Goal-Based Behavioral Customization of Information Systems

Sotirios Liaskos[1], Marin Litoiu[1], Marina Daoud Jungblut[1], and John Mylopoulos[2]

[1] School of Information Technology, York University, Toronto, Canada
{liaskos,mlitoiu,djmarina}@yorku.ca
[2] Department of Information Engineering and Computer Science, University of Trento, Italy
jm@disi.unitn.it

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 diverse needs and experience. We present a requirements-driven approach to behavioral customization of software systems. 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, Software Customization, Adaptive Systems.

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 the environment or in user requirements, adaptive systems are able to change their structure and behavior so that they fit to the new conditions [1,2]. 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 should 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 problem in information systems engineering. 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 extend our earlier work on goal variability analysis [3,4] and 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 through performance of various tasks. The system-to-be is developed and instrumented in a way that the chunks of code that can enable or prevent performance of such user tasks are clearly located and controlled in the source code. After completion and deployment of the application, to address their specific needs and circumstances, individual stakeholders can refine the goal 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. Thanks to having appropriately instrumented the source code, that tree structure can be used as a plug-in which is inserted in the system and enforces 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.

The paper is organized as follows. In Section 2 we present the core goal modeling language and the temporal extension that we are using for representing behavioral alternatives. In Section 3 we show how we connect the goal model with the source code, how we express goal-level customization desires and how we translate them into behaviors of the system. We discuss the feasibility of our approach in Section 4. Finally, in Section 5 we discuss related work and conclude in Section 6.

2 Goal Models

Goal models [5,6] 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

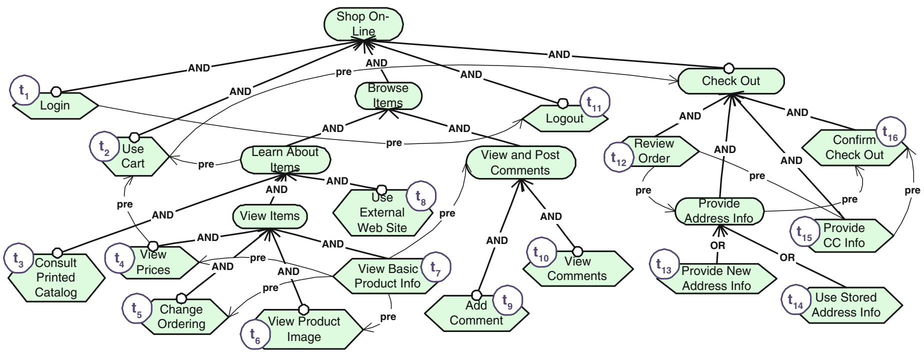


Fig. 1. A goal model

model shows alternative ways by which an on-line store can be used for browsing and purchasing products.

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 [6]. Tasks – the hexagonal elements – describe particular low-level activity that the actors perform in order to fulfill their goals. To ease our presentation, next to each task shape a circular annotation containing a literal of the form t_i has been added, which we will use in the rest of the paper to concisely refer to the task. For example, t_7 refers to the task *View Basic Product Info*.

Tasks can be classified into two different categories depending on what the system involvement is during 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.

Goals and tasks are connected with each other using AND- and OR-decomposition links, meaning, respectively, that all (resp. one) of the subgoals of the decomposition need(s) to be satisfied 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 sub-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.

Furthermore, 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. 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.

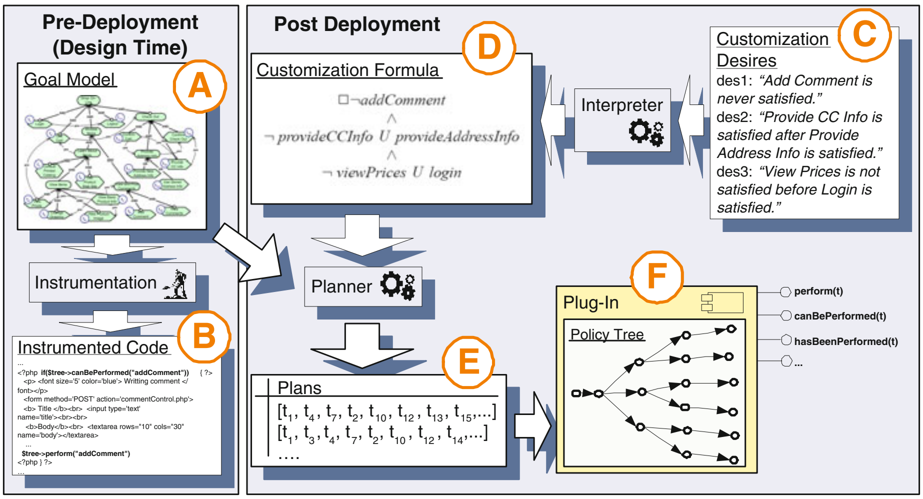
The goal model implies a potentially very large variety of such plans, which are understood as a representation of the variability of *behaviors* that an actor may exhibit in order to achieve their goals. Note that this behavioral variability is to be contrasted with variability of the actual software system, in that the same system variant may be used in a variety of ways by the user. For example, the user of our on-line store may variably choose to use or not to use the *Add Comment* feature, even if that feature is invariably available to them.

