Cedar: Engineering Role-Based Adaptive User Interfaces for Enterprise Applications

Pierre Akiki and Arosha K. Bandara and Yijun Yu

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University
Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk
Cedar: Engineering Role-Based Adaptive User Interfaces for Enterprise Applications

Pierre A. Akiki
The Open University
Milton Keynes, U.K.
pierre.akiki@open.ac.uk

Arosha K. Bandara
The Open University
Milton Keynes, U.K.
a.k.bandara@open.ac.uk

Yijun Yu
The Open University
Milton Keynes, U.K.
y.yu@open.ac.uk

ABSTRACT
Feature-bloated enterprise applications such as customer relationship management (CRM) and enterprise resource planning (ERP) are very large scale, encompassing millions of lines-of-code and thousands of user interfaces (UI). Also, these applications are sold as generic off-the-shelf products to be used by people with diverse needs in required feature-set and backgrounds such as skills, culture, etc. Although several approaches have been proposed for adapting UIs to various user profiles, little work has focused on simplifying enterprise application UIs through engineering adaptive behavior. We define UI simplification as a mechanism for increasing usability through adaptive behavior by providing users with a minimal feature-set and an optimal layout based on their individual profile. In this paper we present Role-Based UI Simplification (RBUIS), a tool supported approach based on our CEDAR architecture for simplifying enterprise application UIs through engineering role-based adaptive behavior. RBUIS is integrated in our generic platform for developing adaptive model-driven enterprise UIs. Our approach is validated from the technical and end-user perspectives by applying it to developing a prototype enterprise application and user-testing the outcome.

Author Keywords
Simplification; Adaptive user interfaces; Role-based; Enterprise applications; Model-driven engineering

ACM Classification Keywords
[Software Engineering]: D.2.11 Software Architectures - Domain-specific architectures; D.2.2 Design Tools and Techniques - User interfaces; [Information Interfaces and Presentation]: H.5.2 User Interfaces – User-centered design

General Terms
Design; Human Factors

INTRODUCTION
The functionality of software applications tends to increase with every release increasing the visual complexity [22]. This phenomenon, referred to as “bloatware” [23], has a negative impact on usability especially for users who do not require the complete feature-set. Also, users have diverse backgrounds (e.g. skills [32], culture [29], etc.), which tend to affect their layout preferences. In this paper, we present Role-Based UI Simplification (RBUIS) as a mechanism for increasing usability by providing users a minimal feature-set and an optimal layout based on their individual profile.

Feature-bloated enterprise applications are sold as generic off-the-shelf products to be used by people with diverse needs in required feature-set and backgrounds. These applications serve various purposes in an enterprise’s functional business areas (e.g. accounting, inventory, etc.) [25]. The literature clearly indicates that these systems suffer from usability problems. One example is given by a study carried out in the Nordic countries [20], which showed that almost 40% of the users find enterprise applications difficult to use to a certain extent. User interface simplification would enhance the usability of these applications by catering to the variable user needs.

One method to achieve UI simplification is for enterprise applications to become adaptive/ adaptable, respectively referring to the ability of tailoring software applications automatically/manually. Adapting a UI’s feature-set could enhance user satisfaction [21] and make complex applications easier to use on mobile devices and by cognitively impaired users [14]. Also, adaptive/adaptable behavior has been used for tailoring UI layout based on various aspects such as: “Accessibility” [27], “Platform” [4], “Natural Context” [3], etc. However, to meet enterprise application needs we propose the following criteria, for implementing UI simplification, based on the scale and complexity of these applications and the existing literature:

- providing a scalable, extensible, and tool supported mechanism capable of integrating in the development and post-development phases of enterprise applications and accommodating multiple adaptation aspects;
- programming role-based adaptive behavior through both visual and code construct hence allowing developers as well as I.T. personnel to define and reuse it;
- preserving designer input [28] on concrete UIs during adaptation instead of a fully mechanized UI generation;
- reducing user confusion [21] by proposing the adapted UI as a simplified alternative to the initial design rather than adapting it while the user is working;
We intend to meet the proposed criteria by using interpreted runtime models that allow more advanced adaptations and could be integrated as part of a generic solution offered as a service. The approach is based on our CEDAR architecture [1]. This paper makes the following contributions:

