Self-Tuning of Software Systems through Dynamic Quality Tradeoff and Value-based Feedback Control Loop

Xin Peng[1], Bihuan Chen[1], Yijun Yu[2], Wenyun Zhao[1]
[1] School of Computer Science, Fudan University, Shanghai 201203, China
[2] Department of Computing, The Open University, Milton Keynes, UK
{pengxin, 09210240005, wyzhao}@fudan.edu.cn, y.yu@open.ac.uk

Abstract. Quality requirements of a software system cannot be optimally met, especially when it is running in an uncertain and changing environment. In principle, a controller at runtime can monitor the change impact on quality requirements of the system, update the expectations and priorities from the environment, and take reasonable actions to improve the overall satisfaction. In practice, however, existing controllers are mostly designed for tuning low-level performance indicators instead of high-level requirements. By maintaining a live goal model to represent runtime requirements and linking the overall satisfaction of quality requirements to an indicator of earned business value, we propose a control-theoretic self-tuning method that can dynamically tune the preferences of different quality requirements, and can autonomously make tradeoff decisions through our preference-based goal reasoning procedure. The reasoning procedure results in an optimal configuration of the variation points by selecting the right alternative of OR-decomposed goals and such a configuration is mapped onto corresponding system architecture reconfigurations. The effectiveness of our self-tuning method is evaluated by earned business value, comparing our results with those obtained using static and ad-hoc methods.
Keywords: Feedback Control Theory; Preference; Goal-Oriented Reasoning; Self-Tuning; Earned Business Value

1. Introduction

Considering alternative architectures and designs, different quality attributes require different optimisations, and often the one better in certain quality dimensions is worse in others (Kazman et al., 2000). Such multi-objective optimisation problems aim to find a set of *Pareto optimal* solutions. A solution is called "Pareto optimal" if there is no other solution better than or equal to it in one of the objectives (Lukasiewycz et al., 2007). Hence, Pareto optimal solutions define a partial ordering, as opposed to the total ordering. In software engineering, tradeoff decisions for multiple quality requirements must take into account multiple factors such as stakeholders' expectations, preferences and priorities (Mylopoulos et al., 2001). In addition, these factors are likely to change in an uncertain and dynamic environment, leading to the necessity to adapt the "optimal" decisions accordingly at runtime.

For example, two conflicting quality requirements such as usability and performance can have quite different priorities in different situations. In "normal" situations, usability measures such as "themed look and feel" can be adopted without having apparently negative influence on performance. However, if the system is overloaded by large number of concurrent requests, its performance should be given a higher priority over its usability. Under such circumstances, it is reasonable to relax the satisfaction level of usability to ensure an acceptable satisfaction level of performance. Otherwise, a drastic degradation of performance will greatly damage the overall quality of the system: few would be able to use an online shopping system, however user-friendly the system is, with a response time of over 5 minutes for a simple operation, e.g., adding products to shopping cart.

Once deployed, software systems in production are no longer as easy to configure and change as they were inside the development houses. Therefore, *self-tuning* mechanisms with little human intervention are more practical to find Pareto optimal configurations for dynamic quality tradeoff. Self-tuning is a kind of self-management capabilities (Kephart and Chess, 2003). A general reference architecture for self-managing systems has been described by Cheng et al. (Cheng et al., 2009b) and Kramer et al. (Kramer and Magee, 2009), which separates the design of such systems into three vertical layers for goals, plans and components. Starting from the highest level, a requirements-driven system for applications is desired (Lapouchnian et al., 2005, 2006, 2007) in the goal layer. Planning algorithms are applied to obtain from the high-level expectations a set of low-level components that can fulfil the requirements (Giorgini et al., 2002; Sebastiani et al., 2004) in the plan layer. At the lowest level, i.e., the component layer, the reference architecture emphasizes the reuse of existing components (Peng et al., 2009), and the generation of highly configurable design (Yu et al., 2008).

Instead of using high-level requirements, however, most existing self-tuning mechanisms in the autonomic computing field except some architecture-based methods like Rainbow (Cheng, 2008) were using low-level performance indicators such as load, memory, bandwidth, and even characteristics of user profiles such as operational error rates (Kephart and Chess, 2003). There is clearly an uncertain gap between goal satisfaction and partially quantifiable metrics (Letier and Lamsweerde, 2004). Living with such uncertainty at runtime, several researchers proposed formal approaches to compensate, e.g., by analysing the problem context through monitoring the changing environment (Fickas and Feather, 1995; Salifu et al., 2007), or by diagnosing the root causes of components for failed goals (Wang et al., 2009).

To link the satisfaction of high-level requirements down to the tuning parameters of the running systems, one key idea is to integrate stakeholder value considerations into runtime self-tuning decisions following the general principles of value-based software engineering (VBSE) (Boehm, 2006). The objective of self-tuning can be interpreted as to maximise the value proposition of stakeholders from the value-based perspective. Thus, if we can measure earned business value based on runtime data, the value measurement can be used as a quantitative indicator for the overall quality satisfaction.

The other key mechanism is to consider using feedback loops as proposed in software cybernetics (Cai et al., 2003). Earlier we have presented a framework to optimise software quality at runtime (Chen et al., 2009). The self-tuning method proposed there was based on low-level control parameters. In this work, we combine goal models with feedback controllers to make dynamic tradeoff among conflicting softgoals (i.e., the goals with no binary satisfaction criteria). Reflecting the earned business value of stakeholders, our controller adjusts the preference ranks of softgoals on the basis of runtime feedback. The *preference rank* of a softgoal S_1 is defined by a priority number R_1, indicating its importance relative to another softgoal S_2 of rank R_2: S_1 is preferred to S_2, if and only if $R_1 > R_2$. Intuitively, when it is impossible to meet all the expectations on the individual quality requirements, the softgoals with lower preference ranks will be conceded until the remaining softgoals are achievable.

Having the traceability from goal models to design alternatives (Yu et al., 2005), *intentional variability* expressed by OR-refinement of goals is used in our approach to support dynamic quality tradeoff among the alternatives. We integrate a preference-based goal reasoner to dynamically configure the goal model according to the preference ranks of softgoals adjusted by the feedback controller. Then, based on the goal configurations produced by the goal reasoner and the traceability links to components, an architecture configurator reconfigures the runtime architecture in response to the changing environment.

This paper is an extended version of our previous work (Peng et al., 2010). Besides more detailed description and explanation of the method, we extend the work with an implementation framework and an experimental study with quantitative evaluation of both the effectiveness of the method and its scalability to goal model size. In addition, we discuss the sensitivity and other issues in feedback control and the difficulties in value-based self-tuning, and compare our method with related work.

The remainder of this paper is organized as follows: Section 2 introduces the background on goal-oriented reasoning and control theory. Section 3 presents our self-tuning method that integrates goal reasoning and feedback control algorithms. Section 4 evaluates the method by an experimental study on an online course registration system. Section 5 presents discussion about issues like sensitivity analysis, local tuning and threats to validity. Section 6 discusses and compares this work with related work before the conclusion in Section 7.

2. Preliminaries

Our requirements-driven self-tuning method is founded on goal-oriented reasoning and feedback control theory. Here, we explain briefly the basics before explaining our adaptations later.

2.1 Goal-Oriented Modelling and Reasoning

Stakeholders are usually modelled by agents or actors in requirements engineering. Their desired states prescribe goals that can be further refined into requirements on the system-to-be or expectations on the environment, depending on whether or not the agents are inside the boundary of system analysis. Once the relationships between agents and requirements are elicited, both the KAOS (Lamsweerde and Letier, 2002) and the Tropos (Castro et al., 2002) methodologies converge to a goal-oriented model, representing AND/OR refinements of goals into requirements and expectations. Some requirements have clear-cut satisfaction criteria (modelled as hard goals and tasks) whilst others do not (modelled as softgoals). Most quality requirements are in the second nature, which has no optimal but only "good enough" (or *satisficing*[1])

1 The term "satisficing" was coined by Herbert Simon (Simon, 1996)

solutions. Such uncertainty of quality requirements makes softgoals suitable for expressing the criteria for comparing alternative design choices (Mylopoulos et al., 2001).

Recently, several formal algorithms for evaluating quality requirements after goal-oriented requirements have been proposed. These algorithms can also be used during the elicitation process to provide the rationale for the milestones in design. For example, a bottom-up label propagation algorithm (Giorgini et al., 2003) infers the quantified satisficing level of higher level goals by summing up both positive and negative evidence from the lower-level goal satisfaction labels assigned by designers. Researchers found that degrees of satisfaction of quality requirements have to be grounded on quantifiable metrics that are typically difficult to obtain and aggregate (Letier and Lamsweerde, 2004). We will come back to this point in Section 3.

Another perhaps more general way of goal-based reasoning considers the entire goal model as a set of logic constraints in the conjunctive normal form, which encodes both refinement and contribution rules. The goal-based reasoning discretises a softgoal into propositions[2] of four levels of satisfaction such that they can be reasoned together with the hard goals in proposition logic (Sebastiani et al., 2004). Before further explaining the encodings, we need to emphasise that the number of propositions to be used in discretisation is fully customisable. It depends on the number of states one would *differentiate* for a quality requirement. In practice, our domain experts often noted that a discretisation of more than 7 ± 2 is against the rule of thumb (Simon, 1996) and should be avoided, or be replaced by introducing an additional level of categorisation. The encoding used in (Sebastiani et al., 2004) turns the whole elicited goal model into a set of logic constraints into the conjunctive normal form, which can be expressed by the following formula:

$$\begin{aligned} \Phi &= \Phi\_facts \wedge \Phi\_rules \\ \Phi\_rules &= \Phi\_labels \wedge \Phi\_refinements \wedge \Phi\_contributions \end{aligned} \quad (1)$$

where the term Φ_facts encodes the given satisfaction levels of certain goals known before the analysis; the term Φ_labels encodes the axioms that the fully satisfied (denied) goals always imply their partial satisfaction (denial); the term $\Phi\_refinements$ encodes the equivalence between the satisfaction (denial) of the AND-decomposed goal and the conjunction of the satisfaction (or disjunction of the denial) of the subgoals, similarly it also encodes the equivalence between the satisfaction (denial) of the OR-decomposed goal and the disjunction of the satisfaction (or conjunction of the denial) of the subgoals; finally the term $\Phi\_contributes$ encodes the implication rules between the satisfaction levels of hard goals to the different satisfaction levels of softgoals, depending on the types of contribution (Make, Break, Help, Hurt). The difference between Make and Help is that a satisfied hard goal cannot make the softgoal fully satisfied if the type of contribution is only Help; similarly, a satisfied hard goal cannot make the target softgoal fully denied by the Hurt contribution types.