3 Enabling Goal-Driven Customization

Let us now see how our framework allows specification of preferred user behaviors and enables subsequent customization of the software system in a way that these preferred behaviors are actually enforced. A schematic of our overall approach can be seen in Figure 2. At design time the system is developed in a way that the code that enables each leaf level task is clearly identified in the source code (frame B in the Figure) and can be disabled or enabled using information appropriately acquired from replaceable customization plug-ins, whose construction takes place after deployment, as described below. After deployment of the application, the users can define behavioral customization constraints at a high-level using structured English (frame C). These constraints are translated into formulae in Linear Temporal Logic (D), which, together with the goal model (A) are provided to a preference-based planner. The latter produces plans of the goal model that best satisfy the given behavioral constraints (E). These plans are finally merged into a structure called *policy tree* (F) which is then plugged into the application so that the latter, thanks to the instrumentation that took place at design time (B), exhibits the behavior that is desired in the original customization constraints. In the rest of this section we describe each of these steps in more detail.

3.1 Connecting Goal Models with Code

To allow interpretation of preferred plans into preferred software customizations, the system is developed in a way such that elements of the source code are associated with tasks of the goal model. In our framework, the nature of this association as well as



```
...
<?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>
    ...
  $tree->perform("addComment")
<?php } ?>
...
```

$$\square \neg addComment \wedge \neg provideCCInfo \; U \; provideAddressInfo \wedge \neg viewPrices \; U \; login$$

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."*

$[t_1, t_4, t_7, t_2, t_{10}, t_{12}, t_{13}, t_{15}, \ldots]$
$[t_1, t_3, t_4, t_7, t_2, t_{10}, t_{12}, t_{14}, \ldots]$
....

Fig. 2. From Customization Desires to Policy Trees

the way it is established is transparent from a particular implementation technology or architectural approach (e.g. agent-, service- or component-orientation) or particular development process that, for example, goal-oriented development methodologies propose (e.g. [7]). It is also independent of variability implementation and composition techniques (e.g. [8,9,10]) in a sense that any such technique could potentially be chosen and applied. Thus, to establish the association between goal models and code we only identify two general principles, which, if applied during development – in whatever architectural or process context – 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 architectural, composition and variability implementation scenarios. In a component-based or service-oriented setting, for example, the mapped code of each task can be associated with existing interfaces or services – or adapters thereof – which may or may not be used by the process engine or other orchestration/composition environment. 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 show how fulfilling the above principles turned out to be a very natural process.

3.2 Adding Customization Constraints

The temporally extended goal model with its precedence links is intended to be an unconstrained and behaviorally rich model of the domain at hand. Indeed, the goal model of Figure 1 describes a large variety of ways by which the user could go about fulfilling the root goal, as long as each of these ways is 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 - frame D in Figure 2). 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 requirements. To construct CFs we use 0-argument predicates such as *useCart* or *browseItems* to denote satisfaction of tasks and goals. These predicates become (and stay) true once the task or goal they represent is respectively performed or satisfied. 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, in a use scenario, 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 for the first time (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 3. 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.

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

Fig. 3. 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 by Dwyer et al. [11], can be used to facilitate construction of CFs without reference to temporal operators. In our application, we used patterns in the form of 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 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. Examples of customization desire expressions can be seen in frame C of Figure 2. 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.

