Goal-based Behavioral Customization of Information Systems

Sotirios Liaskos, Marin Litoiu, Marina Daoud Jungblut, John Mylopoulos

Technical Report CSE-2010-10

November 7th 2010

Department of Computer Science and Engineering
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada

Goal-based Behavioral Customization of Information Systems

Sotirios Liaskos
School of Information Technology
York University
Toronto, Canada
liaskos@yorku.ca

Marin Litoiu
School of Information Technology
York University
Toronto, Canada
mlitoiu@yorku.ca

Marina Daoud Jungblut
Dept. of Computer Science
York University
Toronto, Canada
djmarina@yorku.ca

John Mylopoulos
Dept. of Computer Science
University of Toronto
Toronto, Canada
jm@cs.toronto.edu

ABSTRACT

Customizing software to perfectly fit individual needs is becoming increasingly important in information systems engineering. Users want to be able to customize software behavior through reference to terms familiar to their experience and needs. In this paper, we present a requirements-driven approach to behavioral customization of software. Goal models are constructed to represent alternative behaviors that users can exhibit to achieve their goals. Customization information is then added to restrict the space of possibilities to those that fit specific users, contexts or situations. Meanwhile, elements of the goal model are mapped to units of source code. This way, customization preferences posed at the requirements level are directly translated into system customizations. Our approach, which we apply to an on-line shopping cart system, does not assume adoption of a particular development methodology, platform or variability implementation technique and keeps the reasoning computation overhead from interfering with execution of the configured application.

Keywords

Information Systems Engineering, Goal Modeling, Adaptive Systems, Software Customization

1. INTRODUCTION

Adaptation is emerging as an important mechanism in engineering more flexible and simpler to maintain and manage information systems. To cope with changes in environment or in user requirements, adaptive software changes its structure and behavior based on changes in the environment and requirements [22, 17]. An important manifestation of adaptivity is the ability of individual organizations and users to *customize* their software to their unique and changing needs in different situations and contexts.

Consider, for example, an on-line store where users can browse and purchase items. Normally, an anonymous user can browse the products, view their price information and user comments, add them to the cart, log-in and check-out. But different shop-owners may want variations of this process for different users. They may need, for example, to withhold prices, user comments or other product information unless the user has logged in, or only if the user's IP belongs to a certain set of countries. Or they may wish to rearrange the sequence of screens that guide the buyer through the check-out process. Or, finally, they may wish to disable purchasing and allow just browsing, with only some frequent buyers allowed to add comments – with or without logging in first. The shop-owner would like to be able to devise, specify and change such rules every time she feels it is necessary and then just observe the system reconfigure appropriately without resorting to expert help. But how easy is this?

Satisfying a great number of behavioral possibilities and switching from one to the other is a challenging systems engineering problem. While there is significant research on modeling and implementing variability and adaptation, e.g. in the areas of Software Product-Lines and Adaptive Systems, two aspects of the problem seem to still require more attention. Firstly, the need to easily communicate and actuate the desired customization, using language and terms that reflect the needs and experience of the stakeholders, such us the shop owner of our example. Secondly, the need to allow the stakeholders to construct their customization preferences themselves, instead of selecting from a restricted set of predefined ones, allowing them thus to acquire a customization that is better tailored to their individual needs.

To address these issues, in this paper we introduce a goal-driven technique for customizing the behavioral aspect of a software system. A generic goal-decomposition model is constructed to represent a great number of alternative ways by which human agents can use the system to achieve their goals. Then, to address their specific needs and circumstances, individual stakeholders can refine this model by specifying additional constraints to the ways by which human and machine actions are selected and ordered in time. A preference-based AI planner is used to calculate such admissible behaviors and a tree structure representing these behavioral possibilities is constructed. Meanwhile, the high-level descriptions of these actions are connected with code fragments of the software system, in a way that the tree structure can actually enforce the desired system behavior. This way, high-level expressions of desired arrangements of user actions are automatically translated into behavioral configurations of the software system. Amongst the benefits of our approach are both that it brings the customization practice to the requirements level and that it allows leverage of larger number of customization possibilities in a flexible way, without imposing restrictions to the choice of development process, software architecture or platform technology.

Our proposal constitutes a continuation of our earlier work on exploiting goal model variability – i.e. alternative ways to achieve stakeholder goals – to support software customization. In that early work [19, 20], we have demonstrated the usefulness of specifying and reasoning about stakeholder preferences in selecting requirements alternatives that best fit to the needs of specific users, contexts and situations. In this paper, we take this modeling and reasoning infrastructure one step further to allow requirements-based, preference-driven customization of the actual software system.

The paper is organized as follows. In Sections 2 and 3 we present the core goal modeling language and the temporal extension that we are using for representing and leveraging behavioral alternatives. In Section 4 we show how the elements of the goal model relate with elements of the source code. Then, in Section 5 we show how the result of alternatives analysis at the goal level enables the appropriate customization at the system level. We discuss the feasibility of our approach in 6. Finally, in Section 7 we discuss related work and conclude in Section 8.

2. GOAL MODELS

Goal models [11, 29] are known to be effective in concisely capturing alternative ways by which high-level stakeholder goals can be met. This is possible through the construction of AND/OR goal decomposition graphs. Such a graph can be seen in Figure 1. The model shows alternative ways by which an on-line store can be used for browsing and purchasing products.

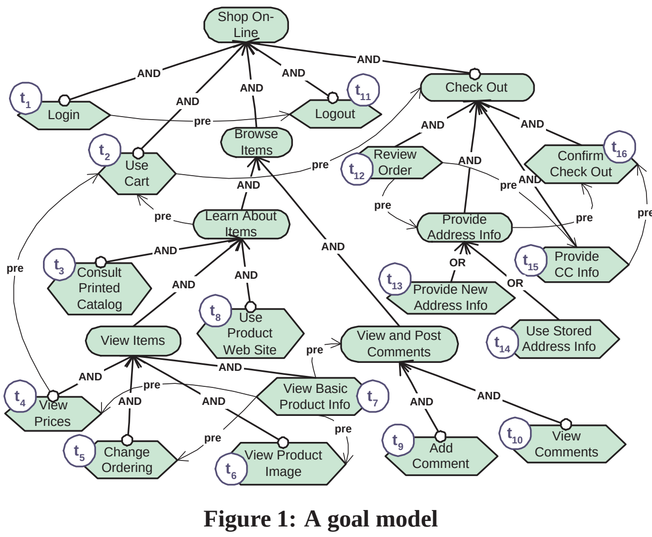


Figure 1: A goal model

The graph consists of *goals* and *tasks*. Goals – the ovals in the figure – are states of affairs or conditions that one or more actors of interest would like to achieve [29]. Thus *Browse Items* is a goal. Tasks on the other hand – the hexagonal elements – describe particular low-level activity that the actors perform in order to fulfill their goals. For example *View Basic Product Info* is a task. To ease our presentation, next to each task shape, a circular annotation containing a literal of the form t_i has been added. In the rest of the paper, we will often use the literal to refer to the task instead of mentioning the entire task description; thus t_7 refers to the task *View Basic Product Info*. Goal models can also contain soft-goals – goals for which there is no clear-cut criterion to decide whether they are satisfied or not. While soft-goals can play a significant role in customization [19], for simplicity we omit any reference to these in this paper.