Although this type of goal reasoning captures the satisfaction/denial levels of goals and requirements, for it to be connected to feedback control from low-level parameters, we will extend it with the notion of preferences and priorities in Section 3.

2.2 Feedback Control Theory and PID Controller

In control theory, a controller follows either a closed-loop control model or an open-loop control model. These two kinds of models differ in whether or not the output of a process influences the control process itself. A closed-loop control is also known as a *feedback control*. Although usually more complex and less stable, feedback control has several advantages over open-loop control, such as guaranteed performance, distinctly reduced process error, improved control precision, disturbance rejection, and reduced sensitivity to parameter variations (Franklin et al., 2006).

The basic structure of a feedback control loop is presented in Fig. 1. Technical experts can specify a *set point,* i.e., the desired characteristics of the system or the monitored *process*. An *output*, or the controlled behaviour of a process, provides *feedback* to the *controller.* The controller tries to minimise the *error,* or delta, between the set point and output, to control the process such that the output is close to the set point.

[2] FullySatisfied (FS), PartiallySatisfied (PS), PartiallyDenied (PD), FullyDenied(FD)

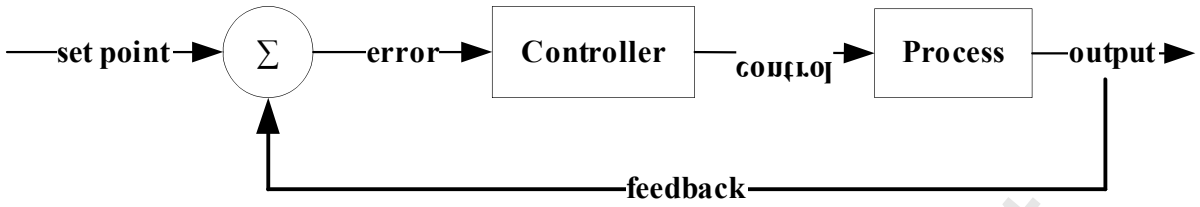


Figure 1. Basic Structure of Feedback Control Loop

Central to the design of a feedback control loop is the choice of the *controller*. We adopt the PID (proportional-integral-derivative) controller (Franklin et al., 2006) in consideration of its precision, stability, and responsiveness. By *precision* the average error is low; by *stability* the standard deviation of errors is low; and by *responsiveness* the response to an error is quick. Equation (2) is a formulation of the basic PID control:

251658240

$$u(t) = Kp * e(t) + Ki * \int_0^t e(\tau)d\tau + Kd * \frac{de(t)}{dt} \quad (2)$$

where *u(t)* is the control variable of the controller at time *t*, and *e(t)* = *set point* - *output* is the error signal at time *t*. An error signal is the triggering sensor of the control loop whilst the control variable here is the actuating executor of the control loop. *Kp*, *Ki*, and *Kd* are the parameters of proportional control, integral control and derivative control respectively, which deal with the current behaviour (reaction to the current error), the past behaviour (reaction based on the sum of the recent errors) and the future behaviour (reaction based on the rate at which the error has been changing) of the process (Franklin et al., 2006). All these control parameters are constants specified at design time (e.g., Kp = 0.5, Ki = 0.3, and Kd = 0.2). Their values reflect the tradeoff among the three performance factors of PID controller, i.e., precision, stability, and responsiveness. In practice, these parameters can be chosen usually by trial and error simulations and other methods (Shaw, 2003).

In absence of the knowledge of the underlying process, PID controllers usually are the best among other feedback loop controllers (Bennett, 1993). This fits our problem due to the uncertainty of a changing environment and a series of conflicting quality requirements. To achieve a desired level of satisfaction for quality requirements, PID controller takes into account the current behaviour through proportional control, the past behaviour through integral control, and the future behaviour through derivative control. In other words, it characterises the temporal behaviour of a process. PID controller synthesizes the benefits from P, PI and PD controllers and overcomes the drawbacks such as P controller's low stability, PI controller's high overshoot, and PD controller's slow response (Franklin et al., 2006). In the domain of software tuning, PID controllers have already been applied successfully in scheduling in real-time operating systems (Steere et al., 1999), automatic stress and load testing tools (Bayan and Cangussu, 2008) and performance optimisation (Chen et al., 2009).

Based on the numerical calculation theory, (2) can be transformed into Equation (3) for simplicity:

$$u(t) = Kp * e(t) + Ki * \sum e(t) + Kd * (e(t) - e(t-1)) \quad (3)$$

However, (3) is related to *all* the past states of the process (sum of the past errors).

To eliminate the accumulation of errors, (3) can be further transformed into Equation (4), the incremental formulation, which is only related to the past three error states: *e(t)*, *e(t-1)*, and *e(t-2)*.

$$u(t) = u(t-1) + Kp * (e(t) - e(t-1)) + Ki * e(t) + Kd * (e(t) - 2 * e(t-1) + e(t-2)) \quad (4)$$

3. Our Method

3.1 Method Overview

To implement a requirements-driven self-tuning system, our method needs to continuously seek opportunities to improve the overall quality satisfaction. Fig. 2 presents an overview. A running system together with the goal reasoner and the architecture configurator forms the process under control. Due to

conflicts among some softgoals, it is often impossible to optimise all the softgoals individually. Therefore a PID controller is used to dynamically adjust the tradeoff decisions, i.e., the preference ranks of related softgoals, to maximise the overall satisfaction based on runtime feedback. In order to provide a runtime feedback reflecting the overall satisfaction, we choose a measurable value indicator from the business perspective and monitor it at runtime.

Intuitively, the controller is designed to prevent a softgoal from getting worse by increasing its preference rank such that different choice in the plans may be found. In other words, every softgoal has an approximately proportional relationship between its preference rank and satisfaction. For example, if the response time is getting too long, the rank of the softgoal "[minimal] response time" should be increased. Given earned business value and quality measurements as feedbacks, our PID controller is designed to adjust the process by tuning the preference ranks of related softgoals to guide the goal reasoning and the following architecture reconfiguration.

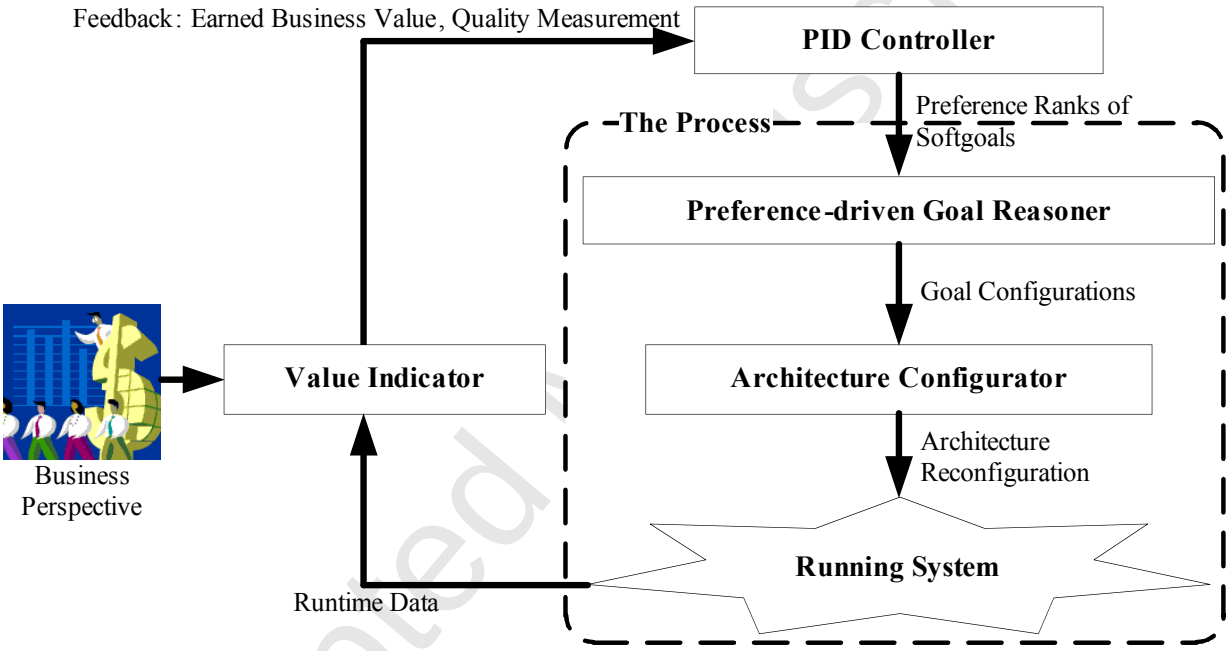


Figure 2. An Overview of Our Method

Following the dynamically tuned preference ranks, our goal reasoner first generates a set of configurations that optimise the achievement of high-ranked softgoals. Each configuration is a selection of the OR-decomposition goals. One "best" configuration can then be chosen by either the "minimum distance" principle (Wang and Mylopoulos, 2009) to make the adaptation smoother or the "maximum distance" principle to make the adaptation more aggressive. Actually, the choice between these two principles is specific to the requirements on adaptation mechanism itself. In this paper, we follow the "minimum distance" principle since stability is preferred. The distance between the next goal configuration and the current one is calculated based on the similarity of configurations, i.e., minimal or no change is required if the two configurations deliver the same level of overall satisfaction. Next the architecture configurator executes the adaptation by reconfiguring the runtime architecture according to the selected goals and the mappings between goals and architectural components. The components corresponding to newly selected goals are bound and integrated, while the components corresponding to those eliminated goals are removed. We assume that such architecture reconfigurations are supported by some kinds of component-based implementation techniques like the service-oriented architecture and a reflective component model such as Fractal (Bruneton et al., 2006).

