Requirements-driven configuration of software systems

Yijun Yu Alexei Lapouchnian Sotirios Liaskos John Mylopoulos
Department of Computer Science, University of Toronto
{yijun, alexei, liaskos, jm}@cs.toronto.edu

Abstract

Configuring large-scale software to meet different user requirements is a challenging process, since end-users do not know the technical details of the system in the first place. We present an automatic process to connect high-level user requirements with low-level system's configurations. The process takes into account different user preferences and expectations, making configuration easier and more user-centered. Since it reuses a software system's configuration mechanisms, the configuration process is transparent to the system development. Moreover, it is very easy to plug different reasoning frameworks into the configuration process. As a case study, we have reengineered the Mozilla Firefox web browser into a requirements-driven software system, without changing its source code.

1. Introduction

Hardware evolution is governed by Moore's law – *CPU speed doubles every 18 to 24 months* [1]; on the other hand, software evolution is governed by Lehman's laws – especially the 2nd – *increasing complexity* [2]. As a consequence, computer hardware is getting ever-cheaper, e.g., an average workstation is typically a Windows box, which costs no more than $1000. On the other hand, the cost for employing an average developer is more expensive than buying 50 workstations, per year.

As the gap is widening, software maintenance cost dominates the operation of a software company. Managing and using large-scale software systems is becoming a grand challenge, sometimes even a nightmare, as too many parameters are to be configured in order for the software to be working properly by different clients and users. Configuring these is a headache for everyday users: Eclipse IDE, e-mail clients and web browsers such as Mozilla Thunderbird and Firefox, which target at populous and diverse user groups, several Linux kernels and distributions, and, of course, popular commercial software such as Microsoft Windows and Office Suite. These software systems typically contain millions of lines of code. The needs for managing such complex software engender the research in autonomic computing [1, 3].

Figure 1 presents the "Options" dialog window from Mozilla Firefox. A user is asked to provide very low-level details, such as "use TLS 1.0" or "Use SSL 2.0" etc. As shown on the screen, they are related to "Security", but it is not clear whether one should select all of them, one of them, or some combinations of them and how this impacts the attainment of the "Security" goal. Furthermore, what will the side-effects of these selections be on other goals such as "Performance", "Convenience", etc.?

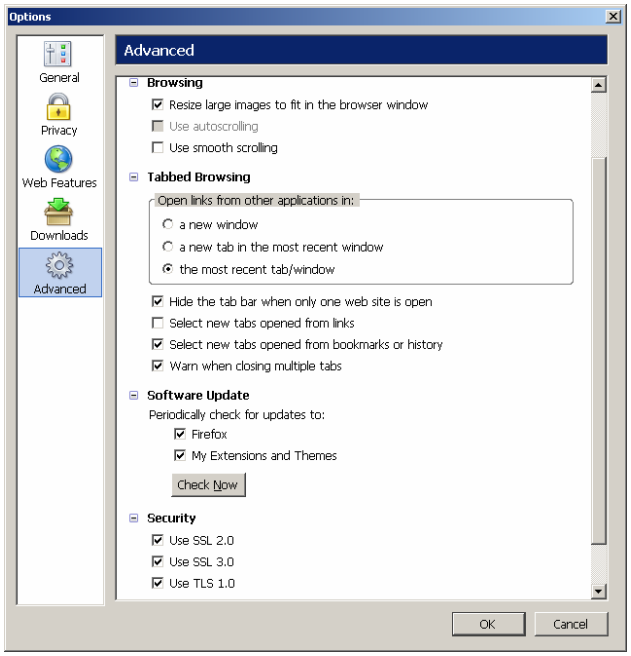

Figure 1. The Options dialog of Firefox

How do we reduce the overhead of controlling large-scale software systems to serve the clients better? How (in case the clients change their requirements) do we agilely reconfigure the software to fit the new client requirements? In this paper, we propose a way to tackle this problem by automating the configuration with goal

models [4]，which has been shown to be possible for a desktop application with an average number of configuration items [5]. Because we consider every individual's requirements in customizing large-scale software, the requirements-driven configuration process is strongly related to the concept of personal and contextual requirements engineering [6, 7]. In [5], for example, user's goals, skills and preferences are proposed as specific personalization criteria for customizing software and tailoring it to particular individuals. On the other hand, since requirements-driven configuration relies on the use of goals [8], a process for generating a goal model that appropriately explains the intentions behind an existing system needs to be considered [3]. In [9, 10], for example this is made possible through reverse engineering directly from the source code.

