goal are in turn affected. By analyzing the goal model, it is also easy to identify how a failure to achieve some particular goal affects the overall objective of the system. At the same time, high-variability goal models can be used to visualize the currently selected system configuration along with its alternatives and to communicate suggested configuration changes to users in high-level terms.

Fourth, goal models provide a unifying intentional view of the system by relating goals assigned to individual autonomic elements to high-level system objectives and quality concerns. These high-level objectives or quality concerns serve as the common knowledge shared among the autonomic computing elements to achieve the global system optimization. This way, the system can avoid the pitfalls of missing the globally optimal configuration due to only relying on local optimizations.

3.2 A Hierarchical Autonomic Architecture

We now outline the architecture for autonomic software systems that can be derived from high-variability requirements goal models. We envision a hierarchy of autonomic elements that is structurally similar to the goal hierarchy of the corresponding goal model. Here, leaf-level goals are to be achieved by the components of the system-to-be, by legacy systems, or by humans. Higher-level goals are used to aggregate the lower-level ones all the way to the root goal. Additional information such as softgoal contributions and annotations is used to determine the best configuration of the system for achieving its main goal.

In the most straightforward case, a single autonomic element is responsible for the whole system. Thus, it is associated with the whole goal model and is said to achieve the root goal of the model. This has certain advantages in that all of the analysis, monitoring, etc. is done in one place, which can be helpful in achieving globally optimal performance. However, there are also potential problems with this approach. A single AE can make the system quite inflexible, hard to maintain, as well as make it impossible to reuse any part of the system.

In the other extreme case, each goal in the goal model can be associated with an autonomic element whose purpose is the achievement of that goal. The managed elements of the leaf-level autonomic elements (which correspond to leaf-

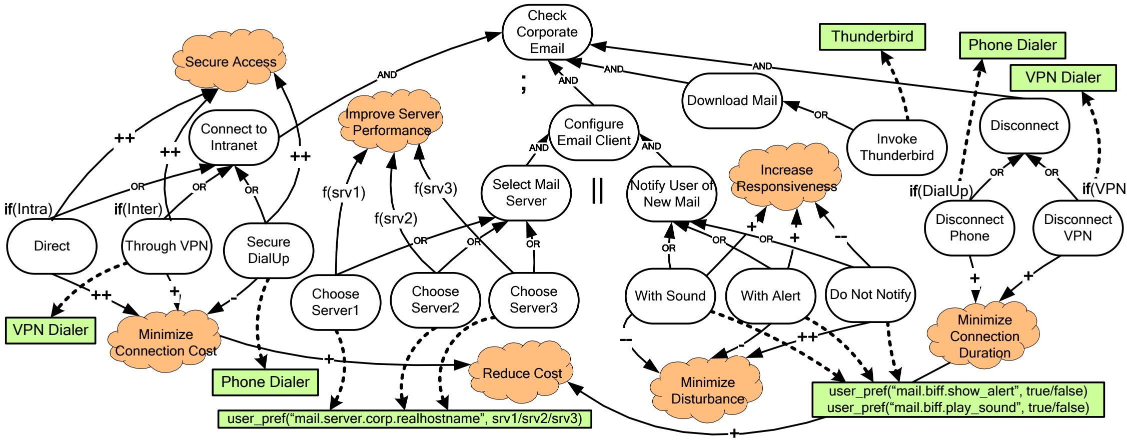

Figure 3: The Check Email example.

level goals) are then the actual components, resources, etc. Leaf-level AEs can tune and optimize these resources to deliver their objective in the best way. On the other hand, higher-level autonomic elements are not directly associated with the actual components, but are used to orchestrate the lower-level elements. The root autonomic element represents the whole software system. Thus, an AE corresponds to any subtree of the goal model. This approach has an advantage that the global high-variability design space is partitioned into autonomic elements with lower variability, thereby facilitating management and administration tasks. Also, this will improve maintainability of the system. Finally, there is the middle ground where a goal model is partitioned among a set of AEs, each of which is responsible not for a single goal, but for a goal subtree.