3.2 A Running Example and Its Requirements Model

The running example to illustrate our method is an online course registration system. It provides a Web-based public service for examinees to register for periodically held public examinations in China[3]. Using the

[3] A running version of the system can be found at: http://www.njzk.net.

system, every examinee can register for a study programme to select courses before an examination deadline. The system consists of about 100K lines of code, including Java classes, HTML pages and database schemas. In our experiments, we use a reengineered and simplified version of the system for evaluation. The reengineering is aimed at turning the system into a service-oriented and high-flexibility implementation that supports component-based architecture reconfigurations.

Some functional and quality requirements of the system are shown in the goal model (Fig. 3). In a typical course registration scenario, an examinee first selects the courses to study ("Course Be Selected"), and then pays registration fees for the selected courses ("Course Be Paid"). Each course selection will be checked by examination rules, and only students who have their course registration fees successfully paid will be accepted by the system. This kind of timing constraints is a prerequisite for the execution of relevant tasks, and can be modelled by precedence links (Liaskos et al., 2010). A deviation of such timing constraints can be regarded as a potential failure, which can be handled by self-repairing mechanisms. As we focus only on self-tuning in this paper, we assume that timing constraints can be ensured by the implementation and therefore omit precedence links in the goal model.

There exist multiple alternatives to achieve these goals. For course selection, an examinee can submit a course registration form either by choosing from the course listed in a complex form ("List and Choose") or by simply entering the course code in a text field ("Input Course ID"). For course payment, two third-party payment services "Pay by AbbPay" and "Pay by UnlPay" are currently available, and either of them supports online payment by credit or debit cards. Note that this OR-decomposition is actually an XOR-decomposition, which means that only one of the sub-goals/tasks can be chosen at each time. The achievement of "Pay by AbbPay" and "Pay by UnlPay" also depends on the social commitments of the service providers (i.e., the third-party payment institutions). Such social relationships can be modelled by the social constructs in i* (Yu, 1997) such as actor dependency and commitment (Chopra et al., 2010). In this paper, we focus on the self-tuning of software systems within the scope of individual actors by switching among alternative solutions. Therefore, for our approach different service providers can be modelled as alternative solutions for specific goals. For example, if a new payment service is introduced, we can add a new task for it and corresponding contribution links to the goal model. Hence, we only use the basic concepts of goal models such as hard goals, softgoals, tasks, contribution links, and AND/OR decompositions. We do not yet consider the social constructs such as actors, dependencies, commitments and resources. Moreover, we use AND/OR decompositions instead of the means-end relationship in i* diagrams (Franch et al., 2011) to link tasks and goals. AND/OR decompositions are simplified propositional logic construct for means-end links in i* diagrams. When description logic and ontology are used, such construct need and can be extended with little change to our algorithms.

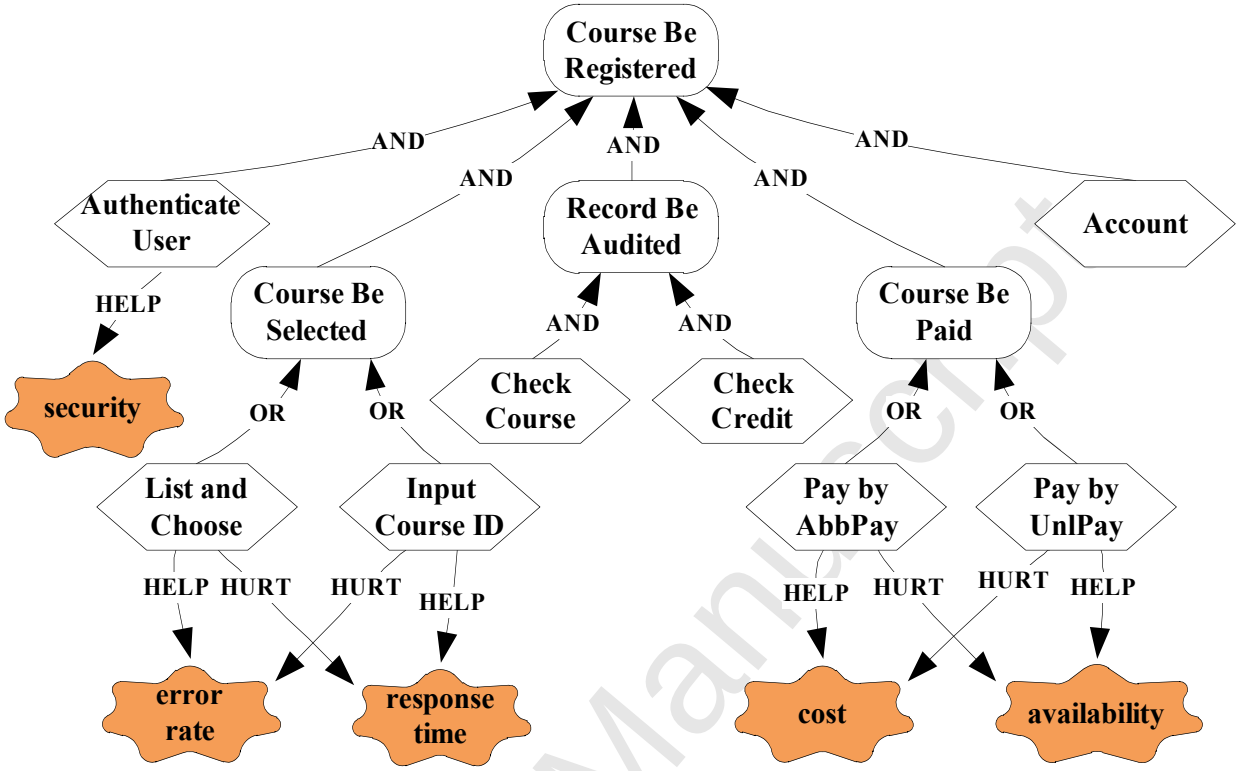


Figure 3. Requirements of the Online Course Registration System

The system runs in a *Software as a Service* (SaaS) fashion: the system owner runs the system under a contract with the education administration and gains profits from every successful course registration. According to this business value proposition, four quality requirements have been elicited, shown as softgoals in Fig. 3. These softgoals contribute to earned business value from different perspectives: "[minimal] response time" is demanded for higher system throughput under a given bandwidth; "[minimal] error rate" is aimed at higher rate of valid course selection; "[minimal] cost" is aimed at lower third-party cost; and "[maximal] availability" is targeted at higher success rate of payment transactions. The alternatives identified for "Course Be Selected" or "Course Be Paid" can exert different influences (e.g., Help, Hurt) on these softgoals. For example, "List and Choose" provides rich client-side aid for users to make correct choices complying with the examination rules, thus helps with the "[minimal] error rate" softgoal, but hurts the "[minimal] response time" softgoal due to its large page size. On the contrary, the alternative "Input Course ID" provides simple mechanisms using a simple textbox for users to input course IDs which minimises the page size, thus helps with the "[minimal] response time" softgoal, but hurts the "[minimal] error rate" softgoal. As third-party service providers, both AbbPay and UnlPay charge a fixed fee for each successful payment transaction. Because AbbPay charges a lower fee, it helps with the "[minimal] cost" softgoal; however the "[maximal] availability" of UnlPay is superior although it is more expensive.

Here the system has an explicit indicator for its earned business value, i.e., the course registration profit in unit time, which amounts to the gain of fees from successful course registrations subtracting the cost paid to third-party payment providers.

Considering environmental changes to the load and the quality of third-party services, dynamic tradeoff decisions must be made to maximise earned business value. For example, when the load is heavy, this goal model may be configured to ensure "[minimal] response time" over "[minimal] error rate"; when the load is light, adverse tradeoff decision can be better for a better earned business value. Since there is an approximately proportional relationship between the preference rank of a softgoal and its satisfaction, we use a PID controller to tune the preference ranks dynamically according to the quality measurements, guided by earned business value.

3.3 Preference-based Goal Reasoning

One contribution of this work is the adaptation of existing qualitative goal reasoning frameworks to business value propositions that are quantitative. The preference ranks assigned to the softgoals form a

partial ordering of the quality requirements. The Preference Based Goal Reasoning algorithm, reduced to a Boolean Satisfiability problem, has been implemented in the new version of OpenOME[4]. The procedure takes as input the given goal model, including a set of hard goals and softgoals, the refinement and contribution relationships, as well as the quantitative labels of expected satisfaction. In addition to these input, dynamic quality tradeoff decisions are expressed by preference ranks of the softgoals because the algorithm does not differentiate two softgoals of the same rank. The preference ranks are changeable input tuned by the feedback controller. The algorithm first encodes the goal model elements into CNF (Conjunctive Normal Form) proposition formulas to feed a SAT solver, and then uses an iterative step to invoke the solver for finite times, depending on the number of ranks needed to express the partial ordering. The procedure tries initially to find a configuration that can accommodate all softgoals to reach their expectations. If not possible, the lowest ranked softgoals will be removed from the encoding so as to accommodate the remaining softgoals, and so on, until either a viable configuration is found or all softgoals have been removed. The algorithm terminates with a valid configuration because we assume that the root hard goal is satisfied by design. Note that the goal reasoning can always find the Pareto optimal solutions since it is reduced to a Boolean Satisfiability problem.