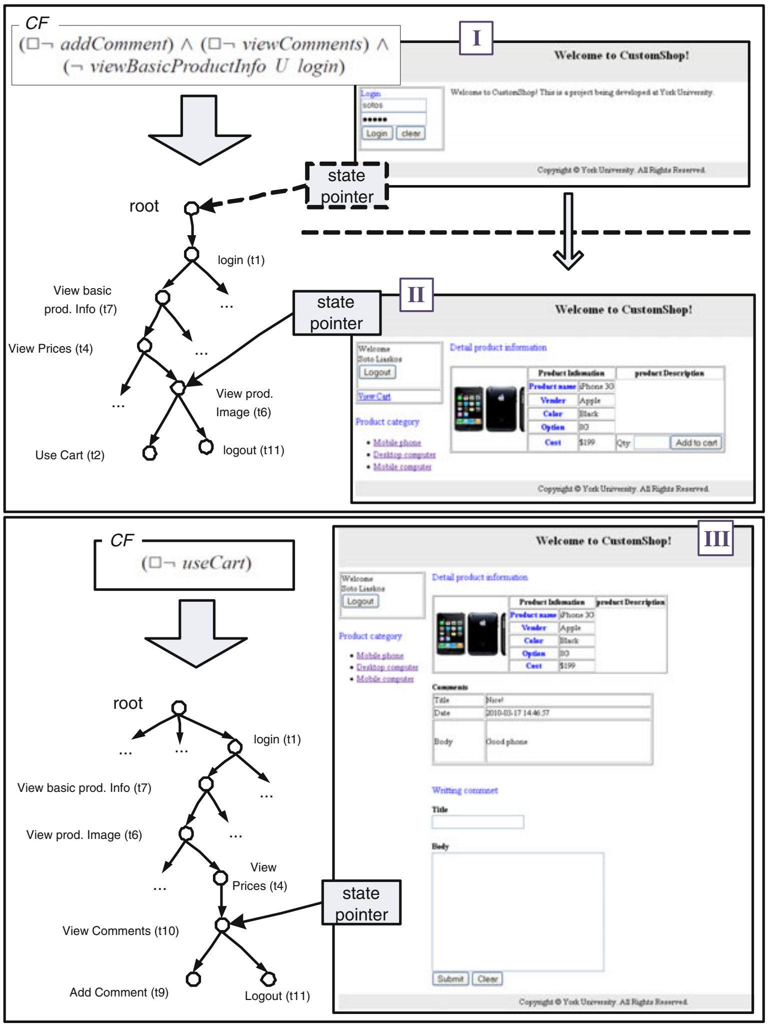
3.3 Identifying Admissible Plans

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 3. 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 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 [12]. The planner is given as input a goal model, automatically translated to a planning problem specification as well as a CF and returns the set of all admissible plans for the CF (frame E in Figure 2). 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 [3].

3.4 Constructing and Using the Policy Tree

We saw that the introduction of a CF dramatically decreases the number of plans that are implied by the goal model into a smaller set of admissible ones that also satisfy



Welcome to CustomShop!

Welcome to CustomShop! This is a project being developed at York University.

Copyright © York University. All Rights Reserved.

Welcome Soto Liaskos

Detail product information

Product Information		product Description
Product name	iPhone 3G	
Vender	Apple	
Color	Black	
Option	3G	
Cost	$199	Qty

Comments

Title	Nice!
Date	2010-03-17 14:46:57
Body	Good phone

Writing comment

Title

Body

Fig. 4. The effect of Customization Formulae

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. In particular, each node of the policy tree represents a task in the goal model. Given a set of plans P – where human-agent tasks have been removed – the policy tree is constructed in a way that every sequence of nodes that constitutes a path from the root to a leaf node is a plan in P and vice versa. It follows that every intermediate node in the policy tree represents both a plan prefix – i.e. the first n tasks of a plan – that can be found in P (by looking at the path from the root) and a set of continuation possibilities that yield complete plans of P (by looking at possible paths towards the leafs).

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 tasks of the plan prefix associated to that node (the *associated prefix*) have 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.