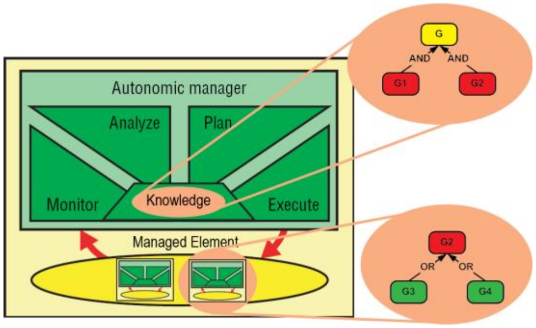

Figure 4: A hierarchical composition of AEs.

It remains to be seen which strategy is the best for partitioning a goal model among autonomic elements. The size of the model is an important factor here.

A fragment of a properly enriched goal model will serve as the core of each AE's knowledge. For example, Figure 4 presents an AE, whose objective is to achieve the goal G. It has a fragment of the goal model showing the decomposition of this goal. Here, the goal G is AND-decomposed into G1 and G2, which means that the goal model identified only *one* way to achieve G. The managed elements of the AE in Figure 4 are themselves autonomic elements that achieve the goals G1 and G2. They have different fragments of the goal model assigned to them. For example, the AE achieving the goal G2 knows that to attain that goal it must either achieve G3 or G4 (the relevant softgoals are not shown). These goals can be in turn handled by lower-level AEs (also not shown).

Because of the hierarchy of AEs, it is possible to propagate high-level concerns from the root AE down to the leaf-level elements, thus making sure that the system achieves its objectives and addresses the quality concerns of its stakeholders. Note that each autonomic element retains the freedom to achieve its goal in the best way it can provided that it satisfies the constraints passed to it by the higher-level AE and/or the user.

3.3 An Illustrative Example

Before we describe our approach in detail, let us introduce a subset of a case study we did to evaluate it (see Figure 3). In the next sections we will be referring to parts of this example to illustrate the steps of the process. The example is a system designed to be used with Mozilla Thunderbird email client to periodically download email from a corporate email server, thus "Check Corporate Email" is its goal. First, the system needs to connect to the secure corporate *intranet*, which can be achieved by either connecting to it directly (through the office network), by using the virtual private network (VPN) connection, or by using a secure dial-up provider. All three ways are considered secure (note the contributions to the "Secure Access" softgoal), but have different costs. Rectangular shapes in the model show how leaf-level goals are implemented. For example, the achievement of "Through VPN" goal is delegated to an existing VPN dialer component. Then, the system configures Thunderbird to use the best email server available by selecting among the three available corporate servers. This is done by automatically changing the configuration file of Thunderbird (specifically, the parameter `mail.server.corp.realhostname`). Also, depending on whether the user prefers not to be disturbed or, conversely, prefers to be very responsive, the system configures Thunderbird to display a visual alert, play a sound, or do nothing when new mail arrives. After that, the system invokes Thunderbird and later disconnects from the intranet to reduce connection costs. As you can see, the example system delivers its functionality by integrating and appropriately configuring existing components.

4 The Approach

In our approach for the development of autonomic software, we take high-variability requirements-level goal models as a starting point. They are used to capture the needs for the new system, both functional and non-functional and the alternatives that exist in the problem domain for meeting those needs, as well as to do the initial analysis of the alternatives with respect to the important quality criteria modeled as softgoals.

A lot of research in the Autonomic Computing area is currently devoted to methods and techniques for developing AC managers that handle IT resources shared among applications. These resources are usually various kinds of servers that can be dynamically allocated to applications that require them. So, the job of these AC managers is to optimize the use of their resources, to protect them, etc. Therefore, they operate in fairly restricted environments (e.g., data centres) and their decisions are implemented in terms of a relatively small set of actions that are available in these domains. This makes the AC managers quite generic (i.e., middleware-like). Moreover, most of the activities of these managers are hidden from the applications since they are quite low-level and thus do not affect these applications in a profound way. All of these characteristics make the field of resource allocation and provisioning ripe for automation.

We, on the other hand, believe that resource allocation/provisioning is just one of the areas that can benefit from autonomic computing ideas and that these ideas can be applied to systems other than AC managers – specifically, to applications themselves. Therefore, our approach is meant to be used to introduce autonomic behaviour into the application software, thus making it more flexible and robust in achieving its goals.