Goals and tasks are connected with each other using AND- and OR-decomposition links. When a goal is OR-decomposed then satisfaction (performance) of one of the subgoals (resp. subtasks) suffices for the satisfaction of the parent goal. For example, the goal *Provide Address Info* is decomposed into *Provide New Address Info* and *Use Stored Address Info* as these are two alternative ways to specify address information. When a goal is AND-decomposed into subgoals or subtasks, then all of the subgoals (subtasks) must be satisfied (resp. performed) in order for the parent goal to be considered satisfied. In addition, children of AND-decompositions can be designated as *optional*. This is visually represented through a small circular decoration on top of the optional goal. In the presence of optional goals, the definition of an AND-decomposition is refined to exclude optional sub-goals from the goals that must necessarily be met in order for the parent goal to be satisfied. For example, for the goal *View Items* to be fulfilled, the task *View Basic Product Info* is only mandatory – tasks *View Prices*, *Change Ordering* and *View Product Image* may or may not be chosen to be performed by the user.

Tasks can be classified into two different categories depending on what the system involvement is in terms of their performance. Thus, *human-agent* tasks are to be performed by the user alone without the support or other involvement of the system under consideration – an external system outside the scope of the analysis may be used though. For example *Consult Printed Catalog* (t_3) belongs to this category because it is performed without involvement of the system. On the other hand, *mixed-agent* tasks are tasks that are performed in collaboration with the system under consideration. Thus *Add Comment* is a mixed-agent task as the user will add the comment and the system will offer the facility to do so. Another example of a mixed-agent task is *View Image*: the system needs to display an image and the user must view it in order for the task to be considered performed. All tasks of Figure 1 are mixed-agent except for t_3 and t_8 which are human-agent tasks.

By recursively decomposing the root goal into subgoals and eventually tasks, we are able to concisely represent alternative ways by which actors can go about fulfilling the goals. These are simply solutions of the AND/OR tree; the presence of optional and alternative sub-goals allows for solution variability to be represented.

3. ADDING A TEMPORAL DIMENSION

3.1 Indicative Constraints

The order by which goals are fulfilled and tasks are performed is relevant in our framework. To express constraints in satisfaction ordering we use the *precedence link* ($\xrightarrow{pre}$). The precedence link is drawn from a goal or task to another goal or task, meaning that satisfaction/performance of the target of the link cannot begin unless the origin is satisfied or performed. For example the precedence link from the task *Use Cart* (t_2) to the goal *Check Out* implies that none of the tasks under *Check Out* can be performed unless the task *Use Cart* has already been performed.

Given the relevance of ordering in task fulfillment, solutions of the goal model come in the form or *plans*. A plan for the root goal is a sequence of leaf level tasks that both satisfy the AND/OR decomposition tree and possible precedence links. Thus, plans exemplify possible behaviors that actors can exhibit when trying to achieve their goals. In plan $[t_1, t_7, t_4, t_2, t_{12}, t_{14}, t_{15}, t_{16}, t_{11}]$ for example, the user logs-in, browses the products with their prices, adds some of them to the cart and then checks out. In plan $[t_1, t_7, t_4, t_9, t_{10}, t_2, t_{12}, t_{14}, t_{15}, t_{16}, t_{11}]$, the user also views and adds comments. However sequences $[t_1, t_4, t_2]$ and $[t_{11}, t_7, t_2, t_{12}, t_{15}, t_{14}, t_1, t_{16}]$ are not valid plans for the root goal, the former because it does not satisfy the tree structure and the latter because it violates precedences.

This variability of plans that a goal model implies can be understood as a representation of the variety of *behaviors* that an actor could exhibit in order to achieve their goals. It is important to notice that these behaviors do not necessarily depend on the system, which at the point in the lifecylce when goal models are produced has not yet been defined. For example, an actor may purchase a product without looking at or adding any user comments. At a different situation the same or a different actor may actually spend time reading the comments. In a third scenario she also adds a comment to the product. These are different behaviors that the user exhibits that do not necessarily imply any variability in the software system per se: the comment viewing and addition feature may be available in all cases, but used only in some of them. Thus, with goal models we focus on variability at the level of possible human behaviors rather than software variability, at the level of e.g. allowable component configurations.

3.2 Adding Customization Constraints

The temporally extended goal model with its precedence links is intended to be the most unconstrained and behaviorally rich model of the domain at hand. Indeed, the goal model of Figure 1 allows the user to follow a large variety of ways to go about fulfilling the root goal, as long as they are physically possible and reasonable.

However the shop owner may wish to restrict certain possibilities. For example, she may want to disallow the user to view the prices unless he logs in first or prevent the user from viewing and/or adding comments, before logging in or in general. She may even go on to disallow use of the cart, again prior to logging in or even for the entire session. In the last case, this would effectively imply turning the system into a tool for browsing products only.

To express additional constraints on how users can achieve their goals we augment the goal model with the appropriate *customization formulae* (CFs). CFs are formulae in linear temporal logic (LTL) grounded on elements of the goal model. Different stakeholders in different contexts and situations may wish to augment the goal model with a different set of CFs, restricting thereby the space of possible plans to fit particular needs.

To construct CFs we use 0-ary predicates of the form *useCart* and *browseItems* to denote satisfaction of tasks and goals. These predicates are true iff the task or goal they represent is performed or satisfied, respectively. Furthermore, symbols $\Box$,$\Diamond$, $\circ$ and U are used to represent the standard temporal operators *always*, *eventually*, *next* and *until*, respectively.

Using CFs we can represent interesting temporal constraints that performance of tasks or satisfaction of goals must obey. Back to our on-line shop example, assume that the shop owner would like to disallow certain users from browsing the products without them having logged in first. This could be written as a CF as follows:

$$\neg\ \mathit{viewBasicProductInfo}\ U\ \mathit{login}$$

The above means that the task *View Basic Product Info* (t_7) should not be performed (signified by predicate *viewBasicProductInfo* becoming true) before the task *Login* (t_1) is performed (thus, predicate *login* becoming true). For another class of users there may be a more relaxed constraint:

$$\neg\ \mathit{viewPrices}\ U\ \mathit{login}$$

Universal and existential constraints are also relevant. For example the shop owner may want to disallow users from adding comments, thus:

$$\Box\neg \mathit{addComment}$$

If, in addition to these, she wants to prevent them from viewing prices, logging in and using the cart, this translates into a longer conjunction of universal properties seen in Figure 2. In effect, with the property of the figure the shop owner allows the users to only browse the products, their basic information and their images.

$$\boxed{\begin{array}{c}(\Box\neg\ \mathit{addComment}\ ) \wedge (\Box\neg\ \mathit{viewPrices}) \wedge \\ (\Box\neg\ \mathit{login}) \wedge (\Box\neg\ \mathit{useCart})\end{array}}$$

