Behavioral adaptation of information systems through goal models

Sotirios Liaskos[a], Shakil M. Khan[b], Marin Litoiu[a], Marina Daoud Jungblut[b], Vyacheslav Rogozhkin[c], John Mylopoulos[d]

[a]*School of Information Technology, York University, Toronto, Canada, Emails:* {liaskos,mlitoiu}@yorku.ca
[b]*Department of Computer Science and Engineering, York University, Toronto, Canada, Emails:* skhan@cse.yorku.ca,djmarina@gmail.com
[c]*Chair of Business Information Systems and Electronic Government, University of Potsdam, Germany, Email:* rogozhki@uni-potsdam.de
[d]*Department of Information Engineering and Computer Science, University of Trento, Italy, Email:* 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 models 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 and an automated teller machine simulator, does not assume adoption of a particular development methodology, platform, or variability implementation technique and keeps the reasoning computation overhead from interfering with the 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 information systems that are easier to maintain and manage. 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 specific users, such as those with proof of in-store or telephone purchase, allowed to add comments. 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 as 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, 5, 6] 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 on 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 extends our earlier publication on the matter [7] in three major ways. Firstly, we offer more details on the reasoning infrastructure that supports generation of customization plug-ins and show more rigorously why it works. Secondly, we show how issues of scalability, repetition, and loops can be addressed through dividing the customization problem into sub-problems and using multiple nested tree structures. Thirdly, to our original on-line cart application study, we add another study on an Automated Teller Machine (ATM) simulation and report on our experiences.

We organize our presentation 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. In Section 4 we offer more formal details of the process. We discuss the feasibility of our approach in Section 5. Finally, in Section 6 we discuss related work and conclude in Section 7.

2. Goal Models

Goal models [8, 9] 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.

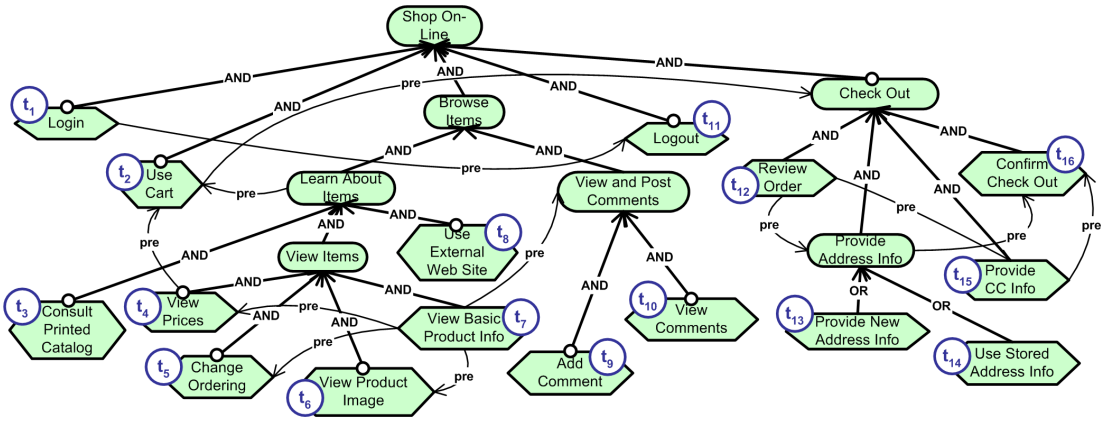

Figure 1: A goal model

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 [9]. 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 the 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 Product 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, only the task *View Basic Product Info* is 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 on satisfaction ordering we use the *precedence link* ($\xrightarrow{pre}$). The precedence link is drawn from a goal or task to another goal or task to specify that the 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 satisfy both the AND/OR-decomposition tree and the associated 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 captures 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 that is potentially exhibited by the user is to be contrasted with the 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 perform or not perform the task *Add Comment*, even if the system feature for performing that task (e.g. a textbox with a "submit" button next to it) 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 the way it is established is transparent from a particular implementation technology or architectural approach (e.g. agent-, service- or component-oriented) or a particular development process that, for example, goal-oriented development methodologies propose (e.g. [10]). It is also independent of variability implementation and composition techniques (e.g. [11, 12, 13]) in the 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 actuation* and *task instrumentation*, as explained below.

Task Actuation. 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

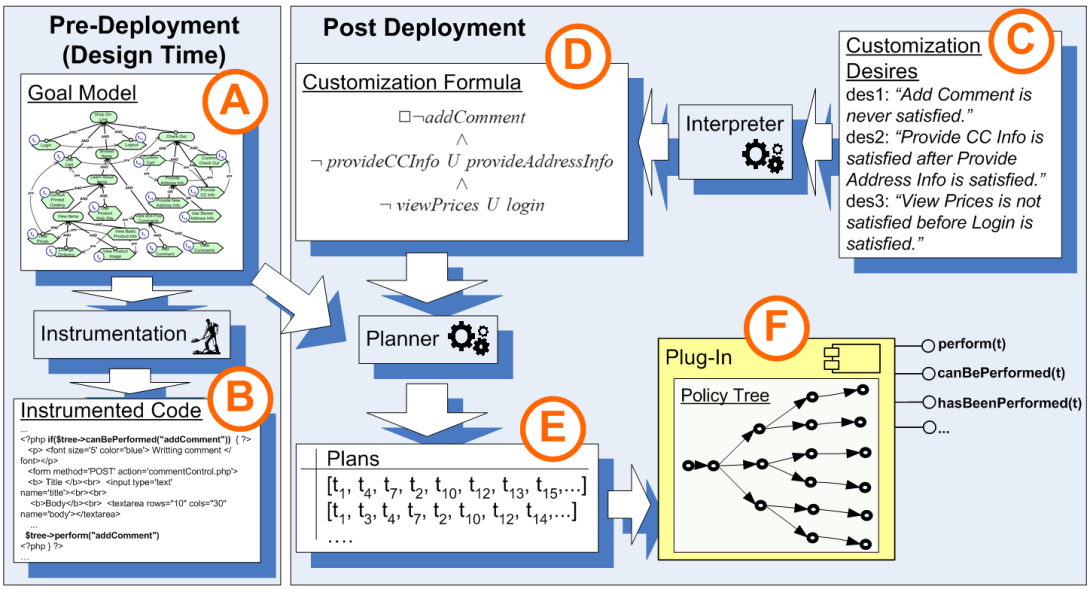

Figure 2: From Customization Desires to Policy Trees

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 the task *Login* is the code for drawing the username and password text boxes as well as the "Submit" and "Clear" buttons 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 studies with both the on-line cart system and the ATM simulator, 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 - [14]) 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:

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

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:

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

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

$$\Box\neg \textit{addComment}$$

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 CF of the figure the shop owner allows the users to only browse the products, their basic information, and their images.