There are a number of ways in which autonomic application software differs from autonomic middleware. First, the autonomic functionality is application-specific, not generic. Second, the changes in the autonomic application behaviour are usually visible to and have direct effect on the user and thus might require his explicit approval and his trust. Third, the autonomic behaviour of an application system is highly influenced by the preferences and priorities of its users.

The above discussion suggests that autonomic application software requires special development methodologies that address its unique characteristics. Thus, the approach presented here that is rooted in software requirements engineering and provides a way to explicitly model and analyze alternative behaviours and how they affect user quality concerns seems a promising way for building autonomic application software.

In this approach, users (perhaps, non-technical) can be in the loop by approving the software changes proposed by the autonomic system as well as by driving them by the means of, for example, shifting priorities from one non-functional concern to another. As noted before, the approach leads to more predictable and trusted systems and thus can be used for developing mission-critical systems with autonomic behaviour where changes in the system's behaviour might have to be approved by an appropriate person before they are enacted. Goal models can help with such user interaction since they explicitly represent goal achievement alternatives as well as are able to present them in high-level terms.

Since the approach relies on the manual elicitation of high-variability goal models, it may not be suited for domains that need very large number of goals. However, once the goal model is developed, the alternatives can be enumerated and analyzed automatically. For example, [4] shows that even naïve algorithms can work reasonably well on a goal model with 750 nodes and 10^6 alternatives.

We now describe the main steps in the process in more detail.

4.1 Developing Goal Models

The process starts by identifying the problem domain, together with the opportunities for its improvement. Specifically, we look at how the introduction of a software system can improve the situation in the domain. The *i** notation [17] can be used at this stage to model stakeholders in the domain along with their needs. This early requirements stage helps us in identifying the goals that the system-to-be will have to achieve. Once the goals of the system are clear, we use goal models to capture and refine them using AND/OR decompositions described in Section 2. The emphasis here is on modeling the variability in the problem domain: we try to capture all the different ways the system's goals can be achieved in that domain. The process for high-variability goal model elicitation described in [10] can help with this task. We refine the goals of the model until we reach the ones that can easily be achieved through developing a software component, delegating the goal to an existing component, a legacy or a COTS (Commercial Off-The-Shelf) system, or a person. Also, as we can see in the Check Email system, some goals can be achieved by appropriately configuring COTS systems.

In our example in Figure 3, the goal of the system is "Check Corporate Email". This goal is refined into subgoals with alternative refinements (e.g., the way one can connect to the corporate intranet) represented by OR decompositions. We stopped the refinement once we identified the goals that could be achieved by the existing COTS systems such as Mozilla Thunderbird, a VPN dialer, or by appropriately configuring the COTS products used in the system.

Non-functional constraints are used for analyzing the alternatives and for selecting the best option for the system's behaviour. They are captured using softgoals in our goal models, so one of the key activities during the elicitation of goal models is to identify the quality constraints that are important in the problem domain. In the Check Email example, the softgoals include "Improve Server Performance", "Increase Responsiveness", and "Minimize Disturbance". Note that the latter two have the generally opposite contributions from the alternative ways of notifying the user of new email messages: the goal "Do Not Notify" breaks (--) the softgoal "Increase Responsiveness" while making (++) "Minimize Disturbance". Thus, the selection of the best notification alternative will depend on how the user prioritizes among these quality constraints. A change in such prioritization will trigger a reconfiguration of the system.

While eliciting goal models, we also add the necessary sequential and parallel annotations as described in Section 2.4. For instance, in the Check Email example, the goal of connecting to the intranet must be achieved before the goal of downloading mail. Similarly, the two aspects of the email client configuration, namely the mail server and the type of new email alert can be done independently, thus the goal "Configure Email Client" is used with the parallel annotation.