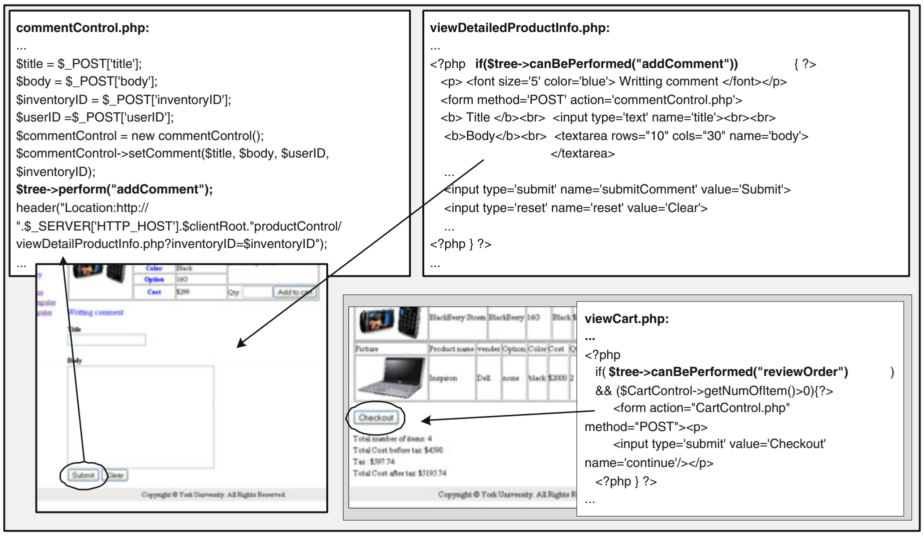
In Figure 4, for example, on the left side of the bottom frame, part of a policy tree can be seen together with the CF it originated from ($\Box\neg$*useCart*). Through use of the planner, that CF results in a set of admissible plans, say P. Some of those plans have a prefix $[t_1, t_7, t_6, t_4, t_{10}, \dots]$. Thus, in the resulting policy tree that is depicted, there is a path from the root to the node t_{10} that constructs this prefix. By looking at the children of node t_{10}, we infer that only two expansions of the prefix at hand will yield a longer prefix that also exists in P and therefore is admissible with respect to the CF: t_9 and t_{11}. In practice, this means that if we are to keep satisfying the CF, we should either perform one of those two actions or repeat actions of the existing prefix (but without moving the state pointer).

An algorithm for constructing a policy tree, from a list of admissible plans that the planner returns can be found in our technical report [13]. It is important to note here that a new plan can always be appended to an existing policy tree in linear time and enrich the behavioral possibilities. This allows us to use partial outputs of the planner immediately while gradually enriching the tree as new plans are generated.

3.5 Conditioning and Instrumenting the Source Code

Let us now see how the policy tree can be plugged into the software system to enable a behaviors that comply with the expressed customization desires. Preparation for this needs to actually happen at design time, when the application is developed. Recall that the system is built 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