Figure 2: A Customization Formula

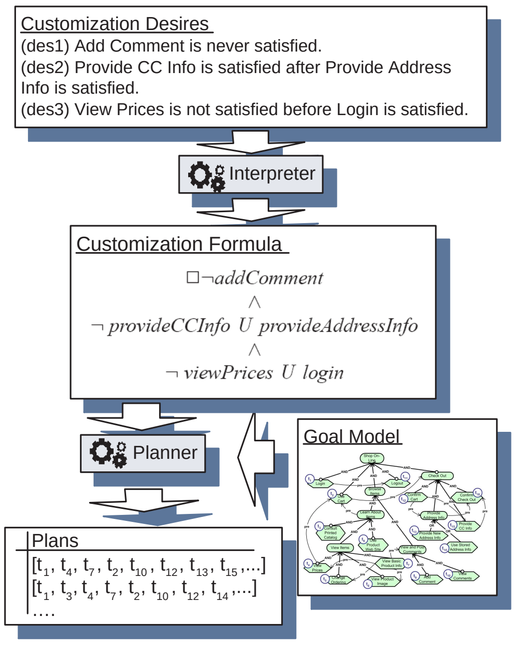
While CFs, as LTL formulae, can in theory be of arbitrary complexity, we found in our experimentation that most CFs that are useful in practical applications are of specific and simple form. Thus simple existence, absence and precedence properties are enough to construct useful customization constraints. Hence, LTL patterns such as the ones introduced in ([12]), can be used to facilitate construction of CFs without reference to temporal operators.

In our application, we used the LTL pattern system through templates in structured language. Thus, CFs can be expressed in forms such as *"h_1 is [not] satisfied before/after h_2 is satisfied"*, to express precedence and response as well as *"h is eventually [not] satisfied"* to express existential properties, where h, h_1, h_2 are goals or tasks of the goal model. A simple interpreter performs the translation of such customization desires into actual LTL formulae. In this way, construction of simple yet useful CFs is possible by users who are not trained in LTL.

3.3 Identifying Admissible Plans

Adding CFs significantly restricts the space of possible plans by which the root goal can be satisfied. Given a CF, we call the plans of the goal model that satisfy the CF *admissible plans* for the CF. Thus, all $[t_7]$, $[t_7, t_5]$,$[t_7, t_{10}, t_6]$, $[t_8, t_7, t_6, t_5]$ and $[t_3, t_7, t_{10}]$ are examples of admissible plans for the CF of Figure 2. However, plan $[t_1, t_7, t_4, t_9, t_2, t_{12}, t_{14}, t_{15}, t_{16}, t_{11}]$, although it satisfies the goal model and its indicative precedence constraints, it is not admissible because it violates the CF – all its conjuncts actually.

To allow the identification of plans that satisfy a given CF, we are adapting and using a preference-based AI planner, called PPLan [5]. The planner is given as input a goal model, appropriately and fully automatically translated to a planning problem specification as well as a CF and returns the set of all admissible plans for the CF. Unless interrupted, the planner will continue to immediately output plans it finds until there are no more such. Details on how the planner is adapted can be found in [19].



Customization Desires
(des1) Add Comment is never satisfied.
(des2) Provide CC Info is satisfied after Provide Address Info is satisfied.
(des3) View Prices is not satisfied before Login is satisfied.
Interpreter
Customization Formula
$\Box\neg addComment$
$\wedge$
$\neg\ provideCCInfo\ U\ provideAddressInfo$
$\wedge$
$\neg\ viewPrices\ U\ login$
Planner
Goal Model
Plans
$[t_1, t_4, t_7, t_2, t_{10}, t_{12}, t_{13}, t_{15}, \ldots]$
$[t_1, t_3, t_4, t_7, t_2, t_{10}, t_{12}, t_{14}, \ldots]$
....

Figure 3: From Customization Desires to Plans

The overall framework and toolset for identifying sets of preferred plans can be summarized in Figure 3. The high-level customization desires expressed in structured English are interpreted into a customization formula, which together with the (appropriately formalized) goal model are, in turn, given as input to a planner. The planner returns plans of the goal model and satisfy the given customization constraints. This process has been found to be useful for allowing users to identify solution alternatives of their goals that best fit their individual needs [19, 20]. In the rest of the paper we show how such preferred requirements alternatives can be interpreted into preferred customizations of the actual system.

4. CONNECTING GOAL MODELS WITH CODE

To enable interpretation of preferred plans into preferred software customizations, a way to connect tasks with elements of source code needs to be established. The literature offers some proposals for goal-driven software development methodologies (e.g. [23]), whereby adoption of a particular implementation technology or architectural approach (e.g. agent- or component-orientation) and, of course, of a particular development process is required. Our goal, however, is to introduce a framework for enabling requirements-driven customization that also has the minimal impact to the way that developers develop their software. Hence we wish to avoid introducing our own variability implementation or software composition technique or restrict ourselves to an existing one from the wealth that has been proposed in the literature [13]. Instead, we identify a minimum set of principles, which, if applied during development, our framework becomes applicable. These principles refer to *task separation* and *task instrumentation*, explained below.

Task Separation. For every mixed-agent task in the goal model there exists a set of statements which are dedicated to exclusively supporting that task – and, thus, serve no other purpose. Furthermore, it should be possible to prevent these statements from executing, preventing in effect the user from performing the task. There is no requirement that these statements are located in the same part of the implementation and not scattered across components, modules, classes etc. – thus the principle is not a suggestion of task-oriented modularization. We call this code *mapped code (fragment)* to the task. Back in the on-line cart example, the mapped code for task *Login* is the code for drawing the username and password text boxes as well as the “Submit” and “Clear” button on the user screen. This code exists exclusively for allowing the user to perform this task. Not drawing those widgets, through conditioning the mapped code, effectively prevents execution of the task. As we will see, we found that the mapped code is predominantly code that conveniently exists in the view layer of an application.

Task Instrumentation Points. For every mixed-agent task, there is a location in the source code where the state of the system suggests that a task has been performed. In the *Login* example this might be the point in which confirmation that the login credentials are correct is sent back from the database and the application is ready to redirect control elsewhere. In the task *Review Order*, this can be the point where a summary of the order has been displayed on the screen – and we assume that the user has successfully performed the subsequent reviewing task.

The above principles are deliberately general and informal so that they can be easily refined and applied in a variety of composition and variability implementation scenarios. In a component-based or service-oriented design, for example, the mapped code of each task can be hidden behind an interface, which may or may not be used by the calling environment [30]. In an aspect-oriented application, on the other hand, modularization need not follow task separation. Instead tasks can be written as advice to be weaved (or not) in appropriate locations in the source code. Later in the paper, drawing from our case study with the on-line cart system, we discuss more on the nature of the mapped code and show how fulfilling the above principles turned out to be a very natural process.

5. IMPLEMENTING CUSTOMIZATION POLICY

Let us now look at how we can use the above mapping from tasks to code together with the goal and CF models in order to enable the appropriate behavioral result in the software system. We do this by: a) constructing a policy tree that represents all available ways by which a customized behavior can unfold, b) conditioning each mapped code fragment based on what the policy tree allows, and c) appropriately instrumenting the source code to sense which tasks have been performed and update the policy tree accordingly. We look these in detail in the following subsections.