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

Figure 3: A Customization Formula

While CFs, as LTL formulae, can in theory be of arbitrary complexity (e.g. can have nested temporal operators), we found in our experimentation that most CFs that are useful in practical applications are of a 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. [15], 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 of $[t_7]$, $[t_7, t_5]$,$[t_7, t_{10}, t_6]$, $[t_8, t_7, t_6, t_5]$, 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 of 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 [16]. 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. We present more details on how the planner is adapted in Section 4.

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 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 the 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 to the particular node) and a set of continuation possibilities that yield complete plans of P (by looking at possible paths from the node and 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 the sense that these tasks can be repeated.

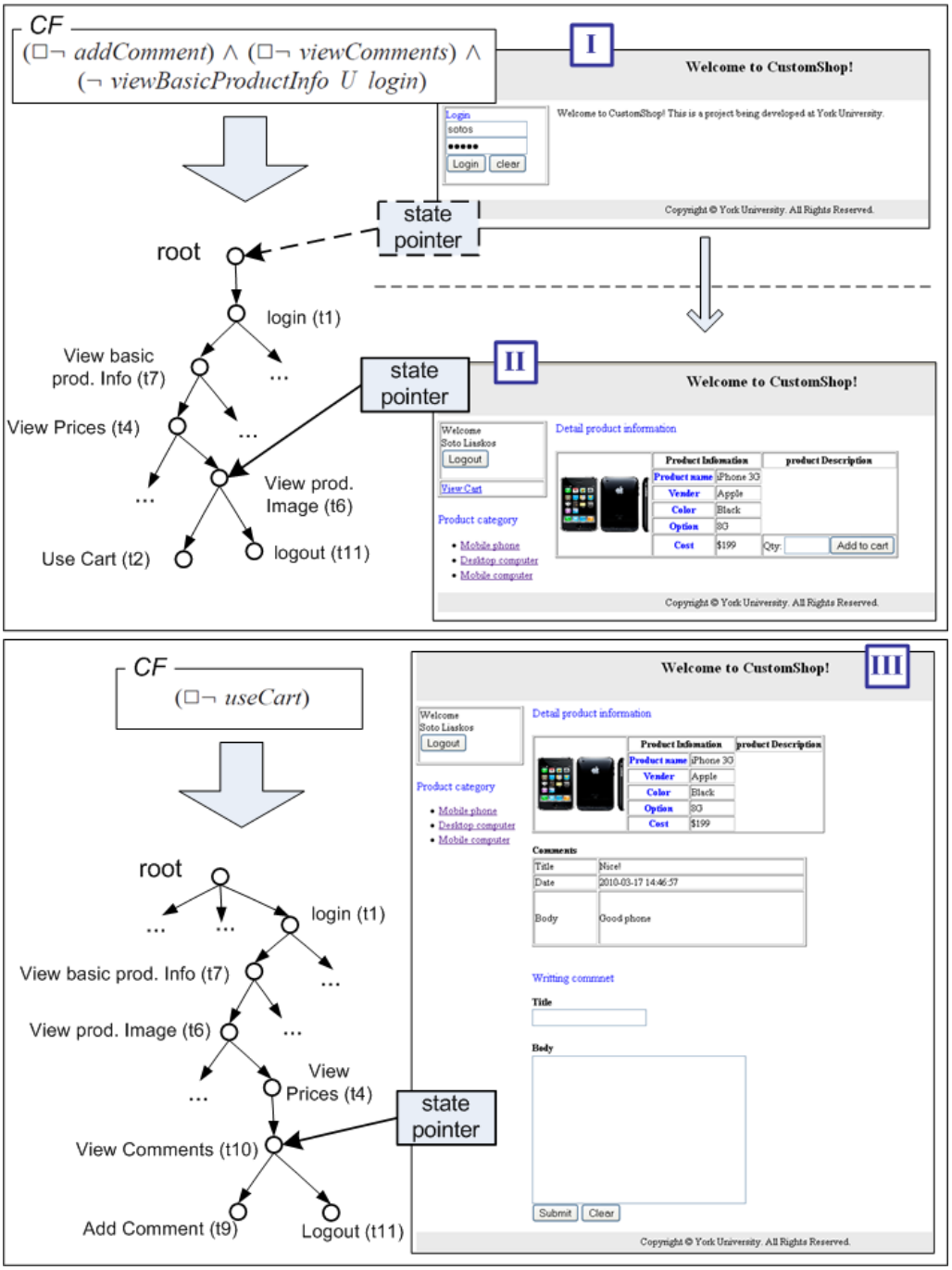

Figure 4: The effect of Customization Formulae

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 the 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 can 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 is presented later in the paper. 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 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 principles pertaining to task actuation 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 can be

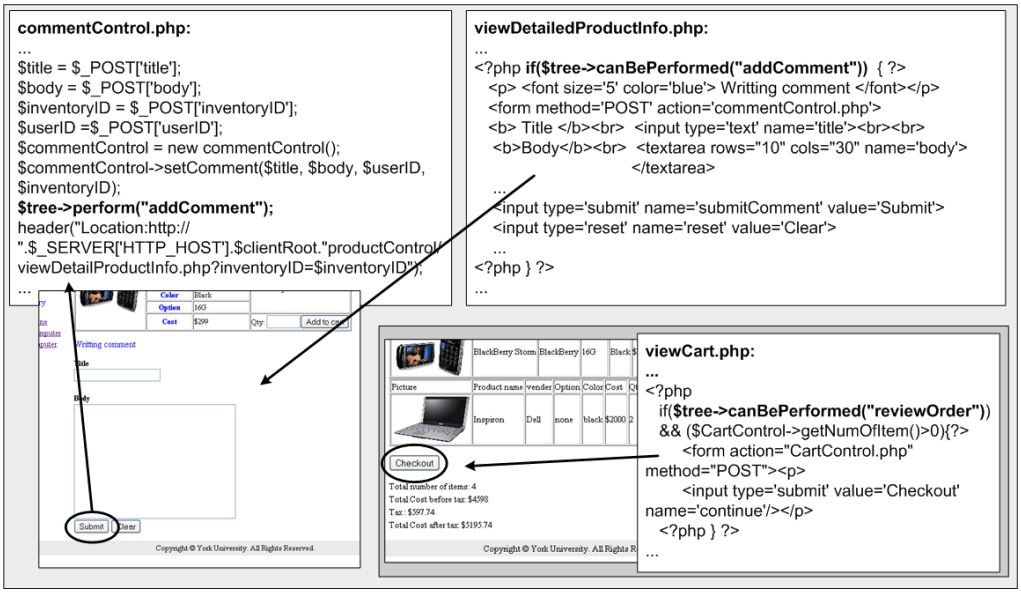

Figure 5: Conditioning and Instrumenting Code

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 cannot 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 condi-