- An approach called Role-Based User Interface Simplification (RBUIS) composed of the following:
  - A mechanism for minimizing the feature-set at runtime by applying roles on task models
  - A mechanism for representing layout optimization adaptive behavior both visually and through code using workflows that are executed on concrete UI (CUI) models at runtime

- Cedar Studio, our Integrated Development Environment (IDE) for devising adaptive model-driven enterprise UIs, provides tool support for our approach

- An evaluation of our approach with a set of studies based on two criteria: (1) technical feasibility and scalability, and (2) end user satisfaction and efficiency

The remainder of this paper is structured as follows. We discuss the related work and briefly explain of our CEDAR architecture. Then, we elaborate on how RBUIS could be applied for minimizing the feature-set and optimizing the layout based on CEDAR. Next, we provide an overview of our IDE Cedar Studio, and an example on building adaptive behavior models for use in our approach. Finally, we highlight the results of a study for evaluating RBUIS.

BACKGROUND AND RELATED WORK

This section briefly discusses existing work in terms of the four criteria we established in the introduction. We categorize existing work into feature-set minimization and layout optimization. These categories make up the simplification process and address the variable user needs in enterprise UIs. Additionally, we provide a brief overview of the CEDAR architecture based on which RBUIS is based.

Feature-Set Minimization

The simplification process should start by providing each user with a minimal feature-set to reduce unnecessary “bloat” [23] present in feature-rich enterprise applications.

Providing a multi-layered user interface design is promoted for achieving universal usability [30]. Other researchers propose using two UI versions, one fully-featured and another personalized, in bloated applications [21]. An earlier research work proposes the use of a “training wheels” UI that blocks advanced functionality from novice users [6]. These propositions demonstrate a sound theoretical basis, useful for providing the users of feature-bloated software applications with a minimal feature-set.

Yet, the demonstrated examples, a basic text editor [30] and the Word 2000 menu [21], are not as complex as enterprise applications. Additionally, a generic, scalable, extensible, and tool supported mechanism is required for applying the feature-set minimization concept in practice.

Approaches from product-line engineering [28] are used to tailor software applications for multiple product-lines. Some studies particularly address tailoring UIs based on a product line approach. MANTRA [4] relies on designing UIs using an abstract model, which is later used to adapt the UI for multiple platforms by generating code particular to each platform. Although product-line based approaches aim at tailoring software applications, focus is given to design-time (product-based) adaptation whereas runtime (role-based) adaptive behavior is not addressed.

Role-based tailoring of the feature-set is sought after in commercial enterprise applications. Microsoft Dynamics CRM 2011 [33] and SAP GuiXT [34] offer such a mechanism. Yet, it is not generic enough to be used with other applications and it requires creating and maintaining multiple copies of the UI manually. Our approach is more general because it operates at the model level.

Layout Optimization

Providing the optimal layout based on each user’s background complements the simplification process. For example, SAP’s usability (world’s leading ERP [15]) is mostly affected by “Navigation” and “Presentation” [31] and its UI does not adapt to the user’s skills [32].

Many existing research works target the adaptation of the UI layout. Each one uses a different approach for handling the adaptive functionality.

Fully mechanized UI construction has been criticized in favor of applying the intelligence of human designers for achieving higher usability [28]. It would be better if the designer could manipulate a concrete object rather than its abstraction [10]. Supple is a system mainly capable of generating UIs adapted to each user’s motor abilities [14]. The UI is automatically generated from an abstract model and the adaptation is treated as an optimization problem. This automation prevents designer input on the concrete UI (CUI), which is the representation as concrete widgets (button, text box, etc.), making the system difficult to adopt for enterprise applications. Also, this approach has been criticized [27] for exceeding acceptability performance times.