4.2 Adding Formal Details

While some GORE approaches (e.g., KAOS [1]) require formal specifications for all goals in goal models, in our approach it is up to the user to determine to what extent the model must be formalized. This means that if automated planning is a feature of the system, then all the goals will most likely be formally specified. Otherwise, the system specification can mostly remain informal. For example, in Figure 3 we only specify preconditions for goals *as needed* by using conditional annotations `if(condition)`. Specifically, in Figure 3 the goal "Connect to Intranet" is OR-decomposed into the goals "Direct" and "Through VPN" referring to the ways one can connect to a corporate intranet. The precondition for "Through VPN" is `Inter`, which is a boolean variable that is true whenever there is internet connectivity (since you have to have the internet connection to be able to use VPN). The precondition for the direct intranet connection is the existing intranet connectivity. Preconditions capture the domain properties that must be true for alternatives to be considered for selection. For instance, if the system has only internet (but not intranet) connectivity, then the "Direct" option is not available, while the VPN and dial-up options are. When multiple alternatives are available, quality criteria (in this case, “Minimize Connection Cost” softgoal) will be used to select the best one.

Likewise, the two alternatives for the “Disconnect” goal, namely “Disconnect VPN” and “Disconnect Phone”, have as preconditions the VPN and dial-up connectivity respectively. Obviously, one can disconnect a dial-up connection only if it has been previously established. Thus, the boolean variable `DialUp`, a precondition for “Disconnect Phone”, must capture the effect (post-condition) of the goal “Secure Dial-Up”. The same applies to the variable `VPN` and the goal “Through VPN”. Therefore, when a VPN connection is established, it will be disconnected by achieving the goal “Disconnect VPN”. The preconditions create requirements for the monitoring component of the system.

4.3 Specifying Softgoal Contributions

In goal models, goals/softgoals can be related to softgoals through help (+), hurt (–), etc. relationships. They represent qualitative evaluations of how particular alternatives affect the modeled non-functional requirements. Many of these do not change throughout the execution of the system. For instance, in Figure 3, the goal “Do Not Notify” [of incoming messages] *makes* (++) the softgoal “Minimize Disturbance”, while the goal [notify] “With Alert” hurts it. This captures the understanding that any alert is a distraction. And this is unlikely to change. On the other hand, there are situations where one would like to model softgoal contributions not as constants, but as functions. In the Check Email example, such softgoal is “Improve Server Performance”. Suppose that the chosen way to improve email server performance in the corporate system is to make email clients connect to servers with the lowest current workload. Since server workload, obviously, varies, to pick the server with the lowest load we must parameterize the contributions to the softgoal “Improve Server Performance” as, for example, done in [9]. To preserve uniformity in treating softgoals and thus to still allow the use of the previously mentioned goal reasoning algorithms, we define the function `f(srv)` (where `srv` is the name of the email server), which maps certain server workload ranges into the already discussed four contribution labels. Here, we assume that maximum server load is 999 concurrent connections. The function is defined through the sensed value `load(srv)`, the current load on the server `srv`.

$$f(srv) = \begin{cases} \text{"++"}, \text{if } \mathit{load(srv)} \leq 300 \\ \text{"+"}, \ \text{if } 300 < \mathit{load(srv)} \leq 600 \\ \text{"–"}, \ \text{if } 600 < \mathit{load(srv)} \leq 800 \\ \text{"––"}, \text{if } 800 < \mathit{load(srv)} \leq 999 \end{cases}$$

Since softgoals represent quality concerns (non-functional requirements), not all of them can be automatically determined to be satisfied or not. It is even harder is assign values to softgoals thus turning them into quantitative entities. Still, many softgoals can be *metricized* – assigned metrics that approximate those concerns. The handling of the goal “Improve Server Performance” above is an example of metricizing a softgoal. Similarly, a popular metric for reliability is *mean time between failure* is another example. There are many examples of such well-understood metrics that can be used to approximate profitability, reliability, performance, etc. However, not all softgoals can be metricized. For example, “Convenience” is a highly subjective criterion.

In general, in order to metricize a softgoal one needs to come up with a *measurable* function approximating that softgoal. Additionally, based on the usual four-valued system for softgoal contribution we use in our goal models the range of the function has to be partitioned into four subranges, each corresponding to the contribution value from “--“ to “++” as done in the example above.