tions 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 only to 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 is 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 on system behavior. In 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 labelled [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 this 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. Formalization and Translation Details

In this section, we look into the formal aspects of the previous steps in more detail. In particular, we show how the goal model and the customization formulae are together converted into a planning problem specification that allows reasoning about admissible plans – through the use of a preference-based planner. We then discuss in detail how the policy tree is constructed and show that the resulting application that uses the tree complies with the specified customization formulae.

We define the semantics of our goal and CF language within the situation calculus [17], a formal language for modeling dynamic domains. The situation calculus allows us to easily exploit existing algorithms and tools for preference-based planning for the purpose of evaluating admissible plans. Note that a similar formalization has been proposed elsewhere [6]; the formalization there however is based on a combination of hierarchical task networks (HTNs – [18]) and PDDL 2.0, a language for specifying planning problems with preferences [19]. Among other differences – which include the limited expressiveness of PDDL temporal preferences compared to LTL, the situation calculus does not offer constructs for defining hierarchies like HTN specifications do, and this requires special treatment as we see below. Through this treatment, the translation is applicable to a larger class of planning systems, e.g. those that do not necessarily allow hierarchical representations of domain information.

In the following we begin with a presentation of the situation calculus and continue with how the goal model and the CFs are translated into a dynamic domain in that language. The translated specification will then be fed to a preference-based planner to generate admissible plans. We then discuss how policy trees can be constructed from the set of admissible plans.

4.1. The Situation Calculus

The situation calculus is a logical language for specifying and reasoning about dynamical systems [17]. A *situation* s in the situation calculus is a *history* of the primitive actions, $\alpha \in \mathcal{A}$, performed from a distinguished initial situation S_0. The special function $do(\alpha, s)$ maps an action and a situation into a new situation, thus inducing a tree of situations rooted in S_0. In the situation calculus, the *state* of the world is expressed in terms of functions and relations, called *fluents*, relativized to a particular situation; for instance, the relational fluent $confirmedCheckout(u, s)$ might mean that the user u has confirmed checking out her cart in situation s. In the following, we will often refer to fluents in

situation-suppressed form, e.g. $confirmedCheckout(u)$ rather than $confirmed$-$Checkout(u, s)$. Another special predicate Poss(α,s) is used to denote that the action α is executable in situation s. Finally, $s \sqsubseteq s'$ means that either $s = s'$ or there is a sequence of actions $\vec{\alpha} = \alpha_1, \alpha_2, \ldots, \alpha_n$ such that $s' = do(\alpha_n, \ldots, do(\alpha_1, s))$[1].

A *basic action theory* $\mathcal{D}$ in the situation calculus comprises of a variety of axioms, the most important being for our purposes here: (1) *action precondition axioms*, one per action α characterizing Poss(α, s), (2) *successor state axioms*, one per fluent, that succinctly encode both effect and frame axioms and specify exactly when the fluent changes, and (3) axioms describing the *initial state* of the system. Other axioms include *unique names axioms for actions* that state that different action terms represent different actions, and domain-independent *foundational axioms* that pertain to the definition of situations per se. The details of $\mathcal{D}$ are described in [17]. Given a goal formula G, a *plan* in the situation calculus is a sequence of actions $\vec{\alpha} = \alpha_1, \alpha_2 \ldots, \alpha_n$ such that for the situation $s = do(\vec{\alpha}, S_0)$, G holds in s and the precondition axioms are satisfied throughout $\vec{\alpha}$, i.e. $\mathcal{D} \models G(s) \wedge$ Poss$(\alpha_1, S_0) \wedge$ Poss$(\alpha_2, S_1) \wedge \ldots \wedge$ Poss(α_n, S_{n-1}), where $S_1 = do(\alpha_1, S_0)$, $S_2 = do(\alpha_2, S_1)$, etc. In the section that follows, we show how to translate our goal model into a basic action theory, focussing on the three types of axioms we mentioned above.

4.2. Translating the Core Goal Model

We define the semantics of our visual goal language using a set of translation rules. Similar translation proposals are introduced in [20] and [21], but for different purposes; a distinguishing feature of our approach is the incorporation of CFs. We first establish a mapping from the primitives of the goal based graphical language to those of the situation calculus:

4.2.1. Primitives

- For every task t that appears in the goal model, introduce an action α_t and a relational fluent *performed(*t, s*)* in the situation calculus domain theory.

- For every goal g, introduce an AND/OR formula $\varphi_g(s)$ constructed from predicates of the type *performed(*t, s*)*. The formula, which we call *attainment formula*, is constructed as follows. Starting from g, each goal is recursively replaced by the conjunction or disjunction of its children, depending on whether g is AND or OR-decomposed – excluding optional subgoals. If

[1] We will use $do(\vec{\alpha}, s)$ as an abbreviation for $do(\alpha_n, \ldots, do(\alpha_1, s))$.

these subgoals are tasks, then the predicate *performed(*t, s*)* is used and the recursion terminates.

We can now specify the precondition, successor state, and initial situation axioms based on the following rules.