5.1 Constructing and Using The Policy Tree

As we saw, the goal model implies a great number of alternative plans of fulfilling the root goal. The addition of CFs dramatically decreases the number of those plans into a set of admissible ones with respect to the CF. The policy tree is simply a concise representation of those admissible plans – with the difference that it includes only the mixed-agent tasks. An example can be seen in Figure 4. In the figure, task literals refer to those of Figure 1. On the right of the figure, a policy tree is seen that corresponds to a list of admissible plans on the left. Each node of the tree represents a task. Let plan prefix p_k of a plan p be the first k tasks of the sequence $[t_1, t_2, \ldots, t_n]$ that comprise p, where $k \leq n$. Given a set of plans P for the goal model, a policy tree is a representation of P such that: p_k is a prefix of a plan $p \in P$ if and only if there is a node in the policy tree for which the path from the root to that node yields p_k. Thus, in Figure 4, every plan (or prefix thereof) from the plan list corresponds to a path from the root to a leaf (resp. intermediate node) of the policy tree. Conversely, every node in the policy tree is associated with a plan prefix of the plan list, which can be found by traversing the tree from the root to that node. If the node is leaf level then the prefix is a complete plan of the plan list.

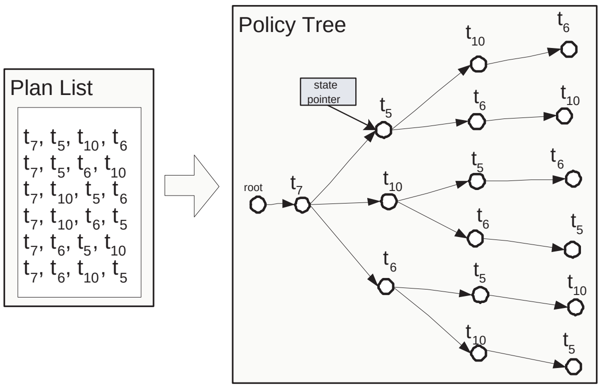


Figure 4: A policy tree and the corresponding list of tasks

The policy tree is also supplied with a pointer that points to one of the nodes of the tree. We call this the *state pointer*. The role of the state pointer is to maintain information about what tasks have been performed in a given use scenario at run time. Thus, the state pointer pointing to a given node means that the plan prefix associated to that node has already been performed. On the other hand, the tasks that can possibly be performed from that point are restricted to the children of the node currently pointed at, or any of the tasks in the associated prefix – in a sense that these tasks can be repeated.

In the example policy tree of Figure 4 the state pointer points to node t_5 (*Change Ordering*), meaning that this was the last new task that was observed to be performed. The node is associated with prefix $[t_7, t_5]$ (t_7 being *View Basic Product Info*) meaning that these are the tasks that are assumed to have been performed by the user so far. Each of them can be repeated by the user. As for new tasks, the user can perform t_{10} or t_6 (the two children of t_5), that is *View Comments* or *View Product Image*. However, the user cannot add any products to the cart as none of the children of t_5 is t_2 (*Use Cart*).

```
INPUT: a set of preferred plans $P$
OUTPUT: a policy tree rooted at $root$
$root :=$ new node
$label(root) := [empty]$
for each new plan /*for each new plan $p = [t_1, t_2, ...] \in P$ */
    $currentNode := root$
    loop $j$ /*for each task $t_j$ in $p$*/
        if $t_j$ is not a mixed-agent task skip to next j
        if $\exists$ child $c$ of $currentNode$ such that $label(c) == t_j$ then
            $currentNode := c$
        else
            $c :=$ new node
            $label(c) := t_j$
            set $c$ to be the child of $currentNode$
            $currentNode := c$
        end if
    end loop /*for each task in plan $p$*/
end for each /*for each new plan $p$ */
```

Figure 5: Building the Policy Tree

An algorithm for constructing a policy tree, from a list of admissible plans that the planner returns, is shown in Figure 5. Observe that appending a new plan to the tree is linear to the length of the plan. It is important to note that the algorithm can keep augmenting the tree as new plans are generated by the planner, which allows use of partial output of the planner and enrichment of the tree as new plans arrive.

5.2 Conditioning and Instrumenting the Source Code

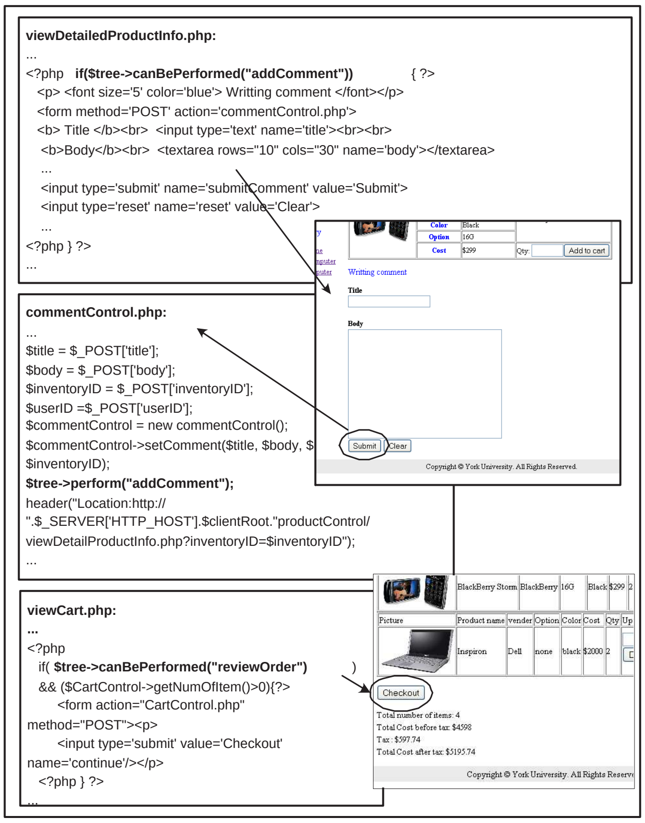
Let us now see how the policy tree can be used to enable behavioral customization of the software system. Recall that the system is build following the principles of task separation and task instrumentation. This means that, on one hand, each mixed-agent task is associated with a set of statements (the mapped code) whose removal can prevent execution of the task, and on the other hand, for each task there is a well defined location in the code that marks completion of the task. The policy tree is integrated by conditioning access to the mapped code based on the position of the state pointer, and by adding statements in the instrumentation points that advance the position of the state pointer accordingly.

More specifically, the former is implemented through the use of the function *canBePerformed(t)*. The function *canBePerformed(t)* returns true iff task *t* is one of the children of the node currently pointed at by the state pointer or part of the associated prefix. In other words, the code fragment can be entered only if the new plan prefix that would result from performing the task that maps to that fragment belongs to at least one of the admissible plans. For example the mapped code of the task *Use Cart* involves buttons for adding items to the cart, text fields for specifying quantities, links for viewing the cart content etc. All these will be displayed only if *canBePerformed(useCart)* is true, that is the task *Use Cart* is in one of the children of the state pointer, or it is part of the path from the root to the state pointer. If this is not the case, the mapped code will not be accessed, preventing rendering of the user interface elements, which in turn prevents performance of the task by the user.