Our encoding extends the original goal reasoning work (Giorgini et al., 2003) by discretising both satisficing and denial of softgoals of 3 labels (fully, partially, unknown) into any $N>3$ labels, depending on the level of details one would tune. In practice, we find $N = 5$ a reasonable value, which means that the interval in [0, 1] is 0.1. As reported earlier (Letier and Lamsweerde, 2004), since goals are expressed as desired states of subject domains, different domains may require different degree of discrimination. In this experimental work, we aim to see whether a relatively small discrimination can still be helpful when the reasoning is combined with the PID controllers.

Algorithm 1. Preference Based Goal Reasoning
input: a goal model with expectations, plus the updated preference ranks of some softgoals
- elements **g, s**: set of hard goals/softgoals
- decomposition relations **d**: $\mathbf{g} \times \mathbf{g} \times \{$ AND, OR $\}$
- contribution relations **c**: : $\mathbf{g} \times \mathbf{s} \times \{ +, - \} \times (0, 1]$
- expectations **e**: $\mathbf{g} \cup \mathbf{s} \times [0, 1]$
- labels **l**: $\mathbf{s} \times [0, 1]$
- preference ranks **r**: $\mathbf{s} \times Z$

output: a set of "good enough" configurations
- If any, output configurations **o**: $2^{\mathbf{g} \cup \mathbf{s}}$

procedure
begin
1. Encode ***g, s, d, c, e, l*** into a CNF proposition formula Φ w.r.t. (Dalpiaz et al., 2009a)
2. *solved* = false; *rank* = 0
3. **while not** *solved* **and** *rank* < *MAX(**r**)* **do**
4. *solved* = solve Φ using SAT solver
5. **if not** *solved* **then**
6. Φ = remove rules in Φ concerning $q \in \mathbf{s}$ where *r(q)* < *rank*
7. **else**
8. ***o*** = decode propositions from Φ concerning satisfied elements in $\mathbf{g} \cup \mathbf{s}$
9. **end if**
10. **end do**
11. **if** *solved* **then return** ***o***
12. **else return** ***null*** **end if**

end

3.4 Modified-PID-based Preference Tuning

One challenge for our feedback controller is to balance the preference ranks of related softgoals while making the adaptation results smoother with less turbulence. To address the challenge, our Modified-PID-based Preference Tuning algorithm takes as input earned business value and quality measurements, and returns the tuned preference ranks (given between 1 and 10, and initially set to 5) of softgoals in response to the changing environment. Then, the goal model reasoning module takes the tuned ranks to select a suitable

4 http://www.cs.toronto.edu/km/openome/

alternative variant for every variation point at OR decompositions. And this control loop executes at regular intervals, for example in the range of the time for a few transactions to finish execution such that the feedback is useful. Too small intervals could defeat the purpose of feedback loop if none transaction can be finished within one interval.

Algorithm 2 Modified-PID-based Preference Tuning

input: **value**: earned business value in the last time unit, plus **qm**: monitored quality measurements in the last time unit

output: **ranks**: tuned preference ranks

procedure

begin

1. *// calculate the delta of earned business value from its set-point*
2. *delta* = (***value*** – *set-point*) / *set-point*
3. update *set-point* as average of all the past earned business values
4.
5. // *decide whether to perform the preference tuning or not*
6. **if** *delta* < α **then** *tuning* = true
7. **else** *tuning* = false **end if**
8.
9. *i* = 0; *length* = the length of ***qm***
10. **while** *i* < *length* **do**
11. *// calculate the error signals and control variables of quality measurements*
12. *e(k)[i]* = (*set-points[i]* - ***qm[i]***) / *set-points[i]* * *isPositive[i]*
13. *c[i]* += Kp * (*e(k)[i]* – *e(k-1)[i]*) + Ki * *e(k)[i]* + Kd * (*e(k)[i]* – 2 * *e(k-1)[i]* + *e(k-2)[i]*)
14.
15. *// update the past error signals and set-points of quality measurements*
16. *e(k-2)[i]* = *e(k-1)[i]*; *e(k-1)[i]* = *e(k)[i]*
17. update *set-points[i]* as average of all the past values of quality measurements
18. **end do**
19.
20. *// tune the preference ranks*
21. **if** *tuning* **then**
22. **while** *i* < *length* **do**
23. ***ranks[i]*** = 5 + *c[i]* * ***ranks[i]***
24. **if** ***ranks[i]*** > 10 **then** ***ranks[i]*** = 10
25. **else if** ***ranks[i]*** < 1 **then** ***ranks[i]*** = 1 **end if**
26. **end do**
27. **return** ***ranks***
28. **end if**
29. **return** ***null***

end

We made three adaptations to the PID control model introduced in Section 2 to fit our specific control problem. First, set points are initially unspecified, and then moved gradually from old values to newly specified ones that reflecting the actual status of runtime environment. Thus, we evaluate set points as the average of all the past values of corresponding indicators (Lines 3 and 17).

Second, the delta of earned business value is calculated as the percentage of increment/decrement (Line 2), which determines whether to tune the preference ranks or not (Line 5-7). The error signal of a softgoal represents the percentage of its deviation from set point (Line 12), and the control variable represents the percentage of increment/decrement of the preference rank based on the past error signals (Lines 13) using (4). It is worth to mention that some quality attributes are positive attributes (to be maximised, e.g., availability) while some others are negative attributes (to be minimised, e.g., response time). To unify them, we multiply negative attributes by -1 and positive attributes by 1 (Line 12).

Third, a dead band[5] of the delta of earned business value, or a tolerable range, is used to avoid oscillations in frequent architecture reconfigurations. Since earned business value should be as large as possible, the dead band shows an entire dead side ($[\alpha, +\infty)$, the value of α will be discussed in Section 5) rather than a band between two values. More specifically, if the delta of earned business value is within the predefined dead band, then the preference ranks will not be tuned (Line 7). Otherwise, the preference ranks should be tuned based on the control variables (Line 20-28). For example, suppose the value of α is -0.05 and the current set point is 200: when the monitored earned business value is 180, the delta is (180 - 200) / 200 = -0.10, which is smaller than α (i.e., not within the dead band), then the preference rank should be tuned; when the monitored earned business value is 210, the delta is (210 - 200) / 200 = 0.05, which is larger than α (i.e., within the dead band), then the preference ranks will not be tuned.

5 In control theory, a dead band refers to a range of the signal where no switching action is needed.

4. Implementation and Experimental Study

We have implemented the proposed method in an extensible framework and validated the method using the online course registration system introduced in Section 3.

4.1 Implementation Framework

In order to provide a generic infrastructure for self-tuning applications, we implement the proposed method in an extensible framework as shown in Fig. 4. The framework provides an administrator console to configure and monitor the self-tuning process. A tuning controller together with the PID controller and goal reasoner are implemented in the framework. The PID controller accepts control parameters configured with the administrator console, and the users can also use default parameters. It periodically invokes the external feedback interface to obtain application-specific earned business value and quality measurements from the running system. The goal reasoner accepts goal models modelled by OpenOME as the basis of goal reasoning. And the tuning controller invokes the external architecture reconfiguration interface to map the goal configurations to architecture-level reconfigurations.

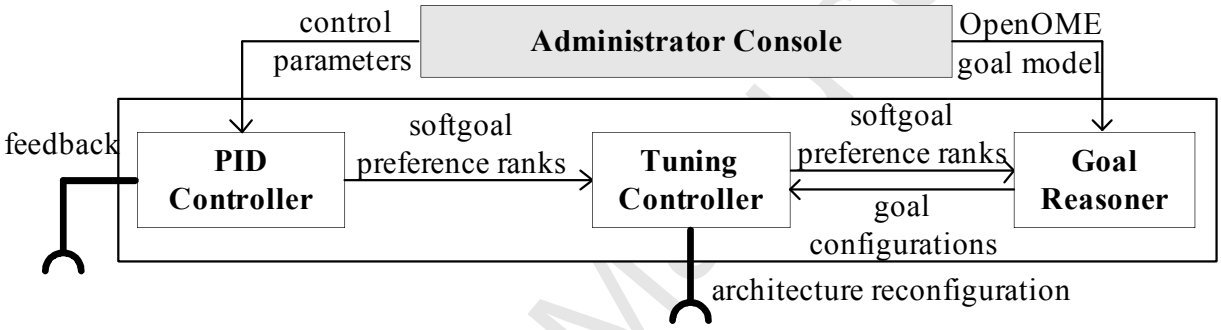


Figure 4. Implementation Framework

To separate the communication between framework and running system and make them distributed and stand-alone, we use RMI-IIOP (Oracle SDN) to program the two external interfaces as RMI interfaces.

When applying the framework, developers should provide their own implementation to the two external RMI interfaces. Implementation to the feedback interface depends on the ways of business implementation and on the characteristics of quality attributes. Typical feedback implementation includes computing earned value of business transactions and quality attributes by analysing the business database and log files.

Implementation to the architecture reconfiguration interface highly depends on the reconfigurable architecture style (Abowd et al., 1995) used in the target system. For example, service-based systems usually implement architecture reconfiguration by service selection and replacement, and component-based systems using reflective component model like Fractal (Bruneton et al., 2006) typically use control interfaces provided by the component infrastructure to implement component adding/removal, binding/unbinding, etc. Some lightweight techniques like parameterization can also be used. For example, the experimental implementation of our online course registration system is based on parameterization and language-level reflection: the system instantiates service components using Java reflection at runtime, and the architecture reconfiguration implementation dynamically modifies the reflection configuration file according to the goal configurations and goal-component mappings.