4.2.2. Action Precondition Axioms

- For every task t in the goal model, construct a precondition axiom as follows. First we will construct a formula φ_{comp}, which holds if t is unnecessary for satisfying the root goal in the goal model, as one or more alternative (competing) tasks to t has already been performed, making t redundant. Given the goal model, consider the path from t to the root goal. Let G_{OR} be the set of all nodes g_{OR} in the path which are OR-decomposed, including the root and t's parent. For each such g_{OR}, consider its children that do not belong to the path from t to the root goal. Let G_{comp} be the set of all such children of all $g_{OR} \in G_{OR}$. Finally let T_g be the set of all leaf level tasks that are successors of a goal g. If g is a task then T_g contains only that task. The formula φ_{comp} is constructed as follows:

$$\varphi_{comp} \equiv \bigvee_{\forall g \in G_{comp}} \left(\bigvee_{\forall t \in T_g} performed(t) \right)$$

Observe that t does not occur in any of the alternatives together with any of the tasks in $T_g, \forall g \in G_{comp}$. Excluding consideration of t together with any of these tasks ensures that the plans for the goal model are minimal with respect to the goal tree, or, in other words, no subset of the tasks included in the plan satisfies the root goal. Thus, φ_{comp} will be true if some of the competing tasks has already been performed making t redundant.

Then consider the set G_{pre} of all hard elements (hard-goals or tasks) h_i such that $h_i \xrightarrow{pre} g_j$, where g_j is t or any ancestor of t in the hard-goals subgraph. The precondition axiom for t is finally the following:

$$\text{Poss}(\alpha_t, s) \equiv \left(\bigwedge_{\forall h_i \in G_{pre}} \psi_{h_i}(s) \right) \wedge (\neg \varphi_{comp}(s))$$

In the above, ψ_{h_i} is simply a fluent of the form *performed(*h_i*)* if h_i is a task, or an attainment formula φ_{h_i} if h_i is a higher-level hard-goal.

Thus, the action α_t associated with some task t is executable in situation s if and only if the actions associated with all the tasks that precede t in the goal model have been performed in s, and t is not redundant in s.

4.2.3. Successor State Axioms

Successor state axioms tell us how exactly fluents (i.e. descriptors of the current state) change their values due to the performance of actions. We define one such axiom per fluent. Hence:

- For each of the fluents *performed(*t, s*)* introduced above, construct a successor state axiom as follows:

$$\text{Poss}(\alpha, s) \supset (\mathit{performed}(t, \mathit{do}(\alpha, s)) \equiv (\alpha = \alpha_t \vee \mathit{performed}(t, s)))$$

Thus, the fluent *performed(*t*)* holds in the situation resulting from action α being performed in situation s, if and only if α refers to the action associated with the task t, i.e. α_t, or *performed(*t*)* was already true prior to the action – provided that α is executable in s. Note that $\supset$ denotes the implication connective.

4.2.4. Initial Situation and Plans

For the initial situation S_0, every predicate of type *performed(*$\cdot, S_0$*)* is specified to be false. If $\mathcal{D}$ is the action theory derived from the goal model and φ_g the attainment formula representing the root goal g, then a plan for the goal model is a plan for $\mathcal{D}$ that achieves φ_g, given the initial situation S_0.

4.3. Customization Formulae

As we saw, Customization Formulae (CFs) describe temporal characteristics of the behavior that emerges while goals are being fulfilled in a particular order and under certain circumstances. Linear Temporal Logic (LTL) is used to form CFs. Thus, adapting [16] to our purposes:

Definition (Customization Formula - CF) A customization formula (CF) is an LTL formula formed with atoms from $H \cup T$, where H is the set of hard-goals and T the set of leaf level tasks of the goal model. It is drawn from the smallest set K for which:
1. $H \subset K, T \subset K$.
2. If ϕ, ϕ_1, ϕ_2 are in K, then so do $\neg\phi$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\circ\phi$, $\Box\phi$, $\Diamond\phi$, $\phi_1 U \phi_2$ and $final(\phi)$.

The symbols $\Box$,$\Diamond$, $\circ$ and U, represent the temporal operators *always, eventually, next* and *until*, respectively. These operators are self-explanatory – we evaluate them over finite paths (i.e. sequences of situations) specified by a pair of situations $[s_1, s_2]$, where s_1 is a starting situation and s_2 is the ending situation. In addition to the above standard operators, the operator $final(\phi)$ is also used, and holds if ϕ holds in s_2 (see below for a formal semantics). Given a CF and a plan for the root goal (therefore: a plan in the corresponding action theory), whether the plan satisfies the CF can be evaluated by appealing to the situation calculus-based semantics of LTL given by Gabaldon [22]. More specifically, let us use the notation $\varphi[s, s']$ to denote that φ holds over the sequence of situations from s to $s' = do(\vec{\alpha}, s)$. The semantics of CFs in situation calculus terms are as follows – again, appropriately adapted from [16][2].

$$g \in H \text{ then } g[s, s'] \equiv \varphi_g(s)$$
$$t \in T \text{ then } t[s, s'] \equiv performed(t, s)$$
$$f \in H \cup T \text{ then } final(f)[s, s'] \equiv f[s', s']$$
$$\Diamond\phi[s, s'] \equiv (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s')\varphi[s_1, s']$$
$$\Box\phi[s, s'] \equiv (\forall s_1 : s \sqsubseteq s_1 \sqsubseteq s')\varphi[s_1, s']$$
$$\circ\phi[s, s'] \equiv (\exists \alpha : do(\alpha, s) \sqsubseteq s')\varphi[do(\alpha, s), s']$$
$$\phi_1 U \phi_2[s, s'] \equiv (\exists s_1 : s \sqsubseteq s_1 \sqsubseteq s')\varphi_2[s_1, s'] \wedge (\forall s_2 : s \sqsubseteq s_2 \sqsubseteq s_1)\varphi_1[s_2, s_1]$$

On top of the Customization Formulae, which can in theory become complex, simpler languages for expressing common temporal properties can be defined. Customization Desires which we saw earlier can be expressions in such a language. We particularly used the system presented in [6], which, inspired by work on LTL patterns [15], considers a simple set of templates for representing four major types of temporal constraints: *existence*, *absence*, *precedence*, and *response* – we thus refer the reader to [6] for more details. In our applications both inside and outside the context of this paper, we have found that such simple temporal expressions suffice to express the vast majority of desires that may typically occur in practice. Users desiring more expressive power can simply write LTL formulae directly.

4.4. Generating Admissible Plans

The situation calculus semantics of the goal model indicate how it can be translated into a planning problem specification. We use PPlan, a preference-

[2]Note that, the semantics for the boolean connectives and quantifiers are already defined in the situation calculus, and thus they require no translation.

based planner [16] as the target of our translation. PPlan reads an action theory and a planning problem specification as well as preference formulae, which are weighted linear combinations of LTL properties. PPlan searches for plans that satisfy both the action theory and the set problem and also the LTL properties in a way that their linear combination is optimized.