Advancement of the position of the state pointer, on the other hand, is implemented through simple *perform(t)* statements inserted in the instrumentation points, where *t* is the task that was just performed. The effect of the *perform(t)* statement is that the state pointer advances to the child labeled with t or stays where it is if t is part of the path from the root to the state pointer.

In Figure 6, examples of conditioning and instrumentation are shown for our PHP-based on-line cart system. The upper frame shows how displaying the widgets for performing the task *Add Comment* is conditional to *canBePerformed(addComment)* being true. Once the user presses the submit button, a different file (commentControl.php) arranges to insert the comment to the database and, among other workings, a call to *perform(addComment)* is made, so that the policy tree advances to the corresponding node. In the lower frame, how customization conditions are mixed with run-time conditions is illustrated. Thus, the “Checkout” button is visible if “Checkout” is allowed by the current customization policy and the cart is non-empty, which is something irrelevant of policy tree – below we discuss more on the extent to which the policy tree can be used to influence the details of the actual control flow.

Use of the policy tree is not restricted to the functions discussed above. For example, as we will see below, the function *hasBeenPerformed(t)*, which returns true iff task *t* is part of the associated



```
viewDetailedProductInfo.php:
...
<?php   if($tree->canBePerformed("addComment"))              { ?>
   <p> <font size='5' color='blue'> Writting comment </font></p>
   <form method='POST' action='commentControl.php'>
   <b> Title </b><br>  <input type='text' name='title'><br><br>
    <b>Body</b><br>  <textarea rows="10" cols="30" name='body'></textarea>
    ...
    <input type='submit' name='submitComment' value='Submit'>
    <input type='reset' name='reset' value='Clear'>
    ...
<?php } ?>
...
```

```
commentControl.php:
...
$title = $_POST['title'];
$body = $_POST['body'];
$inventoryID = $_POST['inventoryID'];
$userID =$_POST['userID'];
$commentControl = new commentControl();
$commentControl->setComment($title, $body, $
$inventoryID);
$tree->perform("addComment");
header("Location:http://
".$_SERVER['HTTP_HOST'].$clientRoot."productControl/
viewDetailProductInfo.php?inventoryID=$inventoryID");
...
```

```
viewCart.php:
...
<?php
   if( $tree->canBePerformed("reviewOrder")          )
   && ($CartControl->getNumOfItem()>0){?>
       <form action="CartControl.php"
method="POST"><p>
       <input type='submit' value='Checkout'
name='continue'/></p>
   <?php } ?>
...
```

Color	Black	
Option	16G	
Cost	$299	Qty:

Writting comment

Title

Body

Copyright © York University. All Rights Reserved.

	BlackBerry Storm	BlackBerry	16G		Black	$299	2
Picture	Product name	vender	Option	Color	Cost	Qty	Up
	Inspiron	Dell	none	black	$2000	2	

Total number of items: 4
Total Cost before tax: $4598
Tax : $597.74
Total Cost after tax: $5195.74

Copyright © York University. All Rights Reserve

Figure 6: Conditioning and Instrumenting Code

prefix of the node currently pointed, proved to be helpful in handling large numbers of task permutation possibilities.

5.3 In Action

Let us review how the techniques described so far are used in practice. At the design stage the software system is developed in sync with a goal model. Development is performed in a way that the code that relates to tasks is activated only if and when this is indicated by the policy tree. The latter is defined based on customization formulae given by the users at run-time. This is achieved by following the analysis process of Figure 3 and by subsequently processing the bulk of resulting plans to construct the policy tree. Thus, the system exhibits behavior that is compliant to these plans by loading and using the resulting policy tree. This way, different customization formulae yield different policy trees, which are, in turn, plugged into the system to alter its behavior accordingly.

Back to our on-line shop example, consider the scenario in which the shop-owner wants to construct CFs for newly identified groups within her customer base. In Figure 7, two different CF scenarios she devised can be seen together with screen-shots showing the effect they have to system behavior. On the scenario on the left the CF prevents the users from – among other things – viewing any product info before they login. In effect this means that once the session starts the only user action that is allowed is logging in. Indeed, in the policy tree, login is the only child of the root. This explains the bare-bones screen that is offered to the users (upper left screen-shot labeled [A]). Further, in the same scenario on the left of the figure, the CF prevents the user to add any comments. Hence, this facility is absent when viewing detailed product information (the bottom left screen-shot [B]). At that stage, however, making use of the cart or logging out is possible as seen in the policy tree. Thus, the button “Add to cart” is visible next to the product and the button “Logout” on the top left of the screen. The scenario on the right side of Figure 7, on the other hand, tailored to e.g. customers from a particular country overseas, prevents use of the cart but does not prevent addition of comments. Thus, at a stage where detailed product information is viewed, the user cannot add the item to the cart as before, but she can post a comment or log-out (screen-shot [C]). This is exactly what the state pointer indicates.

6. APPLYING GOAL-BASED CUSTOMIZATION

Let us now turn our focus to feasibility evidence and other findings that our case study with the on-line cart system offered to us. Our focus in the application revolved around: a) application of the two basic implementation principles laid in Section 4 on how to connect tasks of the goal model with source code, b) how feasible was use of the policy tree and its functions for making code level customization decisions and c) how sensible the proposed customization practice is once system development has been completed. We discuss these aspects below.

6.1 The Product and the Development Process

The on-line cart system we built is a 5KLOC application in PHP. The system follows a common 3-layer architectural style. The separation between layers happens at the level of PHP files, which are the basic modularization elements in PHP together with PHP classes. The view layer contains PHP files that generate HTML/JavaScript to be rendered at the client’s browser. The intermediate application logic layer contains control and entity classes with their member methods or free-standing PHP functions. These classes use, in turn, classes of the storage layer, located in separate files, which handle connection with the DBMS (MySQL in our case).

To develop the system an actor played the role of the analyst (the first author) and two other actors (including the third author) played the role of the developers. The analyst drew and maintained the goal model and the developers, who were both new to goal modeling at that time, developed the system using the goal model as the exclusive tangible representation of the functional requirements. The approach that the developers followed was to treat each of the relevant tasks in the goal model as a high-level description of an acceptance test that the end-system needed to pass. They were also requested to ensure that optional and alternative tasks maintain that status in the implementation. No other constraints or rules that specifically relate to goal-models were enforced other than compliance to established object-oriented 3-layer design principles.

The goal behind following this development process was to understand whether the desired connection with the tasks of the goal model, understood through the mapping principles of Section 4 would emerge naturally in the development of the system, without the need to impose other architectural or process restrictions or adding burden to the development effort. Our result was positive as we explain below.