4.2 Experimental Settings

Our experiment simulates a continuous running of the system based on scenarios reflecting the real-life data logged from the changing environments like concurrent load and operational error rates during the months of active usage of the online course registration system in 2009. The benchmark is encoded into a series of script files, which are used as a common environmental setting to compare our method with two alternative implementations that makes static design-time decisions and hard-coded adaptation respectively. The stress testing tool JMeter 2.3.4 is used to simulate concurrent accesses to the system. JMeter test plan bootstraps the whole execution process according to several data files like user password list and course ID selection list. To simulate the changing environment, we also devised the following fluctuant settings in the experiment scenario:

- ***Execution time of course selection***. As a database-based function, execution time of course selection is highly relevant to the size of related business tables. A separate thread is designed to periodically insert or delete a batch of records from related tables to make a changing server-side execution time.

- ***Rate of valid course selection.*** Stochastically distributed non-existent course IDs are included in the course ID file which is read by JMeter test plan.
- ***Failure rate of payment service***. Execution profiles of both payment services include stochastically distributed failures with different density to reflect their failure rate.

Earned business value in terms of system profit in the experiment is collected by analysing the online course registration database. Besides, log file is also analysed at runtime to capture runtime environment parameters like response time and failure rate for effectiveness analysis.

In order to evaluate our method, the experiment is executed with three different kinds of solutions using the same environmental setting.

- Static Configuration: This solution is the naive solution corresponding to static design decision made *a priori*, which uses a fixed variation configuration for all the variation points.
- Hard-coded Adaptation: In an uncertain running environment experienced by the online course registration system, developers often need to use hard-coded and ad-hoc adaptation mechanism to guess a better performance. Therefore, this solution implements a simple runtime adaptation for the two variation points by introducing ECA (event-condition-action) rules. For "Course Be Selected", the adaptation rule switches between "Input Course ID" and "List and Choose" according to whether the response time is larger than a given switch threshold. For "Course Be Paid", a similar rule is specified to switch between the two alternatives according to the failure rate of payment service.
- Requirements-driven Self-tuning: This solution employs our requirements-driven self-tuning algorithm. To further compare the feedback controller with single global value feedback and several local value feedbacks, we also execute this solution with local value feedbacks. By global we mean that one earned business value is defined and measured for all transactions, by local we mean that multiple values are defined and measured for different kinds of transactions.

Once deployed, the entire feedback control loop in our self-tuning process is timed and unsupervised. In the experiment, we use the *ServletContextListener* interface and the *TimerTask* class in standard Java library to implement the timing property.

4.3 Main Experimental Results and Comparisons

In each experiment with the three different solutions, we collect and analyse the average cumulative profit per 30 seconds in a continuous running of 40 minutes. Besides profit, we also analyse the valid throughput which denotes the amount of course registrations complying with examination rules but not necessarily successfully paid, since it is also an important factor for the overall profit. The valid throughput is calculated by analysing the online course registration database. All the experiments are executed with the same environmental settings, including hardware, network and execution scripts.

TABLE I. EXPERIMENTAL RESULTS OF VALID THROUGHPUT/PROFIT

Solution		Valid Throughput	Profit
Static Configuration	Input Course ID & Pay by AbbPay	166.200	190.000
	Input Course ID & Pay by UnlPay	165.233	198.150
	List and Choose & Pay by AbbPay	165.246	187.664
	List and Choose & Pay by UnlPay	164.905	197.667
	Maximum	166.200	198.150
Hard-coded Adaptation		166.384	206.463
Requirements-driven Self-tuning		170.645	221.214

The first 5 rows in Table I show the results of the experiments with static variation configurations by design-time decisions, as well as the maximum. From the requirements model in Fig. 3, it can be seen that there are four different combinations for the two variation points. We execute the experiment for each configuration and obtain the data as presented in Table I. These results demonstrate that the four static configurations have similar but a little different valid throughput and profit. It should be noted that the maximum row derives from two different configurations, which actually indicates the inability of a static method to satisfy the optimal, i.e., with both the highest valid throughput and profit.

From the 6th row in Table I, we can see that even simple hard-coded adaptation can slightly outperform the best static configuration. The result of our self-tuning method is given in the last line. It can be seen that our method has a 2.56% improvement on valid throughput and a 7.14% improvement on profit compared

with the simple hard-coded adaptation. The improvements are 2.67% and 11.64% respectively if compared with the best static configuration.

4.4 Evaluation of the Detailed Results

From Table I, we can see that our method has notable improvement over static configurations and simple hard-coded adaptation. In order to further validate and evaluate the effectiveness of our method, we then analyse the self-tuning process of our method with respect to the simulated changing runtime environment. Since adaptive method is better than those of static decisions, we evaluate by comparing our self-tuning method with the hard-coded adaptations.

Fig. 5 and 6 record the adaptation process with the changes of profit for the two variation points respectively. The *X* axis denotes discrete time intervals of 30 seconds, while the *Y* axis denotes profit in each time interval. Switch points are identified as points on the curves.

Fig. 5 shows the adaptation process for "Course Be Selected", distinguishing the triangles denoting switches from "List and Choose" to "Input Course ID" and the squares denoting reverse switches. Fig. 6 shows the adaptation process for "Course Be Paid" using a similar legend.

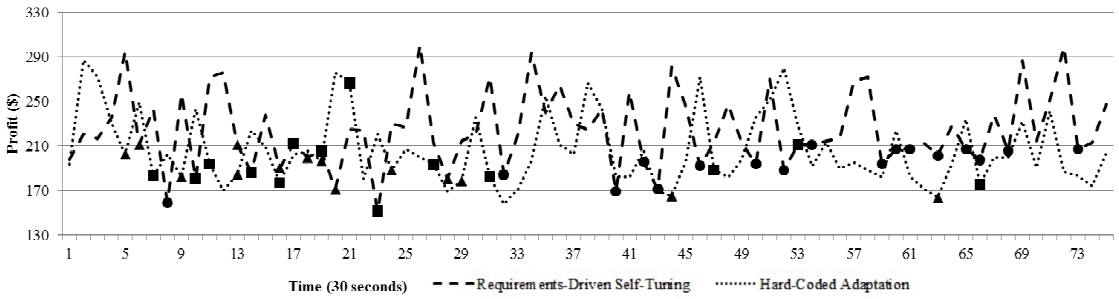


Figure 5. Profit VS. Time ("Course Be Selected") (Requirements-Driven Self-Tuning vs. Hard-Coded Adaptation)

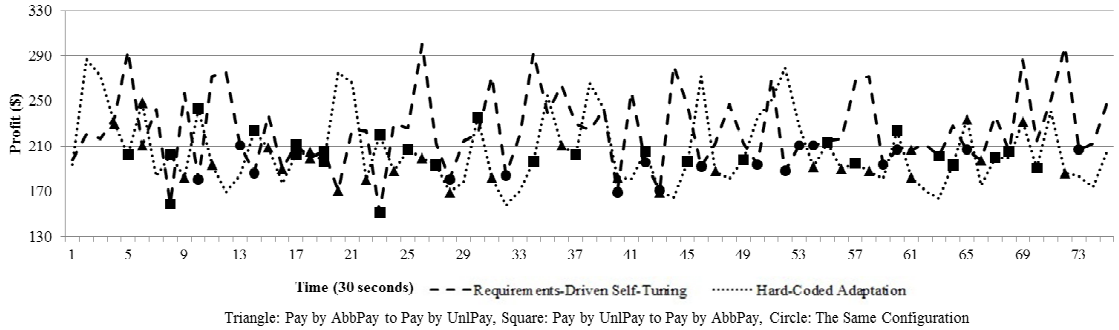


Figure 6. Profit VS. Time ("Course Be Paid") (Requirements-Driven Self-Tuning vs. Hard-Coded Adaptation)

As for our method, there are mainly four situations:

(1) "Course Be Selected" is reconfigured while "Course Be Paid" is not. E.g., at time 10, the profit is decreased probably because the error rate is too high, which drives the reconfiguration from "Input Course ID" to "List and Choose". And this is consistent with the goal model, in which "Input Course ID" has a Hurt contribution to "[minimal] error rate" and "List and Choose" has a Help contribution to "[minimal] error rate".

(2) "Course Be Paid" is reconfigured whilst "Course Be Selected" is not. E.g., at time 8, profit is decreased probably because the cost is too high, which drives the reconfiguration from "Pay by UnlPay" to "Pay by AbbPay", which is also consistent with goals.

(3) Both variation points are reconfigured (e.g., at time 16), which means none of the current configuration of the two variation points is suitable for current environment.

(4) None of the two variation points are reconfigured (e.g., at time 32), which means current configuration is already the best for the current environment although profit is decreased. However, this probably results from the fact that there is only one global earned business value to guide the reconfiguration of

all the variation points, and the relationship between earned business value and variation points is fuzzy and indirect. And extra tentative experiments were performed on our example concerning this problem, which will be discussed in Section 5. Furthermore, sometimes profit still decreased even after reconfiguration (e.g., at time 13, 17 and 19, totally 3 times), which probably still rises from the indirect and non-specific earned business value.

Comparing with the hard-coded method, there are three main differences: (1) as a whole, the self-tuning curve is a little higher than the hard-coded curve, which is already visible in Table I; (2) the frequency of the situation in which profit still decreases even after reconfiguration is less in our self-tuning method (totally 3 times) than in hard-coded method (e.g., at time 11, 21, 27, 31 and 47 in Fig. 5, totally 30 times); (3) the frequency of reconfiguration of self-tuning method (totally 24 times) is less than that of hard-coded one (totally 59 times). In other words, self-tuning is more stable than hard-coded method.

4.5 Performance Evaluation

Although the worse-case analysis of the performance of the SAT solver algorithms for arbitrary proposition formula is NP-complete, in reality the performance results also depend on the structure of the knowledge bases which may be exploited by the SAT solvers. To evaluate an empirical evaluation of the time performance of the proposed preference-based reasoning algorithm, we conducted two sets of experiments with randomly generated goal models. The first set includes 8 experiments with a fixed density of softgoals (i.e., constant ratio between the number of softgoals and the number of goals), indicating that our method can be scaled to 28 quality requirements and 175 functional requirements. The second set contains 24 experiments with floating density of sofgoals, demonstrating that our method can also be applied to larger software systems provided that the number of softgoals is not too large. Note that every two softgoals are associated with a variation point (hard goal) with two alternative sub-goals/tasks.