The process for such automated reconfiguration consists of two major steps. Firstly, one has to set up a goal model in order to connect the user's high-level requirements with the system's low-level configuration items. Secondly, the resulting mapping must be efficiently used by collecting user *preferences* over goals (one goal is more important than another) and *expectations* (a goal needs to be satisfied to a certain degree) and automatically carrying out the configuration.

Using this process, we have successfully configured the Mozilla Firefox browser and the Eclipse IDE for different types of users. The configuration step is fully automated and very efficient, making it well possible for the user to further analyze the resulting system by providing feedbacks.

The remainder of the paper is organized as follows. Section 2 explains the methodology in detail; Section 3 provides implementation details, and Section 4 discusses a case study of the requirements-driven configuration process on the Firefox Web browser. Section 5 discusses further work and concludes the paper.

2. Reengineering into requirements-driven configurable software

The aim of our process is to reengineer a legacy software system, such as Mozilla Firefox, into a requirements-driven reconfigurable system. Therefore, it calls for two necessary steps: (1) *reverse* engineering to understand the legacy system and (2) *forward* engineering to improve the legacy system.

In our case, a legacy system may or may not provide the source code to the reengineer. Thus, we use two kinds of reverse engineering techniques: (1) if the source code is available, the system can be reverse engineered to reveal the implemented goals or purposes of the programmer [11]; otherwise, (2) the system needs to be used and empirically examined in order to discover the alternative ways by which different users may customize the functionality of the system and consequently the alternative ways in which they may want their goals to be fulfilled [10].

Furthermore, once the goal-oriented requirements are obtained, an end user is simply asked to provide their preferences and expectations over the top-level abstract goals. This will drive the software configuration automatically. The degree of automation will depend on how advanced the user is and how much awareness of the low-level configuration details are demanded. Thus, advanced users may employ the method only to obtain a suggestion on how they should configure their system in order to better accommodate their preferences and expectations.

2.1 Reverse engineering for goal models

The objective of reverse engineering in our process is to detect traceability between the low-level implementation with the high-level requirements. Traceability between user goals and the implementation allows the users to understand the system and subsequently configure it in more abstract and less system-oriented way. It is also important to make the user aware of why the system makes certain choices.

In our approach, we do the reverse engineering in two steps:

1. Establish a goal model of the software system;
2. Associate the leaf goals with the configuration items.

A *configuration item* is a variable that can take certain values. A software system can be seen as a huge variability space induced by a large number of configuration items. Some of the configuration items are domain-specific, while others are domain-independent. For example, to configure the look and feel is a taste of the individual, whereas to configure the security task is subject to the software domain. A user's goal model can narrow down the search space by assessing the configuration items.

2.2 Forward engineering with goal models

Having identified the goal models, the objective of forward engineering in our process is to collect individual user preferences and expectations and translate them into software configurations. It is also done in a few steps:

1. **Querying.** Obtain user's preferences and expectations over the high-level goals;

2. **Reasoning.** Convert the user input into satisficing labels of the high-level goals and propagate them downward until leaf goals are reached; Note here the term *satisfice* was used by Herbert Simon [12] to denote the idea of partial satisfaction. The qualitative analysis of the NFR framework [13] is centered on the idea of satisfice.
3. **Configuring.** Convert the leaf goals satisficing labels into values of the configuration items.

Both steps 1 and 3 depend on the software being investigated. During the querying step, a user is asked to either directly provide the preferences and expectations over the goals, or to indirectly provide this information though answering an elicitation questionnaire. The configuring step associates each configuration item with a *default* value in order to attain a certain level of satisfaction for the leaf-level goals.

The reasoning step is independent of the domain of the system to be configured, and is based on the trade-off algorithms discussed in the following section.

3. Implementation

In this section, we briefly discuss the implementation of the methodology. We first describe the reverse engineering approach to establish a goal model. Then, the design of the tradeoff algorithms based on existing goal reasoning algorithms ([8, 14]) is explained. Finally, we show how the query and configuration steps are carried out automatically.