6.2 Applying the Mapping Principles

Examining the artifact after the development process, for each mixed-agent task in the goal-model we found that there was a set of statements whose exclusive purpose was to support the performance of the task. This is what we defined as mapped code above. Mapped code fragments were not necessarily together in the source, but of-

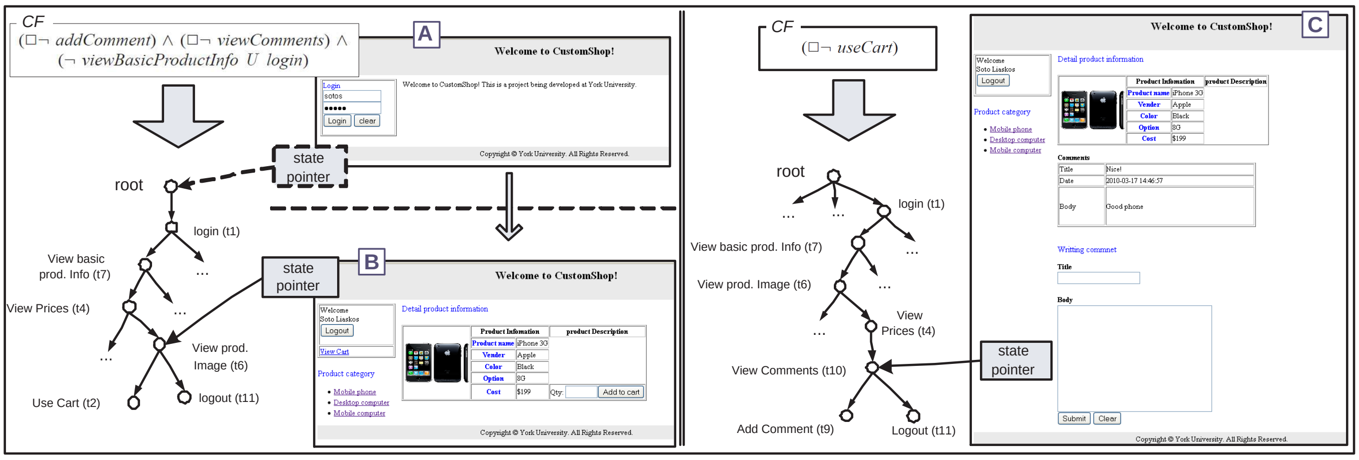


Figure 7: The effect of Customization Formulae

ten scattered across files or objects. Furthermore, the mapped code fragments of tasks were most likely found in the view layer. Back in Figure 6, the task *Add Comment* is mapped to a code fragment in the view layer that renders the HTML code for constructing the appropriate text-boxes and buttons. Code that supports this task may exist at the back layers as well, such as, for example, the data access object for inserting a newly submitted comment in the database – this code however may not be exclusive to that task.

What is important is that, in all cases, by conditioning (i.e. removing) the mapped code on the view layer the developers were able – without problems – to enable (or disable) support for the corresponding task, through availing the users (or not, respectively) to the interface widgets that allow execution of the task. Thus, in the above example, conditioning the PHP/ HTML code that draws the text boxes for adding comments can guarantee that the user shall not have the means to perform this task. Moreover, in our application we did not find such conditioning to have significant undesired ramifications such as destabilizing or complicating development of other parts of the code.

In addition, it was not difficult for the developers to define the places in the source code where completion of each task execution was signified. In tasks such as *View Product Image*, that place was right after the set of statements for displaying the product image to the screen. In task *Use Cart*, this point was the completion of a function of the cart at the application logic layer, such as adding a product to it. This particular example illustrates also that higher-level tasks may need multiple instrumentation.

Hence, the two principles of task separation and instrumentation appeared to be applicable in our case. Moreover, task separation did not require special consideration and effort but rather emerged naturally in the result.

6.3 Using the Policy Tree

At a second stage of the development effort the developers went on to actually enable use of the policy tree for enforcing customization decisions implied by it. To achieve this an extra module was introduced in the design that maintained a model of the policy tree and the position of the state pointer, while exporting the *canBePerformed(t)* and *performed(t)* functions for querying and advancing the pointer. These functions were used to condition mapped code or fill instrumentation points, as discussed earlier.

We found that these activities were possible and did not lead to major revisions or unsolvable issues. We did however have some occurrences of the predicate *canBePerformed(t)* being insufficient to elegantly arrange a customization decision. Those were cases in which more knowledge of the plan prefix that had lead to a given execution point was also helpful. The solution is to introduce additional predicates to query the policy tree whenever this is necessary. In our application, the predicate *hasBeenPerformed(t)*, which returns true if the task t is part of the current plan prefix – i.e. part of the path from the root to the state pointer in the policy tree – proved a sufficient addition.

It is important to note that the policy tree is meant to be a a tool for facilitating communication and enforcement of customization decisions taken at a requirements-level, and not for controlling the overall behavior of the software system. For example, the policy tree accounts neither for repetitions and loops nor for varying run-time parameters of various objects. Thus, the tree is unaware of and unaffected by possibilities such as the user providing wrong log-in credentials or going back and forth correcting address and credit card info in the corresponding screens of the check-out process. In this way, developers can follow their own approach for designing and implementing the control flow of their application, while using the policy tree as a tool for only posing customization constraints.

6.4 Dealing with Permutations

The advantage of using the policy tree for enforcing customization decisions was best exhibited in problems where multiple permutations of steps are possible to complete a process within the same or different customization scenarios. Considering the check-out process of our on-line shop, for example, credit card and address info can be acquired in any order (t_{15} and t_{13} or t_{14} respectively), using two different screens. Although the two tasks can of course be developed independently, problems arise when each step needs to redirect to the next step, or provide to the user the history of steps so far or the steps that can be performed from there. One solution is to require the developer of each screen to be aware of global customization decisions. The space of such possible decisions, however, quickly explodes as the number of steps and therefore permutations increases, making exhaustive encoding of all possibilities, for the purpose of communicating customization, impractical.

To avoid this, we can acquire decisions or content pertaining to global customization from a third “control” module. In our application, this role was supported by the policy tree. Thus, arrays of *canBePerformed(t_i)* and *hasBeenPerformed(t_i)* checks where used to, for example, appropriately label the “Next” buttons of the check-out “wizard” of the on-line store and ensure that they redirect to the correct next step according the preferred plan.

The benefit of using the policy tree for supporting customization control is both that the customization logic itself is, as we saw, derived from a user-centered requirements analysis process and that the heavy computational work that this derivation normally implies is performed off-line. This contrasts our proposal with two possible extremes of either hard-coding customization options in an ad-hoc manner (which seems to be the current state of practice - [18]) or deferring the overall control of the application to a flexible yet computationally expensive on-line reasoning procedure.

6.5 Anchoring the Policy Control Process

An important consideration when applying the technique is scoping behaviors. In our example, a plan prefix reflects the use of the system by one user at a particular time. The same or a different behavior may unfold from the beginning in a different client system (some other customer trying to buy something), or by the same customer later that day. Further, a plan prefix may never develop to a complete plan if the user chooses to abandon use of the system.

With the term *anchor* we refer to any type of entity, or group thereof, whose lifetime is bound to a plan prefix. In our example, the anchoring entity is the simple web session. If, for example, the session expires so does the plan prefix that has been constructed to that point. A new session always means an empty plan prefix (i.e. state pointer points to the root of the policy tree) waiting to be expanded through user actions. In different applications different anchoring entities can be thought. In an application processing business process, e.g. for academic admissions, a student application can be considered as the anchoring entity. Thus, for each new application that arrives a new empty prefix is constructed which is then augmented based on tasks that are performed to process that particular application. In the policy tree, this prefix augmentation is represented as progression of the state pointer towards the leafs.