4.4 Monitoring

For a system to exhibit autonomic behaviour, it must be able to monitor its environment as well as its own behaviour to detect changes, failures, etc. Appropriately enriched goal models described in the previous sections can help in determining what information needs to be captured and analyzed by the system.

First of all, the system must be able to monitor the achievement of its leaf-level goals. These are the goals that are assigned to the system components, or the environment of the system (i.e., legacy systems, humans, etc.) In Requirements Engineering, the latter are viewed as the system’s *expectations* of its environment and so an autonomic system must monitor the achievement of these goals in order to detect if the expectations are still valid.

The monitoring can be done in various ways. If a goal is assigned to a legacy system or a component, it might be possible to query that system/component to get the status of the goal. Otherwise, sensors in the environment can be used to determine if the goal has been achieved without querying the involved component(s). For instance, in the example in Figure 3, after a VPN dialer has been invoked to achieve the goal [connect to intranet] "Through VPN", we used a simple sensor to determine if access to the internal corporate network had been granted by `ping`-ing a known intranet server.

The achievement status of non-leaf goals can usually be deduced using the algorithm of [3] that propagates the satisfaction values of leaf-level goals up towards the root of the model. For example, if the goal "Through VPN" is determined to be achieved, then the goal "Connect to Intranet" is achieved as well by the semantics of the OR decomposition.

The environment of the system also needs to be monitored to determine if preconditions for goals are satisfied. In the Check Email example, the goal [connect to intranet] "Through VPN" requires internet connectivity. Again, a simple `ping`-based sensor is used to determine that. The boolean variable `Inter` used within a conditional annotation applied to "Through VPN" is defined with the help of this sensor. Frequently, a precondition of one goal is the achievement of another. For example, a VPN connection must exist before one can disconnect it. So, the precondition for "Disconnect VPN", the boolean variable `VPN`, is, in fact, the post-condition of "Through VPN". It can be tested as described above.

Since some non-functional requirements (modeled as softgoals) can be metricized using approximating functions, to calculate the values for these functions, we need to capture the data used in their definitions. For example, to evaluate the satisfaction of the softgoal "Improve Server Performance" (as defined in Section 4.3) the Check Email system needs to monitor the current server load value `load(srv)` for all email servers.

As already mentioned, many softgoals are too high-level/subjective to be metricized. Thus, it is not straightforward for the system to, for example, automatically verify that a particular alternative's contribution to a softgoal is correctly captured in the goal model (e.g., that an alternative, in fact, contributes negatively to the softgoal "Convenience"). In these cases, the system might want to confirm with the user(s) that its current configuration meets the users' quality criteria.

4.5 Using COTS Systems

COTS or legacy systems can be given responsibility for achieving goals. This can be done in the usual way through procedure calls, messages, etc. However, another possibility for using legacy software in autonomic systems is through goal-driven configuration [9][19] where AEs will wrap these systems making sure that their behaviour conforms to the quality preferences of system's stakeholders. The use of Thunderbird in our Check Email case study is an example of that. Here, Thunderbird is being dynamically configured to achieve the functional goal "Download Mail" while meeting non-functional requirements such as "Improve Server Performance". This configuration approach has limitations since it depends on the richness of configuration options of legacy systems. However, many complex systems have vast possibilities for configuration yielding thousands or millions of alternatives with very different properties that can be utilized in our approach.

When applied to COTS/legacy systems, our approach can be viewed as defining the infrastructure for flexible, yet predictable integration of these systems to meet higher-level customer needs.

4.6 Goal Model-Based Autonomic Behaviour

Given a goal model characterizing various ways of achieving some root goal G, one can rank these alternatives with respect to their satisfaction of the partially ordered set of quality criteria represented in the model by softgoals. For example, in the Check Email case study, if the softgoal "Minimize Disturbance" is of high priority, then any alternative that uses sound notification when new mail arrives will be ranked lower than any alternative that uses the display notification. Whenever the system needs to switch from one configuration to another, it tries to select the best new alternative that achieves the objective of the system while maximizing the achievement of the set of quality constraints (softgoals).