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

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 } ?>
...
```

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 160
Cost $299 Qty Add to cart
Writting comment
Title
Body
Submit Clear
Copyright © York University. All Rights Reserved.
BlackBerry Storm BlackBerry 160 Black
Picture Product name vender Option Color Cost
Inspiron Dell none black $2000 2
Checkout
Total number of items: 4
Total Cost before tax: $4598
Tax : $597.74
Total Cost after tax: $5195.74
Copyright © York University. All Rights R

Fig. 5. Conditioning and Instrumenting 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 5, examples of conditioning and instrumentation are shown for our PHP-based on-line cart system. The upper right 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 (seen in upper left frame), so that the policy tree advances to the corresponding node. In the lower right 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. It is important to notice, therefore, that the policy tree is not used to completely arrange the details of the control flow of the application but to only enforce more abstract customization decisions that have been made at the requirements level. Note also that use of the policy tree is not restricted to the functions discussed above. For example the function *hasBeenPerformed(t)*, which returns true iff task *t* is part of the associated prefix of the node currently pointed, proved in our application to be helpful in handling large numbers of task permutation possibilities.

Note, again, that the injection of conditioning and instrumentation code discussed above is taking place at design time and based on the goal model. It is therefore independent of the actual structure of the policy tree, which, once the system is up and running, varies based on the customization constraints that are in effect each time.

3.6 In Action

Let us now see a complete example of how a system is customized through expression of high-level customization desires. Back to our on-line shop, consider the scenario in which the shop-owner wants to construct CFs for newly identified groups within her customer base. In Figure 4, 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 top frame the CF prevents the users from – among other things – viewing any product information 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 screen-shot labeled [I]). Later in the same scenario of the top frame the user has logged in and is browsing products. However, the CF prevents the user from adding any comments. Hence, this facility is absent when viewing detailed product information (screen-shot [II]). Nevertheless, at that stage, 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 lower frame of Figure 4, 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 [III]). This is exactly what the state pointer indicates.

4 Applying Goal-Based Customization

Let us now discuss some of the experiences we acquired from our case study with our on-line cart system. A detailed account on this application can be found in [13].

Code Development and Instrumentation. The on-line cart system we built is a 5 thousand lines-of-code (5KLOC) application in PHP, following a common 3-layer architectural style – i.e. separating view, application logic and storage layers. Two developers, senior undergraduate students at that time, where asked to develop the system following a standard textbook object-oriented approach with the only goal model related restriction that the leaf level tasks of the goal model (which was maintained exclusively by the first author) would be treated as acceptance tests for the end-product and that optional and alternative tasks maintain that status in the implementation. Looking at the result afterwards we found that task separation not only was possible but emerged naturally in the development process. Interestingly, the mapped code would tend to appear at the view layer of the application. Furthermore, subsequent conditioning and instrumentation of the mapped code did not pose difficulties either. Policy trees, on the other hand, are plugged as separate globally visible PHP classes in the application. The use of the methods *canBePerformed(t)* and *performed(t)* to query/manipulate the tree did not pose any obvious perception problems or design issues requiring intense problem solving effort.

Anchoring the Policy Control Process. An issue that triggered further investigation is that of 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. 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 anchor is the 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 (through progression of the state pointer) based on tasks that are performed to process that particular application. 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.

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 way, we avoid unpredictably expensive computational steps to intervene in the normal control flow. In our experimentation with several CFs over the bookseller goal model we found that the first hundred of admissible plans can be calculated within a time period ranging between one and 30 minutes. It is important to note that a working customization can be achieved even if a subset (in our case some tens) 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. We definitely anticipate improved performance as the field of preference-based planning is fast progressing. For example, an HTN-based planner with preferences has recently been introduced which offers dramatically better performance through utilization of the domain knowledge expressed as task hierarchies ([14]). The principles applied in this paper are applicable to any preference-based planner that can generate sets of plans.

5 Related Work

Our proposal for requirements-driven software customization relates to research on a variety of topics including adaptive systems, product lines and software/service composition.

General goal-driven adaptation has been proposed by several authors. Thus, Zhang et al. [15] use temporal logic to specify adaptive program semantics. Further, work by Brown et al. [16] 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. Strategy trees have also been used to evaluate alternative reconfigurations of software systems in the context of QoS and structural changes [17]. Our approach differs in that it deals with user goals and behavior adaptation.

Researchers have also proposed different ways to model and bind variability in business processes. Lapouchnian et al. use goal models for analyzing alternative business process configurations [18]. Lu et al. propose the construction of flexible business process templates that lay the basic constraints that must be met [19]. Elsewhere [20,21] variability constructs are added to existing business process notations. In requirements engineering, a constraint language with temporal features has been proposed to analyze families of scenarios [22]. In general, such frameworks do not include an implementation approach, and when they do, this is restricted to specialized frameworks such as workflow engines [21] or e.g. BPEL-based service composition platforms [18].