Providing the adapted UI as an alternative version while maintaining access to the original full-scale UI, was shown to have higher user acceptance [21]. Yet, many platforms perform the adaptations while the UI is in use. MASP targets ubiquitous UIs in smart environments by adapting the UI whenever a context change is detected [3]. The MyUI platform also opts for adapting UIs while the user is working in order to prompt for user confirmation [27]. The choice of this adaptation mode is due to the ubiquitous nature of the target systems (e.g. Smart Homes). Since enterprise applications have a less ubiquitous nature with
more complex WIMP style UIs, proposing the adapted UI as an alternative helps in avoiding confusion.

Scalability is important in any approach targeting adaptive enterprise UIs. DynaMo-AID incorporates a design process for the development of context-aware UIs that are adaptable at runtime [7]. This system generates what is referred to as a task tree forest. As indicated by other researchers, once each tree corresponds to the tasks possible in each context, the combinatorial explosion would make this approach hard to scale for complex systems.

The CEDAR Architecture

We created the CEDAR architecture [1] (Figure 1) as an approach for devising adaptive enterprise application UIs based on interpreted runtime models instead of code generation. The dynamic nature of these models gives more flexibility in performing UI adaptations and allows us to implement CEDAR as a generic service oriented solution that can be consumed by APIs using different technologies.

Figure 1. The CEDAR Architecture

We based CEDAR on three existing reference architectures: (1) CAMELEON [5] (UI Abstraction), (2) Three Layer Architecture [17] (Adaptive System Layering), and (3) Model-View-Controller (MVC) (Implementation).

In the coming sections we will show how our role-based approach for simplifying enterprise application UIs (RBUIS), which is based on the CEDAR architecture, addresses the four criteria established in the introduction.

ROLE-BASED UI SIMPLIFICATION (RBUIS)

To simplify UIs, we need to provide a minimal feature-set and an optimal layout based on each user’s profile. The feasibility of adapting a single UI designed for the least constrained profile was demonstrated in previous research [13]. Our simplification mechanism will follow the same approach. The designer will devise the UI for the least constrained profile at design-time. Afterwards, a role-based approach will be used to simplify the UI at runtime based on each user’s profile.

The standard for role-based access control (RBAC) could be utilized by enterprises for protecting their digital resources [11]. In RBAC, “Users” are assigned “Roles”, which in turn are assigned permissions on “Resources”. In our case the users are the enterprise employees logging into the system with their accounts, and the resources that we want to apply roles to, are the UI and adaptive behavior models. We merged the role-based approach with UI simplification to create Role-Based User Interface Simplification (RBUIS) in the spirit of RBAC. In RBUIS, roles are divided into groups representing the aspects based on which the UI will be simplified (e.g. literacy level, domain knowledge, job title, etc.). RBUIS will be applied after deploying the software in the enterprise. Managing this process could be a joint responsibility between the personnel from the software company in charge of the deployment process and the enterprise’s IT personnel.

Our approach to user interface simplification (RBUIS) comprises the following elements that support feature-set minimization and layout optimization:

**Role-Based UI Models** support feature-set minimization by assigning roles to task models (e.g. ConcurTaskTrees (CTT) [26]) to provide a minimal feature-set based on each user’s profile. This approach allows a practical realization of the concept of multi-layer interface design [30].

**Role-Based Adaptive Behavior Models** support layout optimization through workflows that represent adaptive UI behavior visually and through code. The adaptation is applied on the concrete user interface (CUI) models. Afterwards, adaptive behavior models are tied to roles to specify how the UI will be optimized for each set of users.

**User Feedback for Refinement** allows the users to reverse feature-set minimizations and layout optimizations, and to choose possible alternative layout optimizations. Keeping users involved increases their UI control [21] and feature-awareness [12] affected by adaptive/reduction mechanisms. The following sections describe our approach in detail.