To achieve that, PPlan employs an A* best-first search to identify plans from a specified initial situation to a situation that best satisfies a given preference formula. Beginning from the initial situation and the empty plan, the algorithm progresses through possible next situations that form through the performance of actions, aiming at reaching a situation in which the goal formula is satisfied. The A* evaluation function is a prediction of the best and the worst score the preference formula can possibly acquire in later stages, given the current situation. These are calculated by examining whether it is possible for the basic components of the preference formula to be true or false in subsequent situations, given their LTL semantics and the current situation and partial plan.

Given the absence of hierarchical information in the resulting specification, to this original heuristic we added our own extension aiming at exploiting the structure of the goal tree. In particular, in a given situation, where a subset of the leaf-level tasks of the goal model have already been performed, it is possible to calculate an estimation of the maximum and the minimum number of tasks that need to be performed for the root goal to be satisfied. The distance of a candidate plan from satisfying the root goal is used together with the heuristics that are already employed in PPLan. As a result PPlan searches taking into account the structure of the goal tree and as such exhibits in certain cases much better performance in finding solutions for the goal tree. More details can be found in [23].

Note that use of PPlan in this context does not involve optimization. Thus, CFs are simply added as components of equal weight in the linear combination and only plans with optimal weight (i.e. satisfying each and every component) are considered as admissible.

4.5. Policy Trees and Software Behaviors

The set of admissible plans generated by PPlan are converted into a more convenient form which we call policy tree. Policy trees are defined as follows:

Policy Tree Construction. Let P be the set of admissible plans. A policy tree is a tree structure rooted in r for which: a) each node except r represents a task t in P and b) a plan prefix $[t_1, t_2, \ldots, t_i]$ is in P if and only if there exists a node in

the policy tree for which the unique path from r (excluding the r) to that node is a representation of that prefix.

Figure 6 shows an algorithm for generating the tree. That the algorithm satisfies the definition follows from its constructive nature – we omit a detailed proof. The state pointer and the relationship of the tree to code are defined as follows.

State Pointer and Code Instrumentation. The policy tree is supplied with a unique pointer that points to one of the nodes (initially the root). Every node t in the policy tree is associated with one or more chunks of code c_t – following the corresponding association with the tasks of the goal model. We called these chunks c_t the *mapped code fragment* of t. For a task t to be considered performed, at least one of the c_t must be accessed. Such a chunk of code c_t can be accessed in the application if and only if the state pointer points at t's parent, t itself, or any descendant of t in the policy tree. Moving of the policy tree is only allowed from parent to child signifying performance of the task represented by the child.

INPUT: a set of admissible plans P with human-agent tasks removed
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 $\exists$ 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 6: Building the Policy Tree

Having defined the policy tree as such we can claim the following:

Theorem. *The sequence by which tasks are performed for the first time in the application satisfies the original CFs.*

Proof Sketch. As we saw, the output of the preference-based planner guarantees that each and every optimal plan that returns (call, again, this set P) is going to be compliant to the set of preference formulae (CFs being a special case thereof) as per discussion in [16]. Further, as we saw, a task t cannot be performed unless at least one of the associated chunks of code c_t is accessed and executed. Consequently, all we need to show is that there is no sequence of first execution events of chunks of code $c_{t_1}, c_{t_2}, \ldots$ that does not result in a complete or a prefix of a complete plan of $[t_1, t_2, \ldots]$ in P.

We will show this by contradiction. Let us, then, assume that there is a sequence of executions of chunks of code which does not match any plan in P. Thus, assume that only a (potentially empty) prefix up to task t_i, $i \geq 0$ matches a prefix $[\ldots, t_i]$ of a plan in P and access to $c_{t_{i+1}}$ causes performance of t_{i+1} and maps to a plan prefix $[\ldots, t_i, t_{i+1}]$ that is not found in P. But, by definition of the policy tree, access to $c_{t_{i+1}}$ may have happened only if: a) t_{i+1} is a child of the node pointed by the state pointer or b) t_{i+1} is a task that belongs to the path from the root to the node pointed by the state pointer.

In case (a), by construction of policy trees and definition of the state pointer, $c_{t_{i+1}}$ can be accessed if and only if the prefix that results from appending t_{i+1} to the prefix associated with the state pointer is a prefix of a plan in P. If not, which is our assumption, the policy tree will not allow access to $c_{t_{i+1}}$.

In case (b), if t_{i+1} belongs to the path from the root to the state pointer, and given that the state pointer moves only towards the leafs, t_{i+1} has been performed before. As such, the prefix $[\ldots, t_i, t_{i+1}]$ cannot be defined given that t_{i+1} occurs twice in it.

It is therefore necessarily the case that plan prefix $[\ldots, t_i, t_{i+1}]$ resulting from accessing $c_{t_{i+1}}$ when state pointer is at $[\ldots, t_i]$ is a prefix of a plan in P. □

5. Applying Goal-Based Customization

Let us now discuss some of the experiences we acquired from our case studies with our on-line cart system and the ATM simulation.

5.1. Code Development and Instrumentation

Three Layer On-Line Cart System. The on-line cart system we built is a 5,000 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, were 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.

ATM Simulator. The Automatic Teller Machine (ATM) simulator is a Java-based application derived from an exemplar object-oriented analysis and design process, introduced for educational purposes [24]. It is written in Java and contains 47 classes within 8 packages and a total of about 4 KLOC. A goal model was developed to describe different ways by which user actions can be performed to meet user goals pertaining to the ATM use, such as withdraw cash, transfer funds, inquire etc. Examples of such actions are typing a password, giving the card back, printing a receipt, and performing various ATM functions. CFs then constrain which of those actions can be performed and in what order. The ATM simulation can be seen as a model of an interface through which important functions of a real information system are accessed. As such, it is an example of how customizing the details of user interaction is a way to actually enforce customization decisions in entire business processes.

Looking at the results afterwards, we found that, in both systems, identification of mapped code was possible and emerged naturally in the development process. Interestingly, in the three-layer cart system, 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. In the PHP on-line cart system, policy trees are plugged as separate globally visible PHP classes in the application. In the Java-based ATM application, a policy tree class is declared and instantiated within the context of an ATM object, a reference to which is passed to various other classes which perform various ATM actions (e.g. deposit, withdrawal etc). In both studies, 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.