4.6.1 Self-Configuration

In our approach, when the system is first deployed, it is configured to execute the best alternative for the given (initial) preferences over softgoals. It should continue to execute the chosen alternative until changes in the environment of the system or changes in softgoal priorities invalidate it. If this happens, the system should be reconfigured and the new best alternative must be chosen. For example, in Figure 3, the default means for establishing the intranet connection is the "Direct" connection since it, unlike the other choices, has a "make" (++) contribution to the softgoal "Minimize Connection Cost". Therefore, as the user of the system keeps checking his email while being connected in the office (the precondition `Intra` always holds in this case), the "Direct" option will remain selected. However, if the user tries to check the corporate email from home using his own internet provider, the monitoring component will detect the internet, but not the *intranet* connectivity. Therefore, the precondition for the "Direct" option will not be satisfied and a reconfiguration will be needed. In this case, both of the remaining alternatives will be available since their preconditions are satisfied. The autonomic manager responsible for that part of the system will then use the goal reasoning algorithm of [13] to find an alternative that achieves the goal "Connect to Intranet" while making the best contribution to the softgoal "Minimize Connection Cost". That alternative adopts the goal "Through VPN". This is an example of software reconfiguration based on a change in the environment of the system.

A similar switch from one configuration to another will happen due to the change in user priorities regarding email notification (the softgoals "Increase Responsiveness" and "Minimize Disturbance"). These changes cannot be easily detected as they are normally related to the user's mood, workload, etc. Thus, the user must be able to notify the system about such changes proactively, through the use of a GUI tool. In the case study we used a simple tool (presented in [19]) that allowed users to set priorities over softgoals for the system. Once the user's input is received, the best choice for "Notify User of New Mail" based on the user's new priorities is found as above with the help of a reasoning algorithm. Therefore, in our approach, both the user and the system's environment can cause self-reconfiguration.

4.6.2 Self-Optimization

The email server configuration in Mozilla Thunderbird in our Check Email example is designed to show how self-optimization can be done in our approach (see Figure 3). When the system is first deployed, the values `load(srv1)` through `load(srv3)` are fetched using a simple monitoring component querying the server status database. The contribution values for the servers are then calculated and the server with the lowest workload is chosen. During the subsequent runs of the system new workload values are received and the contributions to "Improve Server Performance" are recalculated. If applicable, a different server is chosen. Since the formula *f* produces only four discrete values for the softgoal contributions, the system will not be able to always select the server with the lowest workload because the reasoning algorithm will not be able to distinguish among servers with relatively similar workloads and thus the same contribution labels. A finer-grained approach is, of course, possible (e.g., one use numerical softgoal contribution values).

4.6.3 Self-Healing

A failure of a software component, COTS/legacy system, or human to achieve a goal delegated to them forces the system to search for ways to heal itself. Using one of the already mentioned goal analysis algorithms [3], the system will propagate the "denied" status of the failed leaf-level goal up the goal model to determine which higher-level goals will in turn be affected by this failure. This failure propagation can be presented to the user/administrator of the system to illustrate the severity of the problem by showing the problematic system parts. The "top-down" goal reasoning algorithm [13] is then used to find a new system configuration that satisfies the top-level goal of the system and as many of the non-functional requirements as possible.

We will now illustrate this using the example in Figure 3. Obviously, a failure of any child of an AND-decomposed goal will propagate to its parent. So, in our Check Email example a failure to establish an intranet connection automatically denies the top-level goal "Check Corporate

Email". In this case, the model has no alternative capable of achieving the top goal.

On the other hand, all of the children of an OR-decomposed goal must fail for it to be denied. For example, in, if the goal [notify user of new email] "With Sound" fails, then its parent goal "Notify User of New Mail" can still be attained since there exist other alternatives for its achievement. From the two possibilities, "With Alert" and "Do Not Notify", and assuming that the user prefers the softgoal "Increase Responsiveness", the algorithm of [13] will select "With Alert" as the new alternative contributes positively to that softgoal (unlike "Do Not Notify").