The extensive literature on software composition, on the other hand (e.g. [9] for a taxonomy), is focusing on specific technologies, frameworks or techniques by which composition can be implemented – e.g. composition of services ([23]), the AHEAD framework and its descendants [24,25] or Aspect Orientation [26], Domain Specific Languages and Generators [27,28]. Use of existing AI planning applications to service composition, in particular, ([29,30] – cf. [10] for a survey), requires certain assumptions such as, for example, availability of cleanly defined services, limited degree of user intervention or the existence of some implementation and execution technique of the desired composition that also alleviates increased reasoning times. Our customization framework attempts to be more generally applicable, has a stronger focus on the implementation aspect without making platform or architectural assumptions and it also focuses on user interactions and therefore families of behaviours (system customizations) rather than single-purpose compositions. At the same time, it focuses on the requirements aspect of the problem, that is how the desired customization result can be communicated through reference to terms related to the experience and the goals of the actual users, rather than technical features of the system.

6 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 systems development. However, it also poses a challenging 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. 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 desire specifications. 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 an extended empirical investigation on the applicability and generality of our basic implementation principles. Such empirical work also includes evaluating with end-users the extent and manner by which they can construct customization desires of various levels of complexity. 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 adding more expressiveness should be attempted. This could include, for example, adaptation of the semantics of satisfaction predicates so that task repetition also becomes subject to CF compliance or addition of run-time instance-level information to the produced policy structure. Such extensions would potentially allow for finer grain customization, but at the significant expense of simplicity, of impact minimality to the design and of maintaining a modest computational cost.

References

1. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proceedings of the 20th International Conference on Software Engineering (ICSE 1998), Washington, DC, USA, pp. 177–186 (1998)
2. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering (FOSE 2007), Washington, DC, USA, pp. 259–268 (2007)
3. Liaskos, S., McIlraith, S.A., Mylopoulos, J.: Towards augmenting requirements models with preferences. In: Proceedings of the 24th International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand, pp. 565–569 (2009)
4. Liaskos, S., McIlraith, S.A., Mylopoulos, J.: Integrating preferences into goal models for requirements engineering. In: Proceedings of the 10th International Requirements Engineering Conference (RE 2010), Sydney, Australia, pp. 135–144 (2010)
5. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
6. Yu, E.S.K., Mylopoulos, J.: Understanding "why" in software process modelling, analysis, and design. In: Proceedings of the Sixteenth International Conference on Software Engineering (ICSE 1994), pp. 159–168 (1994)
7. Penserini, L., Perini, A., Susi, A., Mylopoulos, J.: High variability design for software agents: Extending Tropos. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 2(4) (2007)
8. Gacek, C., Anastasopoules, M.: Implementing product line variabilities. SIGSOFT Software Engineering Notes 26(3), 109–117 (2001)

9. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. IEEE Computer 37(7), 56–64 (2004)
10. Rao, J., Su, X.: A survey of automated web service composition methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering (ICSE 1999), Los Alamitos, CA, USA, pp. 411–420 (1999)
12. Bienvenu, M., Fritz, C., McIlraith, S.: Planning with qualitative temporal preferences. In: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), Lake District, UK, pp. 134–144 (2006)
13. Liaskos, S., Litoiu, M., Jungblut, M.D., Mylopoulos, J.: Goal-based Behavioral Customization of Information Systems. Technical Report CSE-2010-10, York University (2010)
14. Sohrabi, S., Baier, J.A., McIlraith, S.: HTN planning with preferences. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), Pasadena, CA, USA, pp. 1790–1797 (2009)
15. Zhang, J., Cheng, B.H.C.: Using temporal logic to specify adaptive program semantics. Journal of Systems and Software (Special Issue on Architecting Dependable Systems) 79(10), 1361–1369 (2006)
16. Brown, G., Cheng, B.H.C., Goldsby, H., Zhang, J.: 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 2006), pp. 23–29. ACM, New York (2006)
17. Simmons, B.: Strategy-trees: A Novel Approach to Policy-Based Management. PhD thesis, University of Western Ontario (February 2010)
18. Lapouchnian, A., Yu, Y., Mylopoulos, J.: Requirements-driven design and configuration management of business processes. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 246–261. Springer, Heidelberg (2007)
19. Lu, R., Sadiq, S., Governatori, G.: On managing business processes variants. Data and Knowledge Engineering 68(7), 642–664 (2009)
20. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M.H., La Rosa, M.: Configurable workflow models. International Journal of Cooperative Information Systems (IJCIS) 17(02), 177–221 (2008)
21. Sadiq, S.W., Orlowska, M.E., Sadiq, W.: Specification and validation of process constraints for flexible workflows. Information Systems 30(5), 349 (2005)
22. Sutcliffe, A.G., Maiden, N.A.M., Minocha, S., Manuel, D.: Supporting scenario-based requirements engineering. IEEE Transactions on Software Engineering 24(12), 1072–1088 (1998)
23. Baresi, L., Pasquale, L.: Live goals for adaptive service compositions. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 114–123 (2010)
24. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), Washington, DC, USA, pp. 187–197 (2003)
25. Apel, S., Kastner, C., Lengauer, C.: Featurehouse: Language-independent, automated software composition. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), pp. 221–231 (2009)
26. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-m., Irwin, J.: Aspect-oriented programming. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, p. 313. Springer, Heidelberg (1997)

27. Cleaveland, J.C.: Building application generators. IEEE Software 5(4), 25–33 (1988)
28. Czarnecki, K., Eisenecker, U.W.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
29. Sohrabi, S., Prokoshyna, N., McIlraith, S.A.: Web service composition via generic procedures and customizing user preferences. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 597–611. Springer, Heidelberg (2006)
30. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.S.: Automating DAML-S web services composition using SHOP2. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 195–210. Springer, Heidelberg (2003)