TABLE II. REASONING PERFORMANCE WITH FIXED DENSITY OF SOFTGOALS

#G(#SG)	25(4)	50(8)	75(12)	100(16)	125(20)	150(24)	175(28)	200(32)
Average (s)	0.18	0.34	0.47	0.56	0.85	2.09	8.81	42.75
Maximum (s)	0.27	0.45	0.51	0.66	0.91	2.47	9.43	43.10

Table II reports the results of the first set of experiments, showing the worst cases reasoning performance at different scales. Row 1 in Table II lists the size of the goal model in terms of the number of goals (#G) and the number of the softgoals (#SG). Row 2 and 3 give respectively the average and maximum performance of the reasoning algorithm in seconds, including the time to encode the goal model and known facts into CNF proposition formula, as well as the time by the SAT solver to tackle the problem, plus the time to decode the SAT solutions back into goal model configurations.

The first 5 experiments (column 2-6 in Table II) show it takes less than 1 second to find all the configurations when there are no more than 20 softgoals (10 variation points, 1024 configurations). As the size of goal model increases (column 7 and 8) (14 variation points, 16384 configurations), it takes less than 10 seconds, which is still feasible in our method. As the size (column 9) climbs to 32 softgoals (16 variation points, 65536 configurations), it took more than 43 seconds to produce the best configuration, making it infeasible in our method at this scale. Given that most systems have no more than 30 quality requirements to be trade-off (e.g., ISO 9126 standard lists 26 key quality requirements), our method is probably usable for medium-sized systems. To confirm this hypothesis, we conducted the second batch of experiments.

TABLE III. REASONING PERFORMANCE WITH FLOATING DENSITY OF SOFTGOALS

#SG \ #G	75	100	125	150	175	200
8	0.34 (0.37)	0.36 (0.42)	0.38 (0.44)	0.40 (0.45)	0.41 (0.48)	0.44 (0.50)
12	0.47 (0.51)	0.49 (0.54)	0.51 (0.55)	0.52 (0.57)	0.56 (0.59)	0.60 (0.68)
16	0.51 (0.63)	0.56 (0.66)	0.58 (0.66)	0.60 (0.67)	0.64 (0.70)	0.68 (0.73)
20	0.75 (0.83)	0.81 (0.87)	0.85 (0.91)	0.87 (0.99)	0.97 (1.11)	1.01 (1.18)

Table III reports the results of the second set of experiments, showing the average (maximum) worst cases reasoning performance in seconds. Each row represents the time as the number of goals increases from 75 to 200, with a fixed number of softgoals. The elapsed time is mostly less than 1 second, and increases slowly and gently at different scales. According to these experiments, our reasoning algorithm is

scalable to larger goal model size when the number of softgoals is no larger than 20 (10 variation points, 1024 configurations).

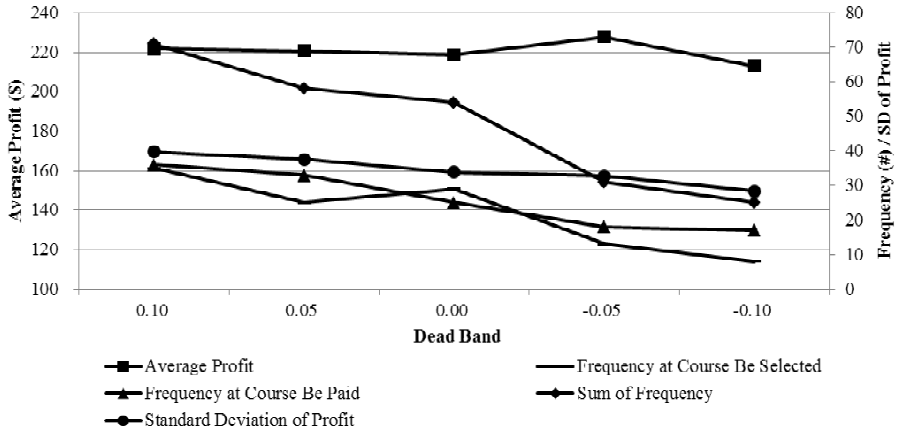
Combining both Tables II and III, we conclude that our method can be applied to medium-sized systems, and can also be applied to large-sized systems provided that the number of softgoals in consideration is not very large. Furthermore, large-sized systems can be decomposed into medium-sized sub-systems that can apply our method. This is referred as local tuning, which will be discussed in Section 5. In additional, since we use RMI-IIOP to implement the framework, it can be deployed separately from the running system. Hence, the affection to some quality attributes by enacting such a self-tuning framework can be ignored.

5. Discussion

5.1 Sensitivity Analysis

As mentioned in Section 3, the dead band, which is determined by the parameter α in our control algorithm, represents the tolerance of value fluctuation. Therefore, the parameter α is highly relevant to the sensitivity of the feedback controller. Intuitively, if it is too small, the controller may response too slow in tuning. Contrarily, if it is too big, the controller will be too sensitive, leading to the problem of oscillation.

To explore how to determine the parameter α, we conduct a series of experiments on the self-tuning method with different α and capture related data as shown in Fig.7, including average profit, standard deviation of profit, and frequency of system adaptation on either of the two variation points. From the frequency data, it can be seen unsurprisingly that the smaller α is, the less frequent self-adaptations are. Furthermore, the standard deviation data shows that smaller α can help to make more smooth system performance. From the most important profit curve, we can see that besides slow controllers, overly sensitive controllers also hurt the overall satisfaction. We believe this may be due to two causes. One is that too frequent adaptations cannot capture the essence of runtime evolution of the system well, and the other is the cost of runtime adaptation itself. Combining these factors, we believe for this experiment the best α should be around -0.05. We also believe the best α, or sensibility of the controller, is system dependent. It is relevant to the business goal and capability and should be determined accordingly by economic experts.



251658240

Figure 7. Average Profit, Frequency of System Adaptation, and Standard Deviation of Profit with Different Dead Band

5.2 Global Tuning versus Local Tuning

Concerning the increasing complexity of software systems, we believe that systems should be decomposed into sub-systems with medium size and for every sub-system a more direct and specific earned business value should be associated; on the other hand, multiple variation points are associated with every sub-systems. Thus, multiple earned business values that are relatively independent can be defined and measured at runtime to guide the feedback control on every sub-system.

For example, the two sub-systems of online course registration system are course registration and course payment. Valid throughput is the earned business value for course registration sub-system and profit is the earned business value for course payment sub-system. To illustrate the motivation, we have modified our method to support multiple earned business values for local tuning, and conducted extra tentative experiments.

Fig. 8 and 9 show how throughput changes within an uncertain environment where the response time and error rate is changing when the modified method is applied. Due to space limitations, the curves of profit, cost, and availability are not included in this paper. The reconfiguration time is marked at all of the three curves distinguishing switching points (reconfigured from "List and Choose" to "Input Course ID") and switching points (reconfigured from "Input Course ID" to "List and Choose"). As shown in these figures, the reconfiguration happened almost always when throughput is at the bottom of the curve. In other words, earned business value is locally minimal. At the same time, one of the two quality requirements "[minimal] response time" and "[minimal] error rate" is also violated, which is consistent with the goal model. For example, at the first switching point, throughput is decreased dramatically because the response time of the "List and Choose" is increased a lot. Thus, the "Course Be Selected" variation point is automatically reconfigured to "Input Course ID" which has a Help contribution to "[minimal] response time" and a Hurt contribution to "[minimal] error rate". Once the reconfiguration happens, throughput increases almost immediately. On the other hand, when the environment is relatively stable, for example, between time 26 and time 41, no reconfiguration has happened.

Compared with Fig. 5 and 6, the curves are smoother, and there are two main differences: (1) earned business value always increases after reconfiguration while earned business value sometimes still decreases even after reconfiguration (e.g., at time 17 in the hard-coded curve of Fig. 5, and at time 11 in the self-tuning curve of Fig. 6); (2) reconfiguration happens at the bottom of the curve while reconfiguration sometimes happens when earned business value is increasing (e.g., at time 13 in the self-tuning curve of Fig. 6, and at time 53 in the self-tuning curve of Fig. 5).

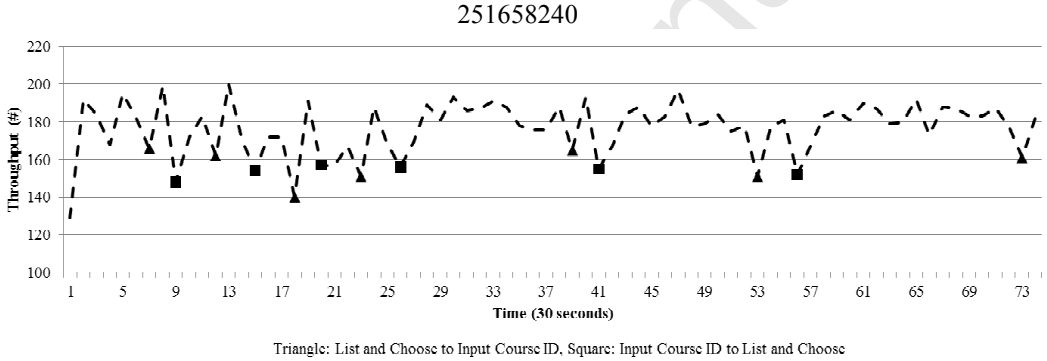


Figure 8. Throughput vs. Time (Local Tuning)