5 Discussion

Kephart and Chess suggest that overall system self-management results from the internal self-management of its individual autonomic elements [6]. Moreover, in their view, autonomic elements are full-fledged intelligent agents that, when assigned individual goals, will use complex social interactions to communicate, negotiate, form alliances, etc. and ultimately deliver the objective of an autonomic system. However, deriving a set of goals and policies that, if embedded into individual autonomic elements, will guarantee certain global system properties is nontrivial. Thus, there needs to be a systematic way of capturing overall system's objectives, decomposing them into lower-level goals, and assigning those goals to AEs. This problem is not addressed in [6]. The approach presented here is requirements-driven and can be used to systematically derive goals for individual AEs/agents given the overall goals of the system.

Multiagent systems promise to provide a very flexible, scalable and open platform for software applications. However, the cost of introducing agent infrastructures that rely on complex interaction protocols, planning, etc. may outweigh their benefits in the domains where, for example, well-understood performance models already exist and can be used for automated optimization of software systems. At the same time, there are also concerns that a fully agent-based solution may not be acceptable in certain domains such as mission critical systems, business support systems, etc. where predictability, reliability and transparency are of paramount importance. Similarly, trust can be a major issue in the acceptance of AC systems. We believe that while being less flexible, our methodology provides the capability to analyze important process alternatives thus increasing the system's predictability and transparency while improving the users' trust in it.

In [8], an agent-oriented requirements engineering method is introduced that translates *i** models (which are a superset of the goal models described here) into high-level formal agent specifications that support formal representation of and reasoning about goals and knowledge of agents. That approach is similar to the one presented here in the sense that it is requirements-driven and uses a similar goal-oriented notation. However, the method of [8] does not emphasize the variability aspect of goal models as much as we do here. Therefore, we view the two techniques as complementary to each other. By allowing leaf-level goals in our approach to be delegated to intelligent software agents, we will help with the design of systems that support a set of previously analyzed and trusted alternatives and do not require complex multiagent infrastructures as long as one of the identified alternatives can be applied. In situations when no alternative is satisfactory, the full capabilities of intelligent software agents such as the ability to reason about their goals, to communicate with each other at a semantic level, to dynamically form teams, etc. can be invoked. We plan to work on such hybrid approach in the future.

6 Conclusion

The essential characteristic of autonomic computing systems is their ability to change their behaviour automatically in case of failures, changing environment conditions, etc. In this paper, we outline an approach for designing autonomic computing systems based on goal models that represent *all* the ways that high-level functional and non-functional stakeholder goals can be attained. These goal models can be used as a foundation for building software that supports a space of behaviours for achieving its goals and that is able to analyze these alternatives (with respect to important quality and other criteria), its own state, and its environment to determine which behaviour is the most appropriate at any given moment. For such systems, goal models provide an intentional view unifying all the system components and demonstrating how they must work together to achieve the overall system objective. Goal models also support requirements traceability thus allowing for the easy identification of parts of the system affected by changing requirements. When properly enriched with relevant design-level information, goal models can provide the core architectural, behavioural, etc. knowledge for supporting self-management. Of course, an appropriate monitoring framework as well as, perhaps, learning mechanisms need to be introduced to enable self-management. The use of our approach with intelligent software agents is also possible.

The benefits of this method also include the increase in predictability and transparency of systems as well as the users' trust in them.

Presented here is a vision for the requirements-driven design of autonomic software. A lot of work remains to be done to test the applicability of this approach and its scalability (though there is evidence that automated reasoning can be done on very large goal models). Heuristics need to be developed for decomposing the system into a hierarchy of autonomic elements. A particularly interesting research area is the integration of agents into the approach. This way the system will be able to come up with new alternatives for meeting its objectives whenever the predefined configurations fail. We are also working on larger case studies, particular in the area of adaptive business processes and patient care.

About the Authors

Alexei Lapouchnian is a Ph.D. candidate in the Department of Computer Science at the University of Toronto. He received his M.Sc. from York University. His research interests are in requirements engineering, business process modeling, autonomic computing, and software agents.

Yijun Yu is a research associate in the Department of Computer Science at the University of Toronto. He holds a Ph.D. from Fudan University in China. He is interested in quality aspects of software engineering, autonomic computing, software reengineering, and software performance tuning.