3.1 Reverse engineering

A goal model consists of a set of AND/OR decompositions that *refine* a high-level goal into a set of low-level subgoals. On top of these rules, a set of quantitative *contributions* shows how the satisficing of one goal influences the satisficing of the others. Such a quantification can have probabilistic semantics [8] or it may be cast into a framework of qualitative contribution links. Thus, we can use contribution links such as HELP (+), HURT (-), MAKE (++) or BREAK (--), to show how the satisfaction of the origin goal influences the satisfaction of the target goal.

The source of a goal model can be recovered from the system structure and behavior. In terms of structure, a system/subsystem decomposition paradigm, which follows the *divide and conquer* metaphor, is often a natural match for the AND/OR goal decompositions. For example, inheritance can be seen as the implementation of an OR decomposition of the subject whereas aggregation may be the implementation of an AND decomposition. In terms of behavior, the system achieves certain goals by performing transitions from one state to another. Here, the state/substate hierarchy that can be defined in a statechart has been shown to naturally map to the respective goal/subgoal decomposition graphs[15]. Static program analysis using program slicing techniques can reveal the system's implemented goals [9]. Observing the execution log/trace of the system can also reveal patterns in its dynamic behavior [10]. Combined with a testing framework one can make sure certain functional goals are indeed satisfied [9, 10].

Leaf-level goals may be associated with Boolean predicates on the value of one to many configuration items. For configuration items that are already Boolean, such as "use SSL 2.0" or "use SSL 1.0", such mappings are straightforward. For non-Boolean configuration items, such as a "keeping history record for N days" an extra step is required to find the default value of the configuration item that satisfies the goal. For example, we can represent the leaf-level goal "Keep a good record of my web surfing history" as a Boolean predicate "N >= 5", and associate the fully satisficed value of the goal with "N=10" and the fully denied value of the goal with "N=0". This way, a direct mapping is set from the configuration of domain-specific parameters to the configuration of the goal model.

3.2 Tradeoff algorithms

When a goal is decomposed into multiple alternatives (OR-subgoals), the contribution of each subgoal to the satisfaction of top-level goals can be compared with the expectations and preferences, in order to rate the choices and thus make decisions.

Bottom-up reasoning propagates the labels that describe the degree of satisfaction of leaf goals upwards to obtain the corresponding labels for the top-level goals [8]. This can be used to validate the requirements.

Top-down reasoning propagates the labels of the top-level goals downwards to obtain the labels for the minimal number of leaf-level goals [14]. This can be used to predict the minimal configuration that can satisfy the user's requirements. Since the top-down reasoning relies on a *satisfiability problem*[1] (SAT) solver [16, 17] which deals with binary propositions, it is important to design an encoding mechanism such that at least discrete labels (full/partial satisficing/denial) of goals can be translated into the binary propositions.

3.3 User interface and questionnaire design

An interface to the configuration system consists of a dialog and/or a questionnaire wizard. In the dialog, each

[1] That is, deciding whether a given Boolean formula in conjunctive normal form has an assignment that makes the formula "true."

top-level hard goal is presented as a *checkbox*, whereas each top-level softgoal (e.g. performance, security, usability) is presented as a *slider* by which the satisficing expectation is set. Preferences are shown by the order of the sliders from top to bottom. Although a slider-based user interface design can directly present the needed input, it is not guaranteed that all the user's expectations can be met by the system design *at the same time*. For example, a full satisfaction of performance, security, maintainability and usability goals is simply impossible. The interdependency and constraints among these goals are defined by the underlying goal model. Thus we also designed an alternative wizard to ask user a set of elicitation questions in order to derive the expectations and preferences with respect to the goals. In these questions, we avoid using technical terms, rather, using familiar terms to everyday user. For example, "Are you using the browser with a public-domain computer?" The simple Yes/No answer to such questions can lead to elicited preference such as whether "Privacy" is important or not. Thus for elicitation, we can use a goal model which connect the preferences/expectations of the high level goals with answers to concrete questions at the leaf level and use bottom-up label propagation to obtain the preference/expectation labels as an input for the configuration step.

3.4 Configuration step

The configuration of the system is done automatically. First, the software system is analyzed for its configurability in terms of whether there exists a persistent record of the configuration (if our configurator interacts with the subject software through a file interface) or an in-memory API for its configuration (if our configurator interacts with the subject software directly through APIs).

Based on the configuration in the goal model (the selected leaf-level goals), a script is generated to populate the configuration data with the default values associated with the leaf-level goal satisfaction labels. Since the reverse engineering step has already produced the appropriate mapping, this task is now quite straightforward. The last step is to automate the reconfiguration by running the script, either before restarting the subject software or during the execution of the software system.