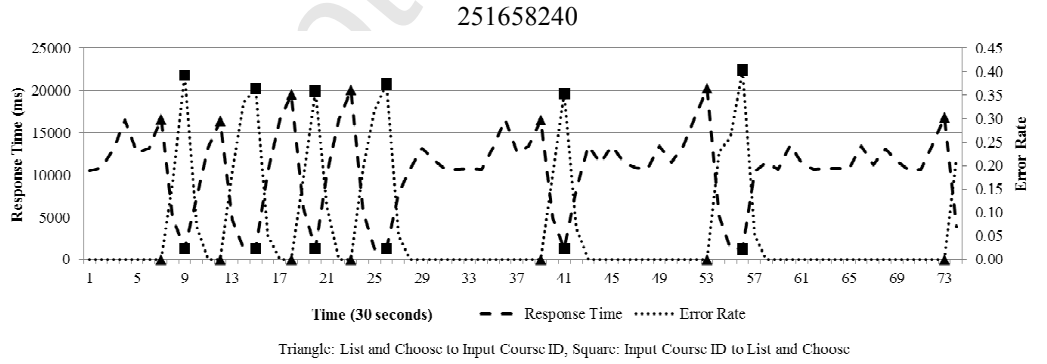


Figure 9. Response Time, Error Rate vs. Time (Local Tuning)

5.3 Threats to Validity

Value-based perspective (Boehm, 2006) is helpful to elicit a satisfaction indicator measuring the essential business value of stakeholders. Generally speaking, value-based feedback, usually by computing earned business value of successful transactions, is a reasonable indicator for self-tuning. However, when applying our value-based self-tuning method, whether a *combined value measurement* can be defined and a *quick value feedback* can be achieved are the two main threats to validity. The former determines whether a value indicator can be used as the feedback at runtime, while the latter determines whether the feedback is quick enough to indicate runtime tuning.

A. Combined value measurement. A business system often provides a series of services and undoubtedly the value feedback should reflect a combined value measurement to all the different kinds of services. However, not all of the services have obvious value formulation on a common value basis, e.g., economic profits. For example, besides course registration, the online course registration system may provide other services like online consulting and course video preview. These services do not produce profits themselves, but they have indirect, sometimes important, influence on the final economic profits. For instance, some users may decide to register courses after they preview the course video or get questions answered by online consulting. Combined value measurement based on a common basis is essentially a business issue that should be solved from two aspects. For those highly relevant services, i.e., satisfaction levels of some services have significant influence on others, their combined value measurement may be formulated from the aspect of business and market analysis. For example, the relative value of course video preview and online consulting to course registration can be measured by answering the question "how much percentage of the users will register a course only after they preview the course video or get answers by online consulting". This can be computed by a market investigation and user behaviour analysis. For those independent or loosely related services, local satisfaction indicators may be elicited for some critical quality tradeoff to enable local tuning if quick global feedback is not possible.

B. Slow value feedback. The value measurement may be too slow as feedback for self-tuning in some cases. A basic assumption behind value-based self-tuning is that variations of all the quality attributes can be transmitted to the value measurement, making it an instant feedback for self-tuning. However, the transmission cycle may be too long to be used as an instant feedback. For example, some quality degradation may hurt the reputation of the system, and then result in losing of customers. This, in turn, will greatly influence the business volume, but the influence is too late to be reflected in runtime value measurement. One possible solution to this problem is to capture and measure some early indication of long-term influences as part of the value measurement. For example, instant online customer complaints can be collected and considered as a negative factor to earned business value. Precise value measurement of this kind of early indication requires corresponding analysis and prediction model, e.g., customer complaints and business losing trend analysis.

In addition, lacking a clear traceability link between goals and architectural elements can also be a problem due to the well-known difficulty of mapping requirements and designs. Just as Wang et al. stated (Wang et al., 2009), if detailed traceability links between low-level elements are not available, higher level traceability links can still be used to relate high level goals to the components of larger-scale systems to exert a higher-level control. More than one layers of feedback controls can be used together to manage the complexity of large-scale systems (Kephart and Chess, 2003). In near future, we intend to perform the experiment on such a system.

Last but not the least, there may be difficulty in applying our method to a legacy system. On one hand, the system has to provide runtime data to facilitate the implementation of feedback interface; on the other hand, the system has to provide pre-defined interfaces or comply with specific architecture (e.g., service-oriented system, component-based system) to facilitate the implementation of architecture reconfiguration interface. Possible solutions to this difficulty are to combine architecture-based self-adaptation method such as Rainbow (Cheng, 2008) or to reengineer the legacy system.

6. Related Work

Requirements-driven self-tuning. Our work falls in the area of requirements-driven self-managing systems (Cheng et al., 2009a, 2009b; Cheng and Altee, 2007; Kephart and Chess, 2003; Kramer and Magee, 2009; Lapouchnian et al., 2005). A desired self-managing systems can reconfigure to meet multiple runtime needs (self-reconfiguring (Dalpiaz et al., 2009b)); can adapt its solution in different contexts to maintain the core requirements (self-healing (Salifu et al., 2007)); can tune its key performance to accommodate quality requirements (self-optimising (Sadjadi et al., 2004)); and can protect it from attacks that may harm or cause damage to valuable assets (self-protecting (Chung, 1993)). These four self-* properties were first proposed by IBM autonomic computing (Kephart and Chess, 2003). Feedback loops of monitoring, analysis, planning and execution activities are seen as indispensable part for such self-managing systems. Although a few requirements-driven self-reconfiguration and self-healing methods have been proposed (Dalpiaz et al., 2009a, 2009b; Salifu et al., 2007), it is our belief that connecting high-level requirements to low-level tuning parameters is a mandatory step for self-tuning. We do not intend to address all non-functional requirements such as including security ones. Although regarded as one kind of non-functional requirements (Glinz, 2007; Haley et al., 2008), security requirements are fully satisficed

through self-protecting instead of trading them offs with other quality requirements through self-tuning. In this work, we aim to address remaining quality requirements such as performance, usability, etc., those may let the system to trade off.

Architecture-based self-adaptation. Architecture-based self-adaptation methods tie high-level architecture elements with low-level quality indicators. Most of these methods support specific architecture styles and fixed quality attributes (Batista et al., 2005; Hinz et al., 2007); and little is general enough to be applied to different architecture styles and support flexible quality attributes (Cheng, 2008). These methods can support much more complex architecture reconfigurations, but do not consider dynamic tradeoff decisions for self-tuning at the requirements level. Hence, it is valuable to integrate our method with architecture-based self-adaptation methods to better support requirement-driven self-tuning with more sophisticated architecture reconfigurations.

Rationales behind quality requirements and goal reasoning. Measuring some quality attributes as "key performance indicators" may appeal to the quick-to-market development of self-tuning systems; however, it is the lack of rationale behind these metrics makes it hard to explain why certain indicators are the "key" whilst others are not. Our work shows that the goals modelled from stakeholders can directly provide such a rationale and allows for various goal-based reasoning methods to be plugged into the self-tuning framework. For the selection among alternatives, quality requirements have been used to perform both qualitative and quantitative reasoning (Giorgini et al., 2002; Sebastiani et al., 2004). Such automated goal reasoning algorithms can suffer from being too general to "good enough" criteria on domain-specific metrics, therefore it was suggested that they are to be performed partially (Letier and Lamsweerde, 2004). We address the limitation of discretising every softgoal uniformly into 3 labels by allowing more flexible encoding for domain-specific softgoals, which allows us to perform preference-based goal reasoning automatically on domain-specific softgoals. On the other hand, human intensive judgements for tradeoff decisions (Horkoff and Yu, 2008) are considered useful and necessary especially for eliciting mission-critical quality requirements such as security (Eliha and Yu, 2009). One of the findings in (Eliha and Yu, 2009) is that 10 degrees is often enough to discriminate different level satisfaction of softgoals. Our experiment also showed that an interval of 0.1 was sufficient for the example driven by relatively small number of quality requirements. Recently, a HTN (Hierarchical Task Networks) preference-based planer is used to efficiently search for alternatives that best satisfy the given preferences (Liaskos et al., 2010), which can in future be integrated into our framework to provide more efficient goal reasoning.

QoS management. In terms of applications, the proposed requirements-driven self-tuning can be used not only for traditional software systems. As shown in our experiments, quality of service (QoS) for Web-based software systems can also benefit from it. Quality-driven self-adaptive methods have been widely used for runtime tuning, e.g., QoS-driven service selection and composition (Alrifai and Risse, 2009; Cardellini et al., 2009; Menasce and Dubey, 2007). However, most of these proposals do not account for the rationale behind the QoS. The most related work to us in this area is a requirements-driven self-reconfiguration mechanism applied to business processes (Lapouchnian et al., 2007), which uses a variant of preference-based goal reasoning to adjust the parameters. The main difference is that we do not perform manual adjustment of the goal models preferences, instead, the PID controller can fully automatically apply increment/decrement of the preference rank to softgoals to make the switching more smooth and agile.

Value-based software engineering. In terms of business values, our method shows that it is possible to integrate value calculations into the goal models (Boehm, 2006). Interestingly, value-based requirements engineering framework has been applied to model the business requirements for large finance organisations (Gordijn et al., 2006). In that work, complementary to us, spreadsheet formulas were used to metricise the goals to evaluate the give-and-take dependencies across organisations. In this work, we found it not always easy to find a precise calculation for every softgoal, in addition, a more precise calculation provided by domain experts can be plugged into our feedback control framework as set points.

7. Conclusions

Runtime quality tradeoff algorithms based on the mapping from goals to architectural adaptations have been proposed to improve the overall satisfaction of requirements. In this paper, we proposed a requirements-driven self-tuning method that involves a feedback controller to tune the expected satisfaction levels of softgoals so as to configure hard goals accordingly. Architectural adaptations were conducted based on the linking between hard goals and architectural elements to dynamically optimise the overall satisfaction.

Our experimental study on a Web-based system has shown that combining PID control theory with preference-based goal reasoning is effective in runtime self-tuning for a real-life software system. For our method to obtain quick feedback, a value-based perspective was applied to select more local satisfaction indicators. In future, we plan to further improve the work by customising different feedback control algorithms, and by applying our method to more open and uncertain systems such as SOA applications.