5.2. 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. In the on-line cart example, 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 on-line cart 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 the ATM application, the anchor is the transaction – once one such is over, the state pointer returns to the root node.

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.

5.3. Nested Policy Trees

A direct consequence of choosing anchors for the policy trees is that an application can be designed to contain many and different policy trees instantiated, manipulated and consulted simultaneously at run-time. A useful way by which multiple policy trees can be used together is through nesting. Nested policy trees are a result of viewing certain subtrees of the main goal decomposition as separate and governed by their own "local" CFs. Thus we can ignore the details under those subtrees from the main analysis and treat each of them as a separate goal model. As a consequence, in the resulting policy tree of the main decomposition, some nodes do not represent tasks but hard goals of the goal model to which such separate subtrees are rooted. As such, those nodes can be associated with a separate policy tree dictating the behavioural details for fulfillment of the corresponding goal. Satisfaction of that hard-goal can be assumed only after performance of a sequence of sub-tasks from the root to a leaf of the separate subtree.

More specifically, assume that node t_N of the main policy tree is associated with a hard-goal and as such with a nested policy tree. Once *canBePerformed(t_N)* is true, the nested policy tree is constructed with its state pointer pointing to its root. The application can perform the tasks in the nested tree in a sequence permitted by that nested tree – its state pointer will move accordingly. Once a leaf task of the nested tree is performed the nested tree is destroyed and the state pointer

of the main tree now points at t_N – implying that its performance has been completed. Repetition of t_N is now (or later) possible, by constructing a new nested policy tree and following the same process.

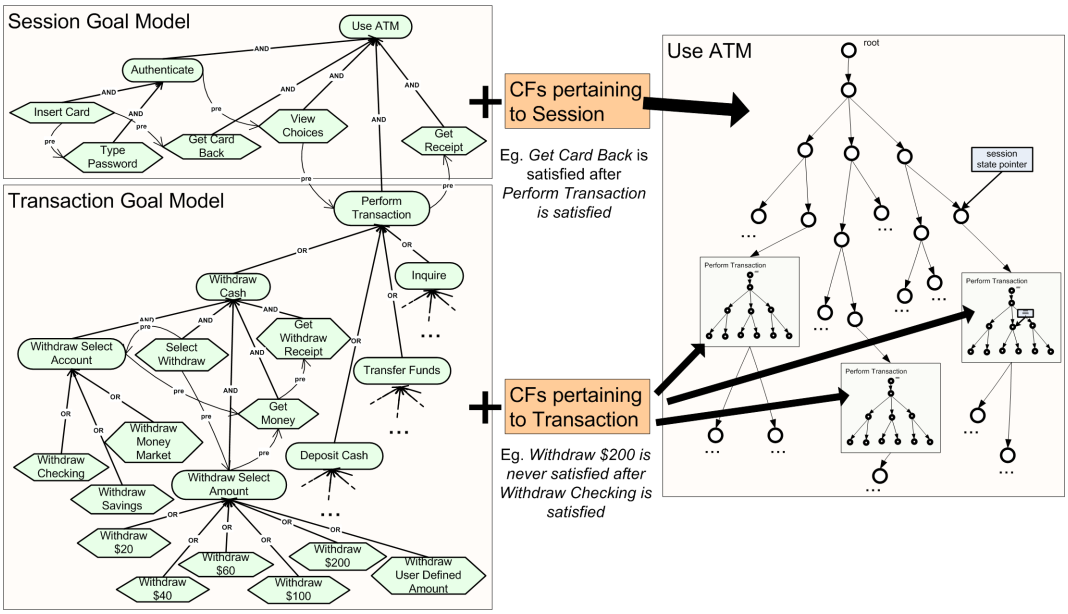

Figure 7: Nested Policy Trees

In Figure 7, an example of nested policy trees as we applied it in the ATM system is shown. Within one session with an ATM, a user may wish to perform several transactions. Each transaction must obey its own customization desires that are independent of the customization desires pertaining to the session. Thus, while the order by which tasks such as authenticating or getting the receipt form one domain of concern (from a customization point of view), the transaction details per se constitute a separate domain of concern. As such, the CFs that are constructed for the *Perform Transaction* subtree are completely separate (mention different tasks, that is) from the CFs that are written for the *Use ATM* goal. The latter treat *Perform Transaction* as a mere task. In the policy tree that results from applying CFs to the *Use ATM* goal, nodes that refer to *Perform Transaction* are nested policy trees, whose structure results from the application of separate CFs over the subtree rooted to that goal.

During execution, the state pointer of the main tree progresses as session tasks are performed, such as inserting the card, entering password, etc. Once *Perform*

Transaction is possible, the corresponding nested policy tree is constructed. The pointer of the nested policy tree progresses as the tasks pertaining to the particular transaction (e.g a cash withdrawal) are performed. Once the transaction is complete (a leaf task of the nested tree is performed) the main state pointer now points to the *Perform Transaction* node, signifying its completion. Another transaction can always be performed before or after the tasks that follow, since by definition the policy tree allows revisiting the same path. However for the particular node, *Perform Transaction*, repetition means that a different nested tree will be constructed and, therefore, a different path can potentially be followed there (e.g. deposit instead of withdrawal).

In our application of nested policy trees in the ATM case, we found that the technique allows for a) more tractable and efficient reasoning about CFs and plans, b) effectively dealing will loops and repetitions. Thus, instead of performing one reasoning task with one large tree, we perform many different reasoning exercises with several different but smaller trees. At the same time, while in the one monolithic policy tree repetition through different branches is not allowed (to maintain satisfaction of the CFs), nested policy trees lift this constraint through assuming that the temporal scope of the "local" CFs does not include repetitions, allowing variation in repetition through resetting the nested tree. The obvious cost is that we cannot impose global customization desires for the entire goal model.

5.4. Dealing with Permutations and Multiple Choices

The advantage of using policy trees 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 challenge is that the developer of a particular step is not aware of the global customization decisions so that she e.g. labels the interface accordingly. In addition, when the developer knows that at a particular point in the code many possibilities exist as to which tasks can be performed and in what order, there is a challenge as to how to organize the sequence of *canBePerformed()* checks.