Interestingly, different anchoring entities can be treated by different policy trees. For example different users of our on-line store (identified through e.g. a cookie mechanism) may experience different behavioral customizations, through assigning a separate policy tree to each of them.

6.6 Using Customization Formulae

Once the prototypical on-line store was up and running we tried a variety of customization formulae and observed their effect in constraining the system behavior via construction of the corresponding policy tree. Our main experimentation revolved around multiple policies as to when login should be performed with respect to other tasks, as well as what the allowed sequence of the check-out screens should be. These were customized though customization formulae of precedence (translated through the *U* operator in LTL). In combination to these constraints we also added existential ones dictating: whether comments can be added or viewed, whether the image should be displayed, whether the cart could be used or even whether login was possible. All these constraints were chosen based on what we thought could be realistic needs of a shop owner.

The formulae would successfully translate into a system that behaved accordingly. We strongly believe that the perceived complexity of LTL does not obstruct the process in any way, both because very simple LTL formulae suffice for achieving interesting customization results and because the use of templates is always possible for completely hiding the LTL details.

We, however, found that our framework suggests a customization practice we have not been accustomed to. The users, instead of choosing from a set of predefined customization options, which reflects today’s practice (cf. [18]), are instead asked to construct and “run” their own customization desires. While this adds significant flexibility and allows for defining customizations that are otherwise impossible in the current paradigm (e.g. arranging complicated permutations), it also implies that extra steps need to be taken for the users to understand and validate the customization constraints they pose, before these are enacted in the system. Our work on preference-based exploration of requirements alternatives [20] may offer a way by which this understanding can be facilitated. We however believe that more experimentation with real users is required to fully understand the practice of preference-driven customization.

6.7 Performance and Tool Considerations

The construction of a policy tree is an off-line activity and can afford longer computation times on separate computing infrastructure. This practice is to be contrasted with an approach in which an AI planner or other reasoning machinery is used at run-time, demanding unpredictably expensive computational steps to intervene in the normal control flow. It is important to note that a working customization can be achieved even if a subset of all admissible plans is provided, though the resulting policy may prevent behaviors that are otherwise desired. The policy tree can keep being updated as the planner returns new plans.

To acquire a sense of the time required for generating a useful set of plans with the current planner implementation, we tried different examples of CFs over the goal model of Figure 1 while varying the maximum amount of plans. The result can be seen in Table 8. Rows represent different CF scenarios. Thus CFs *browse* and *browse2* constraint use of the cart and the check-out system – the latter constraining, among other things, comments as well. Scenarios *loginFirst*, *checkPrices* and *useCartX* require user login before browsing, using the cart and checking out, respectively – the last come in different variations on aspects such as the ordering of check-out screens. For simplicity we omit the full LTL specification of those customization formulae. The cells show how long it took for an Intel Xeon QuadCore at 2GHz, 6KB Cache and approx. 780MB RAM reserved for the computation to calculate the first 20, 40 etc. admissible plans for each of those customization scenarios – times are in seconds. Note that the conversion of the preferred plans into a policy tree is computationally insignificant in comparison and thus not the actual concern in this discussion.

Scenario	**Top 20**	**40**	**60**	**80**	**100**
browse	2	4	10	40	65
browse2	3	23	43	60	130
loginFirst	28	148	420	1302	1777
checkPrices	21	67	165	381	684
useCart	25	108	206	509	859
useCart2	19	56	114	191	286

Figure 8: Time to Generate the first N Plans (in sec)

We definitely anticipate much better performance as the field of preference-based planning is fast progressing. For example, an HTN-based planner with preferences has been introduced which offers dramatically better performance through utilization of the domain knowledge expressed as task hierarchies ([26]). The principles applied in this paper are applicable to that planner as well. Further, to our knowledge, an efficient preference-based planner that readily returns a policy rather than a set of plans (lifting thereby the need to construct the policy tree as a separate step) is yet to be introduced in the AI planning community.

6.8 Reflection: Advantages and Challenges

Overall, our exploration with the on-line store strongly demonstrated three basic advantages of our proposal. Firstly, it makes software customization a requirements problem, whereby users customize the system by talking about their goals and activities rather than features of the software. Secondly, customization is constructive, meaning that users express their own desires as to how the system should behave, and not selective, where users would be restricted in a predefined set of choices, which limits the customization possibilities. Thirdly, the implementation is impacted to a minimal degree in a way that application of our approach can be possible independent of methodological, architectural and platform choices.

The aspect that we found challenging in our application was that of quality assurance. In our proposal the space of possible customizations dramatically increases, in a way that testing becomes a more challenging activity. We believe that this is a necessary consequence of any effort to produce high-variability adaptable designs. Our naive testing practice was based on selecting a set of characteristic customization formulae and devising test plans for each. We believe, however, that since the goal model itself is a descriptor of all customization possibilities, it can potentially be used for producing more educated quality assurance plans.

Finally, the case study per se allows for generalizability arguments that cannot exceed certain limits. Firstly, our implementation is a relatively small 5KLOC system and therefore the influence of scale to the proposed practice is yet to be empirically explored. It must be noted, however, that measures of size that seem more applicable to our case may as well be aspects such as the number and complexity of user interactions. In that regard, we consider our prototypical online cart system to be a good model of a real working system of this type.

Secondly, the software we developed is an example of a web-based system for supporting business processes. Applications in different classes of systems would perhaps offer more evidence on the breadth of applicability of the proposal. The same is, finally, true with the platform and architectural style that was chosen, and, particularly, with the user interface technology and architecture that was applied. Our follow-up empirical investigation involves different classes of systems that employ alternative and more complicated interactions with the users.

7. RELATED WORK

Our proposal for requirements-driven software customization relates to work on adaptive systems as well as product lines and product derivation.

General goal driven adaptation has been proposed by several authors. Thus, Zhang et al. [32] use temporal logic to specify adaptive program semantics. Further, work by Brown et al. [6] uses goal models to explicitly specify what should occur during adaptation. Their approach uses goal models to specify the adaptation process; in our approach the adaptation is the indirect result of imposing customization and precedence constraints on goals. Baresi and Pasquale [3] model adaptation strategies for service composition using goal models. While their method applies to service composition, our approach is more general and is not tight to a specific technology or application domain. Finally, strategy trees have been used to evaluate alternative reconfigurations of software systems in the context of QoS and structural changes [25]. Our approach differs in that it deals with user goals and behavior adaptation.