Sotirios Liaskos is a Ph.D. candidate in the Department of Computer Science at the University of Toronto. He received his M.Sc. from the University of Toronto. His main interest is in the area of Goal-Oriented Requirements Engineering, in particular in capturing and analyzing variability in software requirements.

John Mylopoulos holds a Ph.D. from Princeton University (1970) and is a professor of Computer Science at the University of Toronto. His research interests include requirements engineering, data semantics and knowledge management.

References

[1] A. Dardenne, A. van Lamsweerde and S. Fickas. Goal-Directed Requirements Acquisitions, *Science of Computer Programming,* 20:3-50, 1993.

[2] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In Proc. *9th International Workshop on Software Specification and Design*, p. 50. IEEE Computer Society, 1998.

[3] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, R. Sebastiani. Reasoning with Goal Models. In Proc. *21st International Conference on Conceptual Modeling (ER2002),* Tampere, Finland.

[4] B. Hui, S. Liaskos, and J. Mylopoulos. Requirements Analysis for Customizable Software: Goals-Skills-Preferences Framework. In Proc. *11th IEEE International Requirements Engineering Conference (RE'03)*, Monterrey, CA, pp. 117–126, September 2003.

[5] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study (CMU/SEI-90-TR-21, ADA235785). Technical Report, SEI/CMU, 1990.

[6] J. Kephart and D. Chess. The vision of autonomic computing, *Computer,* 36(1):41–50, 2003.

[7] A. Lapouchnian, S. Liaskos, J. Mylopoulos, Y. Yu. Towards Requirements-Driven Autonomic Systems Design. In Proc. *ICSE 2005 Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005),* St. Louis, Missouri, USA, May 21, 2005. ACM SIGSOFT Software Engineering Notes 30(4), July 2005.

[8] A. Lapouchnian and Y. Lespérance. Modeling Mental States in Agent-Oriented Requirements Engineering. In Proc. *18th Conference on Advanced Information Sys-*

tems Engineering (CAiSE'06), Luxembourg, June 5-9, 2006.

[9] S. Liaskos, A. Lapouchnian, Y. Wang, Y. Yu, S. Easterbrook. Configuring Common Personal Software: a Requirements-Driven Approach. In Proc. *13th IEEE International Requirements Engineering Conference*, Aug 29-Sep 2, 2005, Paris, France.

[10] S. Liaskos, A. Lapouchnian, Y. Yu, E. Yu, J. Mylopoulos. On Goal-based Variability Acquisition and Analysis. In Proc. *14th IEEE International Requirements Engineering Conference*, Minneapolis, USA, Sep 11-15, 2006

[11] R. Murch. Autonomic Computing. Prentice Hall, 2004.

[12] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using non-functional requirements: a process-oriented approach, *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[13] R. Sebastiani, P. Giorgini, J. Mylopoulos. Simple and Minimum-Cost Satisfiability for Goal Models. In Proc. *CAiSE 2004*, Riga, Latvia.

[14] W. Spears, K. De Jong, T. Baeck, D. Fogel, H. Garis. An Overview of Evolutionary Computing. In Proc. *European Conference on Machine Learning*, 1993.

[15] K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic Service Matchmaking among Agents in open Information Environments, *ACM SIGMOD Record*, Special Issue on Semantic Interoperability in Global Information Systems, A. Ouksel, A. Sheth (Eds.), 28(1):47–53, 1999.

[16] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. Proc. *ICSE'00*, Limerick, Ireland, June, 2000.

[17] E. Yu. Modeling Organizations for Information Systems Requirements Engineering. In Proc. *1st IEEE International Symposium on Requirements Engineering*, San Diego, CA, 1993, pp. 34-41.

[18] Y. Yu, J. Mylopoulos, A. Lapouchnian, S. Liaskos, and J.C.S.P. Leite. From stakeholder goals to high-variability software designs. Technical Report CSRG-509, University of Toronto, 2005. Available at: ftp://ftp.cs.toronto.edu/csrg-technical-reports/509/.

[19] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos. Requirements-Driven Configuration of Software Systems. In *WCRE 2005 Workshop on Reverse Engineering to Requirements (RETR'05)*, Pittsburgh, PA, USA, November 7, 2005.