While the solution to these challenges largely depends on developer intuition and based on the information of the problem at hand, our practices in both applications seem to suggest a strategy that is based on recursive exhaustion of possible

next steps. In particular, for every point in the code where a customization decision is to be made, the developer can (programmatically) identify all tasks that could potentially be performed based on the information in the goal tree. For example after inserting the card to the ATM, the goal model suggests that the tasks: *take card back*, *enter password* and *display receipt warning message* may only follow. At run-time, any subset of these may appear as a child of a policy tree depending on the specified CFs. Through recursive invocations of the *canBePerformed()* function the developer will need to identify that admissible subset. From that she can arbitrarily pick the task to perform next and repeat the process of calculating the feasible set over again. When a need to identify which task has been performed emerges, the predicate *hasBeenPerformed()* can be used. Thus, for arranging the checkout sequence of screens in our on-line cart application, arrays of *canBePerformed()* and *hasBeenPerformed()* checks were 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.

In general the CF-based customization approach allows for separation of the customizability aspect of the application, allowing developers of individual components to work without having to consult e.g. an exhaustive list of possible global customizations. As we saw, however, depending on the nature of the application, some consideration of possibilities often needs to take place, and the goal model, which policy trees must necessarily satisfy, offers a useful representation for that.

5.5. Using Customization Formulae

How easy is it to use CFs in practice? To examine this we tried a variety of customization formulae that we thought to be relevant in realistic cases of ATM and on-line cart systems and observed their effect in constraining system behavior. In the on-line cart system, 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 checkout 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.

In the ATM example, we focussed on our experience in using such systems and observing several different arrangements of steps as well as scenarios pertaining

to the context in which an ATM is used. Thus ATMs that exist in e.g. convenience stores often don't have a mechanism for "swallowing" the card for the entire session, meaning that the card needs to be taken back before even the password is entered (which can be expressed as a precedence CF). Further, ATMs that are expected to be busier and less accessible to maintenance should prevent withdrawal over a certain amount, complicated transactions other than withdrawal, or printing of a receipt (if paper supply is difficult). At the same time ensuring that the user does not forget her card can be a matter of deciding whether to give it back before or after giving the receipt – again arranged through precedence CFs. We can imagine, for example, that third-party ATMs often found in e.g. airports or stores, would like to imitate the interface of the ATMs offered by the bank that the current user is using – information that we assume can be easily recognized in their bank card.

In both our applications, the formulae would translate without problems into systems that behaved accordingly. We 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 possible for 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. [25]), are instead asked to construct and "run" their own customization desires. While this adds significant flexibility and allows for defining customizations that are otherwise currently impossible or difficult (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. Work on preference-based exploration of requirements alternatives [6] 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.

5.6. 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 models of both the on-line cart and the ATM simulation example, while varying the maximum amount of plans. The result for the on-line cart of Figure 1 (21 elements including 16 leaf-level tasks) 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. For example, it took 23 seconds for the planner to return the first 40 plans for CF *browse2*; more plans were found later. 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 for the on-line Cart (in sec)

For the ATM simulation, a model with 57 elements out of which 43 are leaf-level tasks, we tried CFs that correspond to different contexts in which the ATM can be situated. For example, *remoteMall* is a CF concerning an ATM situated in a mall in some remote location, which makes its supply with receipt paper and banknotes an expensive process; as such it does not allow printing a receipt and withdrawal of large amounts of cash. The *mallCommon* CF scenario on the other hand assumes that the mall is more accessible but busier, thus functions other than

withdrawal are allowed. The other scenarios, such as *branch*, allow for a wider range of functions. The performance results for the ATM (using one policy tree) on the same hardware configuration can be seen in Table 9 – times are again in seconds.

Scenario	**Top 20**	**40**	**60**	**80**	**100**
mallCommon	1	2	3	5	6
mallRemote	7	20	54	54	54
streetStandard	68	247	482	482	482
branch	17	56	114	208	1035

Figure 9: Time to Generate the first N Plans for the ATM (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 offering dramatically better performance through utilization of domain knowledge expressed as task hierarchies [26] – that planner has already been used for analysing stakeholder preferences [6]. 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.

5.7. Reflection: Advantages and Challenges

Overall, our exploration with the on-line cart system and the ATM simulation demonstrated three basic advantages of our proposal. Firstly, it makes software customization a requirements problem, whereby users can 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 system design 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 test plans.

Finally, the two applications per se allow for generalizability arguments that cannot exceed certain limits. Firstly, both implementations are relatively small (a few thousands lines of code) 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 both our prototypical systems to be good models of real working systems of their respective types. Secondly, the systems we developed are an example of a web-based system for supporting e-commerce business transactions and a front-end conversing-style of interface for performing a particular task, respectively. 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.

6. 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 and Cheng [27] use temporal logic to specify adaptive program semantics. Further, work by Brown et al. [28] 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. Simmons, on the other hand, uses strategy trees to evaluate alternative reconfigurations of software systems in the context of QoS and structural changes [29]. Our approach differs in that it deals with user goals and behaviour 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 [30]. Lu et al. propose the construction of flexible

business process templates that lay the basic constraints that must be met [31]. Elsewhere [32, 33] 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 [34]. Elsewhere, Liaskos et al. [6] propose the use of a preference-based planner for the purpose of eliciting and understanding stakeholder attitudes. Here we move this idea one step further to show how the result of planner-based analysis can be used to actually customize a system. In general, existing frameworks do not include an implementation approach, and when they do, this is restricted to specialized frameworks such as workflow engines [33] or e.g. BPEL-based service composition platforms [30].

The extensive literature on software composition, on the other hand (for a taxonomy see e.g. McKinley et al. [12]), is focusing on specific technologies, frameworks or techniques by which composition can be implemented – e.g. composition of services [35], the AHEAD framework and its descendants [36, 37] or Aspect Orientation [38], Domain Specific Languages and Generators [39, 40]. Use of existing AI planning applications to service composition, in particular, ([41, 42] – cf. Rao and Su for a survey [13]), 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.

7. 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 allow the 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 instead of selecting 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 applications in the on-line cart system and the ATM simulator 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] P. Oreizy, N. Medvidovic, R. N. Taylor, Architecture-based runtime software evolution, in: Proceedings of the 20th International Conference on Software Engineering (ICSE'98), Washington, DC, USA, ISBN 0-8186-8368-6, 177–186, 1998.

[2] J. Kramer, J. Magee, Self-Managed Systems: an Architectural Challenge,

in: Future of Software Engineering (FOSE '07), Washington, DC, USA, 259–268, 2007.