In product lines the dominant approach for representing variability are feature models [15, 9] which are appropriately instantiated [10, 31] for the derivation of individual members of the product family [24]. However, feature models as well as other variability models used in the area [2, 14], apart from being solution-centered rather than user-centered, they also aim at representing a great diversity of software characteristics and, therefore, they inevitably have relaxed semantics. This prevents them from being the optimal tools for reasoning exercises such as the one we use in this paper – e.g. temporal reasoning. We have indeed not found feature model instantiation proposals with characteristics similar to the ones we propose here. The extensive literature on software composition, on the other hand (e.g. [21] for a taxonomy), is either focusing on component-based or service-oriented architectures (e.g. [27]) or focuses on specific technologies, frameworks or techniques by which composition can be implemented – e.g. the AHEAD framework and its descendants [4, 1] or Aspect Orientation [16], Domain Specific Languages and Generators [8, 9]. Again, we could not find any proposals for approaching the requirements aspect of the problem, that is how the desired compositional result can be communicated through reference to terms related to the experience and the goals of the actual users, rather than technical details.

As far as the connection between goal models and implementation is concerned, this problem has been addressed from a variety of angles. For example, specific methodologies for generating agent oriented or component-based designs have been proposed in the context of two major goal-oriented frameworks, that is Tropos [7, 23] and, to some extent, KAOS [28]. Our contribution, however, is not a goal-driven or, in general, model-driven software development approach, but instead a proposal for using goal and preference models for software customization that has the minimum possible impact to the way the system is architected and developed.

8. CONCLUSIONS

Tailoring the behavior of a software system to the needs of individual stakeholders, contexts and situations as these change over time has emerged as an important need in today's software development. However, it also poses a challenging software engineering and maintenance problem.

The main contribution of our paper is a technique to exactly allow this translation of high-level customization requirements into an appropriately configured system, in a flexible and accessible way. Generic goal models are used to concisely represent a large number of alternative behaviors that actors may exhibit in order to fulfill their goals. The system is developed in a way that each such behavior can potentially be supported. After deployment, however, stakeholders can specify additional customization constraints, in the form of simple temporal formulae, restricting the space of possibilities to meet their current needs. Using an AI preference-based planner, a tree representing admissible behaviors is constructed and plugged in the application, for the latter to adapt its behavior to the new customization constraints.

The merits of our approach lie in the following features. Firstly, it offers a direct linkage of software customization with user requirements using goal models and high-level customization formulae. This way customization is performed through talking about the user activity and experience rather than features of the system to be. Secondly, our proposal for constructive customization, where users express their exact needs, versus selective, where users select from predefined options, allows for flexibly leveraging a much larger space of customization possibilities, leading to systems that are better tailored to the exact needs of users. Thirdly, the proposed approach implies minimum impact to the implementation process, being transparent to the architectural, modularization, process and platform choices the engineers have made, as long as two simple mapping principles are followed and the ability to maintain and query the policy tree is arranged. Our application in the on-line cart system offered us strong evidence that both the customization practice per se and the engineering and development intervention that enables it are feasible and exhibit the above advantages.

Our proposal opens a variety of possibilities for future research. One of them is work on improving the performance of the reasoning mechanism. Recent advances in preference-based planning indicate that there is significant room for such improvement. Most interesting would also be an empirical investigation of the use of the goal model and the planner as tools for allowing developers to better understand and implement the space of behavioral alternatives when following various development approaches. In the context of such an empirical effort, our basic implementation principles should also be tested for their applicability and generality. Furthermore, application of the technique in a variety of system types would allow better understanding of whether the current form of the policy tree offers the right level of information or whether it should be enriched into a complete behavioral model for the application – and how. While the latter has been deemed impractical and restricting in this paper, further empirical work can help us actually confirm that.

9. REFERENCES

[1] S. Apel, C. Kastner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 221 –231, 2009.

[2] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesch, and A. Vilbig. A Meta-model for Representing Variability in Product Family Development. In *Proceedings of the 5th International Workshop on Software Product-Family Engineering (PFE5)*, Siena, Italy, 2003.

[3] L. Baresi and L. Pasquale. Live goals for adaptive service compositions. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10)*, pages 114–123, 2010.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 187–197, Washington, DC, USA, 2003.

[5] M. Bienvenu, C. Fritz, and S. McIlraith. Planning with qualitative temporal preferences. In *Proc. of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, 2006.

[6] G. Brown, B. H. C. Cheng, H. Goldsby, and J. Zhang. Goal-oriented specification of adaptation requirements engineering in adaptive systems. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems (SEAMS '06)*, pages 23–29, New York, NY, USA, 2006. ACM.

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

[8] J. C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, 1988.

[9] K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *Proc. of the 3rd Software Product Line Conference (SPLC'04)*, pages 266–283, 2004.

[11] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 411–420, Los Alamitos, CA, USA, 1999.

[13] C. Gacek and M. Anastasopoules. Implementing product line variabilities. *SIGSOFT Softw. Eng. Notes*, 26(3):109–117, 2001.

[14] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Software and System Modeling*, 2(1):15–36, 2003.

[15] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *In Proceedings of the European Conference on Object-Oriented Programming ECOOP*, 1997.

[17] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007.

[18] 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 (RE'05)*, Paris, France, 2005.

[19] S. Liaskos, S. A. McIlraith, and J. Mylopoulos. Towards augmenting requirements models with preferences. In *Proc. of the 24th International Conference on Automated Software Engineering (ASE'09)*, pages 565–569, 2009.

[20] S. Liaskos, S. A. McIlraith, and J. Mylopoulos. Integrating preferences into goal models for requirements engineering. In *Proceedings of the 10th International Requirements Engineering Conference (RE'10)*, Sydney, Australia, 2010.

[21] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56 – 64, july 2004.

[22] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In

Proceedings of the 20th International Conference on Software Engineering (ICSE'98), pages 177–186, Washington, DC, USA, 1998.

[23] L. Penserini, A. Perini, A. Susi, and J. Mylopoulos. High variability design for software agents: Extending Tropos. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2(4), 2007.

[24] R. Rabiser, P. Grunbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *Proceedings of the 11th International Software Product Line Conference (SPLC '07)*, pages 141–150, Washington, DC, USA, 2007.

[25] B. Simmons. *Strategy-trees: A Novel Approach to Policy-Based Management*. PhD thesis, University of Western Ontario, February 2010.

[26] S. Sohrabi, J. A. Baier, and S. McIlraith. HTN planning with preferences. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 1790–1797, Pasadena, CA, USA, 2009.

[27] S. Sohrabi, N. Prokoshyna, and S. A. McIlraith. Web service composition via generic procedures and customizing user preferences. In *Proceedings of the 5th International Semantic Web Conference (ISWC06)*, pages 597–611, Athens, GA, USA, November 2006.

[28] A. van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, pages 25–43. 2003.

[29] E. S. K. Yu and J. Mylopoulos. Understanding "why" in software process modelling, analysis, and design. In *Proceedings of the Sixteenth International Conference on Software Engineering (ICSE'94)*, pages 159–168, 1994.

[30] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. S. do Prado Leite. From goals to high-variability software design. In *Proc. of the 17th International Symposium on Foundations of Intelligent Systems*, 2008.

[31] H. Zhang, S. Jarzabek, and B. Yang. Quality prediction and assessment for product lines. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE'03)*, pages 681–695, 2003.

[32] J. Zhang and B. H. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software (Special Issue on Architecting Dependable Systems)*, 79(10):1361 – 1369, 2006.