4. Firefox: a case study

We represent user high-level requirements in an XML-based input language, as follows.

```
<input:model>
<soft name= "Performance">
        <rule op="AND"/>
        <soft name= "Browsing Performance"/>
        <soft name= "System Performance"/>
</soft>
<soft name= "Usability">
    <rule op="OR"/>
    <soft name= "Ease of Search"/>
    <soft name= "Convenient access to Information"/>
    <soft name= "User Tailorability">
        <rule op="OR"/>
        <soft name= "Programmability"/>
          <soft name= "User Flexibility"/>
    </soft>
</soft>
<soft name= "Security">
    <rule op="HURT" target="System Performance"/>
    <rule op="HURT" target="Browsing Performance"/>
</soft>
<soft name= "Allow changes in Content Appearance">
    <rule op="HELP" target="User Flexibility"/>
</soft>
<goal name= "Filter Advertisement/Spyware/Popups">
    <rule op="HELP" target="Performance"/>
    <rule op="HELP" target="Security"/>
    <rule op="HURT" target="Content Availability"/>
</goal>
</input:model>
```

In this input language, a *model* is given by a list of root *goal*s which are recursively decomposed in a nested XML element structure. A *soft*goal is a goal that can be satisfied to a degree less than 1. It usually represents quality attributes. A number of *rule*s show what kind of decomposition was used for a goal or softgoal, or which kind of contributions was used between a *source* hardgoal and a *target* softgoal.

Each user provides a profile including the preferences and expectations for the softgoals:

```
<input:profile>
<soft name="Security" rank="4" value="6" />
<soft name="Allow Interactive Content" rank="8" value="8"
/>
<soft name="Convenient Access to Information" rank="10"
value="10" />
<soft name="Performance" rank="9" value="1" />
<soft name="Content Availability" rank="1" value="10" />
<soft name="Allow changes in Content Appearance"
rank="6" value="4" />
<soft name="User Flexibility" rank="3" value="6" />
<soft name="Speed" rank="7" value="3" />
<soft name="Programmability" rank="3" value="8" />
<soft name="Modularity" rank="5" value="1" />
<soft name="Usability" rank="2" value="6" />
</input:profile>
```

For every root softgoal, a *rank* attribute represents the partial order among the preferences and a threshold *value*

represents the expectation from the user. The profile can be generated from a user interface dialog (Figure 2).

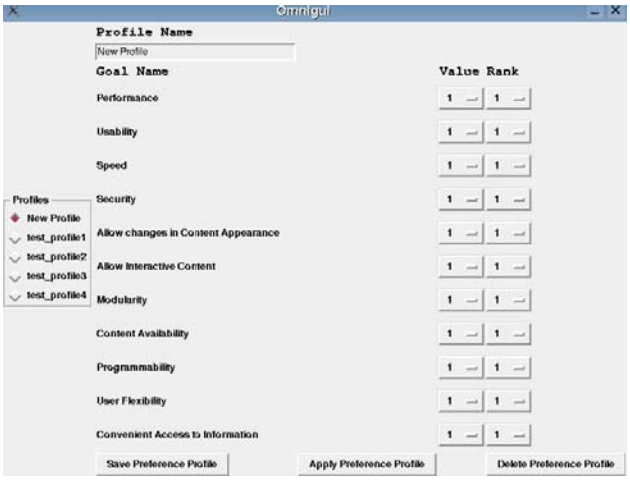

Figure 2. A simplified user preference dialog as the interface to the configurator

The reasoning algorithm is invoked by the configurator command automatically, to produce an output as follows:

```
<output:configuration>
<goal name="adFilterStrength" value="on" />
<goal name="tabBrowsingOn" value="off" />
<goal name="cookiesEnabled" value="off" />
<goal name="daysToCachePages" value="on" />
</output:configuration>
```

The goal model can be visualized as a goal graph and the reasoning can be invoked and its results shown in OpenOME [18], our requirements engineering tool, where both bottom-up and top-down goal reasoning algorithms are implemented and can be invoked by the two buttons on the toolbar (Figure 3). Behind the scenes, an XSLT script fully automatically generates the corresponding property configuration in the Firefox default installation directory.. The following JavaScript script code is an example of such property configuration:

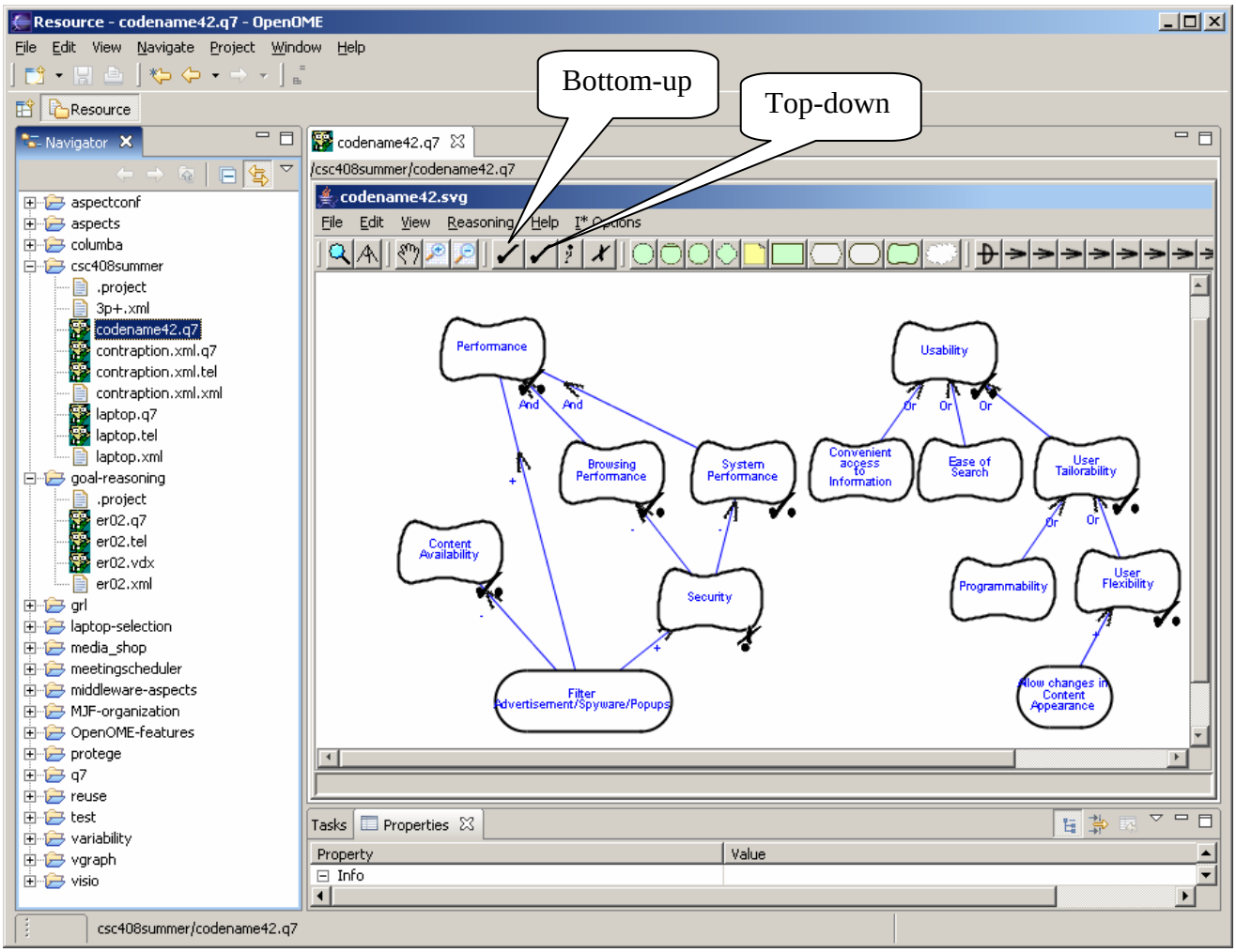

Figure 3. The goal model and its reasoning in OpenOME, an Eclipse plugin for requirements engineering

```
user_pref("network.image.imageBehavior", 2);
user_pref("network.cookie.cookieBehavior", 2);
user_pref("webdeveloper.disabled", false);
user_pref("browser.display.use_document_colors", true);
user_pref("javascript.enabled", false);
user_pref("webdeveloper.disabled", false);
user_pref("adblock.enabled", true);
user_pref("tidy.options.browser_disable", false);
user_pref("font.size.variable.x-western", 19);
user_pref("image.animation_mode", "normal");
user_pref("extensions.prefbar.display_on", 0);
user_pref("security.enable_java", false);
user_pref("security.default_personal_cert", "Select
Automatically");
user_pref("browser.cache.disk.enable", false);
```