Acknowledgments

This work is supported by National Natural Science Foundation of China under Grant No. 90818009, National High Technology Development 863 Program of China under Grant No. SS2012AA010102, and the EU FP7 SecureChange project (http://securechange.eu).

References

Abowd. G., Allen. R., Garlan. D., 1995. Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering and Methodology 4 (4), 319-364.

Alrifai, M., Risse, T., 2009. Combining global optimization with local selection for efficient QoS-aware service composition. In: Proceedings of the 18th International Conference on World Wide Web, pp. 881-890.

Batista, T., Joolia, A., Coulson, G., 2005. Managing dynamic reconfiguration in component-based systems. In: 2nd European Workshop on Software Architectures, pp. 1-17.

Bayan, M., Cangussu, J.W., 2008. Automatic feedback, control-based, stress and load testing. In: Proceedings of the 2008 ACM Symposium on Applied Computing, pp. 661-666.

Bennett, S., 1993. A history of control engineering 1930 - 1955, 1st ed. Hitchin, Herts., UK, UK: Peter Peregrinus.

Boehm, B., 2006. Value-based software engineering: overview and agenda. In: Value-based software engineering, pp. 3-14.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B., 2006. The FRACTAL component model and its support in java. Software: Practice and Experience 36 (11-12), 1257–1284.

Cai, K., Cangussu, J.W., DeCarlo, R.A., Mathur, A.P., 2003. An overview of software cybernetics. In: Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice, pp. 77-86.

Cardellini, V., Casalicchio, E., Grassi, V., Presti, F.L., Mirandola, R., 2009. QoS-driven runtime adaptation of service oriented architectures. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 131-140.

Castro, J., Kolp, M., Mylopoulos, J., 2002. Towards requirements-driven information systems engineering: the Tropos project. Information Systems 27 (6), 365-389.

Chen, B., Peng, X., Zhao, W., 2009. Towards runtime optimization of software quality based on feedback control theory. In: Proceedings of the First Asia-Pacific Symposium on Internetware, pp. 1-8.

Cheng, B.H.C., Atlee, J.M., 2007. Research directions in requirements engineering. In: Workshop on the Future of Software Engineering, pp. 285-303.

Cheng, B.H.C., de Lemos, R., Garlan, D., Giese, H., Litoiu, M., Magee, J., Muller, H.A., Taylor, R., 2009a. SEAMS 2009: software engineering for adaptive and self-managing systems. In: 31st International Conference on Software Engineering - Companion Volume, pp. 463-464.

Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J., 2009b. Software engineering for self-adaptive systems: a research roadmap. In: Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar], pp. 1-26.

Cheng, S.W., 2008. Rainbow: cost-effective software architecture-based self-adaptation. PhD Thesis.

Chopra, A.K., Dalpiaz, F., Giorgini, P., Mylopoulos, J., 2010. Modeling and reasoning about service-oriented applications via goals and commitments. In: Proceedings of the 22nd International Conference on Advanced Information Systems Engineering, pp. 113-128.

Chung, L., 1993. Dealing with security requirements during the development of information system. In: Proceedings of Advanced Information Systems Engineering, pp. 234-251.

Dalpiaz, F., Giorgini, P., Mylopoulos, J., 2009a. An architecture for requirements-driven self-reconfiguration. In: Proceedings of the 21st International Conference on Advanced Information Systems Engineering, pp. 246-260.

Dalpiaz, F., Giorgini, P., Mylopoulos, J., 2009b. Software self-reconfiguration: a BDI-based approach. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems – Volume 2, pp. 1159-1160.

Elahi, G., Yu, E., 2009. Modeling and analysis of security trade-offs - a goal oriented approach. Data Knowledge Engineering 68 (7), 579-598.

Fickas, S., Feather, M.S., 1995. Requirements monitoring in dynamic environments. In: Proceedings of the 2nd IEEE International Symposium on Requirements Engineering, pp. 140-147.

Franch, X., Guizzardi, R.S.S., Guizzardi, G., López, L., 2011. Ontological analysis of means-end links. In: Proceedings of the 5th International i* Workshop, pp. 37-42.

Franklin, G.F., Powell, J.D., Naeini, A.E., 2006. Feedback control of dynamic systems, 5th ed. Upper Saddle River, NJ, USA: Prentice-Hall.

Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R., 2002. Reasoning with goal models. In: Proceedings of the 21st International Conference on Conceptual Modeling, pp. 167-181.

Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R., 2003. Formal reasoning techniques for goal models. Journal on Data Semantics I 1 (1), 1-20.

Glinz, M., 2007. On non-functional requirements. In: 15th IEEE International Requirements Engineering Conference, pp. 21-26.

Gordijn, J., Yu, E., Raadt, B.V.D., 2006. E-service design using i* and e3 value modelling. IEEE Software 23 (3), 26-33.

Haley, C.B., Laney, R.C., Moffett, J.D., Nuseibeh, B., 2008. Security requirements engineering: a framework for representation and analysis. IEEE Transactions on Software Engineering 34 (1), 133-153.

Hinz, M., Pietschmann, S., Umbach, M., Meissner, K., 2007. Adaptation and distribution of pipeline-based context-aware Web architectures. In: The Working IEEE/IFIP Conference on Software Architecture, pp. 15-.

Horkoff, J., Yu, E, 2008. Qualitative, interactive, backward analysis of i* models. In: 3rd International i* Workshop, pp. 43-46.

Kazman, R., Klein, M., Clements, P., 2000. ATAM: method for architecture evaluation. Technical Report, CMU/SEI2000-TR-004, Software Engineering Institute, Carnegie Mellon University.

Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36 (1), 41-50.

Kramer J., Magee J., 2009. A rigorous architectural approach to adaptive software engineering. Journal of Computer Science and Technology 24 (2), 183-188.

Lamsweerde, A.V., Letier, E., 2002. From object orientation to goal orientation: a paradigm shift for requirements engineering. In: 9th International Workshop on Radical Innovations of Software and Systems Engineering in the Future, Revised Papers, pp. 325-340.

Lapouchnian, A., Liaskos, S., Mylopoulos, J., Yu, Y., 2005. Towards requirements-driven autonomic systems design. ACM SIGSOFT Software Engineering Notes 30 (4), 1-7.

Lapouchnian, A., Yu, Y., Liaskos, S., Mylopoulos, J., 2006. Requirements-driven design of autonomic application software. In: Proceedings of the 2006 conference of the Centre for Advanced Studies on Collaborative Research, pp. 80-94.

Lapouchnian A., Yu, Y., Mylopoulos, J., 2007. Requirements-driven design and configuration management of business processes. In: Proceedings of the 5th International Conference on Business Process Management, pp. 246-261.

Letier, E., Lamsweerde, A.V., 2004. Reasoning about partial goal satisfaction for requirements and design engineering. In: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 53-62.

Liaskos, S., McIlraith, S.A., Sohrabi, S., Mylopoulos, J., 2010. Integrating preferences into goal models for requirements engineering. In: Proceedings of the 2010 18th IEEE International Requirements Engineering Conference, pp. 135-144.

Lukasiewycz, M., Glaβ, M., Haubelt, C., Teich, J., 2007. Sovling multi-objective pseudo-boolean problems. In: Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing, pp. 56-69.

Menasce, D.A., Dubey, V., 2007. Utility-based QoS brokering in service oriented architectures. In: Proceedings of the IEEE International Conference on Web Services, pp. 422-430.

Mylopoulos, J., Chung, L., Liao, S., Wang, H., Yu, E.S.K., 2001. Exploring alternatives during requirements analysis. IEEE Software 18 (1), 92-96.

Oracle SDN. Java RMI over IIOP. http://java.sun.com/products/rmi-iiop/.

Peng, X., Shen, L., Zhao, W., 2009. An architecture-based evolution management method for software product line. In: Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering, pp. 135-140.

Peng, X., Chen, B., Yu Y., Zhao, W., 2010. Self-tuning of software systems through goal-based feedback loop control. In: Proceedings of the 18th IEEE International Requirements Engineering Conference, pp.104-107.

Sadjadi, S.M., McKinley, P.K., Stirewalt, R.E.K., Cheng, B.H.C., 2004. Generation of self-optimizing wireless network applications. In: Proceedings of the 1st International Conference on Autonomic Computing, pp. 310-311.

Salifu, M., Yu, Y., Nuseibeh, B., 2007. Specifying monitoring and switching problems in context. In: Proceedings of the 15th IEEE International Requirements Engineering Conference, pp. 211-220.

Sebastiani, R., Giorgini, P., Mylopoulos, J., 2004. Simple and minimum-cost satisfiability for goal models. In: Proceedings of the 16th International Conference on Advanced Information Systems Engineering, pp. 20-35.

Shaw, J.A., 2003. The PID control algorithms: how it works, how to tune it, and how to use it, 2nd ed. Process Control Solutions.

Simon, H.A., 1996. The sciences of the artificial, 3rd ed. Cambridge, Mass. MIT Press.

Steere, D.C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., Walpole, J., 1999. A feedback-driven proportion allocator for real-rate scheduling. In: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, pp. 145-158.

Wang, Y., McIlraith, S.A., Yu, Y., Mylopoulos, J., 2009. Monitoring and diagnosing software requirements. Automated Software Engineering 16 (1), 3-35.

Wang, Y., Mylopoulos, J., 2009. Self-repair through reconfiguration: A requirements engineering approach. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, pp. 257-268.

Yu, E.S.K., 1997. Towards modeling and reasoning support for early-phase requirements engineering. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, pp. 226-235.

Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J., Leite, J.C.S.P., 2008. From goals to high-variability software design. In: Proceedings of the 17th International Symposium on Foundations of Intelligent Systems, pp. 1-16.

Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., Leite, J.C.S.P., 2005. Reverse engineering goal models from legacy code. In: Proceedings of the 13th IEEE International Conference on Requirements Engineering, pp. 363-372.