[3] J. Mylopoulos, L. Chung, S. Liao, H. Wang, E. Yu, Exploring Alternatives During Requirements Analysis, IEEE Software 18 (1) (2001) 92–96.

[4] S. Liaskos, S. A. McIlraith, J. Mylopoulos, Towards Augmenting Requirements Models with Preferences, in: Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09), Auckland, New Zealand, 565–569, 2009.

[5] S. Liaskos, S. A. McIlraith, S. Sohrabi, J. Mylopoulos, Integrating Preferences into Goal Models for Requirements Engineering, in: Proceedings of the 10th International Requirements Engineering Conference (RE'10), Sydney, Australia, 135 – 144, 2010.

[6] S. Liaskos, S. McIlraith, S. Sohrabi, J. Mylopoulos, Representing and reasoning about preferences in requirements engineering, Requirements Engineering (REJ) 16 (2011) 227–249.

[7] S. Liaskos, M. Litoiu, M. D. Jungblut, J. Mylopoulos, Goal-Based Behavioral Customization of Information Systems, in: Proceedings of the 23rd International Conference on Advanced Information Systems Engineering, CAiSE 2011, vol. 6741 of *Lecture Notes in Computer Science*, Springer, 2011.

[8] A. Dardenne, A. van Lamsweerde, S. Fickas, Goal-Directed Requirements Acquisition, Science of Computer Programming 20 (1-2) (1993) 3–50.

[9] E. S. K. Yu, J. Mylopoulos, Understanding "Why" in software process modelling, analysis, and design, in: Proceedings of the 16th International Conference on Software Engineering (ICSE'94), 159–168, 1994.

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

[11] C. Gacek, M. Anastasopoulos, Implementing product line variabilities, SIGSOFT Software Engineering Notes 26 (3) (2001) 109–117.

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

[13] J. Rao, X. Su, A Survey of Automated Web Service Composition Methods, in: J. Cardoso, A. Sheth (Eds.), Semantic Web Services and Web Process Composition, vol. 3387 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 43–54, 2005.

[14] A. Pnueli, The Temporal Logic of Programs, in: Proceedings of 18th Annual Symposium on Foundations of Computer Science (FOCS77), 46–57, 1977.

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

[16] M. Bienvenu, C. Fritz, S. McIlraith, Planning with Qualitative Temporal Preferences, in: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR06), Lake District, UK, 134–144, 2006.

[17] R. Reiter, Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press, 2001.

[18] D. Nau, Y. Cao, A. Lotem, H. M. noz Avila, SHOP: Simple Hierarchical Ordered Planner, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), 968–973, 1999.

[19] A. Gerevini, D. Long, Plan Constraints and Preferences in PDDL3, Tech. Rep. Department of Electronics for Automation, University of Brescia, 2005.

[20] X. Wang, Y. Lesperance, Agent-oriented requirements engineering using ConGolog and i*, in: AOIS-2001, Bi-Conf. Workshop at Agents 2001 and CAiSE'01., 2001.

[21] G. Gans, M. Jarke, G. Lakemeyer, T. Vits, SNet: A modeling and simulation environment for agent networks based on i* and ConGolog, in: Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02), Toronto, Canada, 2002.

[22] A. Gabaldon, Precondition Control and the Progression Algorithm, in: D. Dubois, C. Welty, M. Williams (Eds.), Principles of Knowledge Representation and Reasoning: Proceedings of the 9th International Conference (KR2004), AAAI Press, 634–643, 2004.

[23] S. Liaskos, Acquiring and Reasoning about Variability in Goal Models, Ph.D. thesis, University of Toronto, 2008.

[24] R. C. Bjork, An Example of Object-Oriented Design: An ATM simulation. http://www.cs.gordon.edu/courses/cs211/atm-example, 2004.

[25] S. Liaskos, A. Lapouchnian, Y. Wang, Y. Yu, 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, 9–18, 2005.

[26] S. Sohrabi, J. A. Baier, S. McIlraith, HTN Planning with Preferences, in: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09), Pasadena, CA, USA, 1790–1797, 2009.

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

[28] G. Brown, B. H. C. Cheng, H. Goldsby, 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), ACM, New York, NY, USA, 23–29, 2006.

[29] B. Simmons, Strategy-trees: A Novel Approach to Policy-Based Management, Ph.D. thesis, University of Western Ontario, 2010.

[30] A. Lapouchnian, Y. Yu, J. Mylopoulos, Requirements-driven design and configuration management of business processes, in: Proceedings of the 5th International Conference on Business Process Management (BPM'07), Brisbane, Australia, 246–261, 2007.

[31] R. Lu, S. Sadiq, G. Governatori, On managing business processes variants, Data and Knowledge Engineering 68 (7) (2009) 642 – 664.

[32] F. Gottschalk, W. M. van der Aalst, M. H. Jansen-Vullers, M. La Rosa, Configurable Workflow Models, International Journal of Cooperative Information Systems (IJCIS) 17 (02) (2008) 177+.

[33] S. W. Sadiq, M. E. Orlowska, W. Sadiq, Specification and validation of process constraints for flexible workflows, Information Systems 30 (5) (2005) 349 – 378.

[34] A. G. Sutcliffe, N. A. M. Maiden, S. Minocha, D. Manuel, Supporting Scenario-Based Requirements Engineering, IEEE Transactions on Software Engineering 24 (12) (1998) 1072–1088.

[35] L. Baresi, 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), 114–123, 2010.

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

[37] S. Apel, C. Kastner, C. Lengauer, FEATUREHOUSE: Language-independent, automated software composition, in: Proceedings of the 31st International Conference on Software Engineering (ICSE '09), 221 –231, 2009.

[38] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 313+, 1997.

[39] J. C. Cleaveland, Building Application Generators, IEEE Software 5 (4) (1988) 25–33.

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

[41] S. Sohrabi, N. Prokoshyna, S. A. McIlraith, Web Service Composition Via Generic Procedures and Customizing User Preferences, in: Proceedings of the 5th International Semantic Web Conference (ISWC06), Athens, GA, USA, 597–611, 2006.

[42] D. Wu, B. Parsia, E. Sirin, J. Hendler, D. Nau, Automating DAML-S Web Services Composition Using SHOP2, in: The SemanticWeb - ISWC 2003, vol. 2870 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 195–210, 2003.