5. Conclusion

Through the Mozilla Firefox case study we show how goal-oriented requirements can be used to guide the configuration process automatically. The goal models are provided by domain experts, the user profiles are obtained by the users directly through a simplified user interface, and the configuration is carried out without further human intervention. Currently, we are investigating how to apply the requirements-driven configuration mechanism to other applications and how to detect problems that reconfiguration may cause when it is performed while the software system is running. We also plan to implement a Firefox extension plugin to expose our tool to the larger user community and to solicit feedback from users.

6. Acknowledgement

Much of the implementation is done by all of our 26 undergraduate students in the Software Engineering course offered in the summer of 2005 at the University of Toronto [11]. Some examples in this paper are taken from the *codename42* project team including Dimitri Stroupine, Faiz Hemani, Hareem Arif, Sani Hashmi and Zia Malik. The authors would also like to thank Xin Gu for giving XSLT tutorial to the students.

7. References

[1] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Syst. J.*, vol. 42, pp. 5-18, 2003.

[2] M. M. Lehman and J. F. Ramil, "Evolution in software and related areas," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*. Vienna, Austria: ACM Press, 2001.

[3] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu, "Towards requirements-driven autonomic systems design," in *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*. St. Louis, Missouri: ACM Press, 2005.

[4] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," in *Selected Papers of the Sixth International Workshop on Software Specification and Design*: Elsevier Science Publishers B. V., 1993.

[5] B. Hui, S. Liaskos, and J. Mylopoulos, "Requirements Analysis for Customizable Software Goals-Skills-Preferences Framework," in *Proceedings of the 11th IEEE International Conference on Requirements Engineering*: IEEE Computer Society, 2003.

[6] A. Sutcliffe, S. Fickas, and M. M. Sohlberg, "Personal and Contextual Requirements Engineering," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering*: IEEE Computer Society, 2005.

[7] S. Fickas, "Clinical requirements engineering," in *Proceedings of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM Press, 2005.

[8] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Reasoning with Goal Models," in *Proceedings of the 21st International Conference on Conceptual Modeling*: Springer-Verlag, 2002.

[9] Y. Yu, Y. Wang, S. Liaskos, A. Lapouchnian, and J. Mylopoulos, "Reverse Engineering Goal Models from Legacy Code," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering*: IEEE Computer Society, 2005.

[10] S. Liaskos, A. Lapouchnian, Y. Wang, Y. Yu, and S. Easterbrook, "Configuring Common Personal Software: A Requirements-Driven Approach," in *Proceedings of the 13th IEEE International Conference on Requirements Engineering*: IEEE Computer Society, 2005.

[11] Y. Yu, A. Lapouchnian, S. Liaskos, and X. Gu, "The Software Engineering summer course (CSC408H1S)," *http://www.cdf.toronto.edu/~csc408h/summer*, 2005.

[12] H. A. Simon, *The Sciences of the Artificial*: Massachusetts Institute of Technology, 1996.

[13] L. Chung, B. A. Nixon, E. S. K. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*. Boston Hardbound: Kluwer Academic Publishers, 1999.

[14] R. Sebastiani, P. Giorgini, and J. Mylopoulos, "Simple and Minimum-Cost Satisfiability for Goal Models," in *Proc. CAiSE*: LNCS, 2004.

[15] Y. Yu, J. Mylopoulos, A. Lapouchnian, S. Liaskos, and J. C. S. d. P. Leite, "From stakeholder goals to high-variability software designs," University of Toronto CSRG-509, 2005.

[16] S. A. Cook and D. G. Mitchell, "Finding Hard Instances of the Satisfiability Problem: A Survey " in *Satisfiability Problem: Theory and Applications*, vol. 35, *Discrete Mathematics and Theoretical Computer Science*, J. G. Dingzhu Du, and Panos M. Pardalos, Ed.: American Mathematical Society, 1997, pp. 1-17.

[17] D. Le Berre, A. Parrain, O. Roussel, and L. Sais, "SAT4J: A satisfiability library for Java," 2005.

[18] Y. Yu, E. S. K. Yu, L. Liu, and J. Mylopoulos, "The OpenOME requirements engineering tool," in *http://www.cs.toronto.edu/km/openome*, 2005.