**MINIMIZING THE FEATURE-SET**

In order to minimize the feature-set we will rely on the concept of multi-layer interface design. This concept allows the users to control different sub-sets of the UI at any moment. For example, novice users could be given access to layer 1 and as they develop expertise could gain access to the upper layers at any time. RBUIS provides a practical approach for controlling the different UI layers. The meta-model for applying RBUIS on task models (CTT) is shown in Figure 2. CTTs were chosen to represent the task models due to their support of temporal constraints, which help in determining if simplifying a task could affect other tasks.
Feature-Set Minimization with RBUIS

Applying RBUIS on task models allows the minimization of the feature-set by revoking access to tasks based on roles hence achieving a role-based multi-layer interface design. Since we are initially designing the UI for the least constrained profile, the default policy will grant all roles access to all the tasks. This could be considered as a layer containing all the features. Afterwards, access could be revoked by allocating roles to tasks thereby creating separate layers, which users could gain role-based access to. Since users could be allocated multiple roles from the existing role categories, priorities will be used to provide enough flexibility to specify how roles override each other. Upon assigning the access rights to block tasks based on roles, a property (concrete operation) will specify whether to make a task invisible, disable it (keep data visible / protect data), or fade it until first use. The task model is mapped to the Abstract UI (AUI), which is in turn mapped to the CUI to hide, disable, or fade the relevant UI widgets.

![Figure 2. Meta-Model of Applying RBUIS on Task Model](image)

Less Time Consuming Access Rights Allocation

Since enterprise applications encompass a large number of tasks that are used by hundreds of users, we need to make the allocation of access rights on the task models as little time consuming as possible. Traditionally, enterprise application users are allocated roles. This could be considered as a positive starting point. We will resort to the following features to minimize the time taken to allocate roles to tasks in the task models:

- A default policy grants access to all roles on all the application’s task models hence making it only necessary to override this policy where access should be revoked. Each task will be implicitly allocated a fixed role called “All-Roles”, which represents all the roles in the system and is granted access to execute the task. Access to the task will be revoked to all other explicitly assigned roles.
- Sub-tasks will inherit the access rights of the parent tasks while maintaining the ability to override these rights.
- In some cases the same functionality is replicated in many places within the application. Usually developers create visual components (CUI level) that could be reused in different places. By making task models reusable within one another, access rights allocated to a task model could roam with it whenever it is used again while maintaining the ability to override the initial rights. This feature is illustrated in Figure 2 with the recursive relationship “Is Embedded In” on the “TaskModel” class. Each embedded task model is connected to a source and a target task as shown on the “TaskModelRelation” class.
- Rules could be defined and applied to sets of task models based on each task’s properties (ID, name, type, etc.). RBUIS rules are defined through our support tool (Cedar Studio) in the form of conditions using SQL syntax. Also, check lists are given to associate task models and roles with each rule. One basic example would be to revoke access to the role “Cashier” on all “Interaction” tasks with the words “Enter Discount” in the task name.

Applying RBUIS to Task Models at Runtime

Based on the CEDAR architecture, the UI models will be loaded on the server and the adaptive engine will apply RBUIS at runtime. To apply the concrete operations on the CUI, the Task Model is mapped to the AUI, which is in turn mapped to the CUI. A certain order should be followed to perform the elimination since each user could be allocated multiple roles simultaneously. The meta-model allows the assignment of priorities on different levels. The designer could specify where the priority is read from (“RoleGroup”, “Role”, “TaskRole”, “UserRole”). Task-based assignments have a higher priority than rule-based ones unless specified otherwise. The following example demonstrates the process assuming the priorities were set at the “TaskRole” level:

- **UserA**: Novice, Manager
- **TaskX**: 1. All-Roles (Allow) 2. Accountant (Deny-Hide) 3. Novice (Deny-Disable)

An excerpt of our algorithm is shown in Algorithm 1, the full version is included in the appendix. Following this algorithm “UserA” is allowed to perform “TaskX” since “Manager” has the highest priority. In contrast, if “Novice” had a higher priority than “All-Roles”, then “UserA” would have been denied access to “TaskX” hence disabling its CUI as indicated by the concrete operation.

The running time of our algorithm is estimated to be polynomial: \(O((m \ast (n \ast l \ast p \ast (2 \ast j \ast \log j + k) + n)))\), where \(m = \) Num. of Task Models, \(n = \) Num. of Tasks in a Task Model, \(j = \) Num. of User Roles, \(k = \) Num. of Blocked CUI Elements for a Task, \(p = \) Num. of Parent Tasks for a Task, and \(l = \) Num. of Task Roles.
Algorithm 1. Feature-Set Minimization (Excerpt)

1. Simplify-Task (TaskID, UserRoles[], TaskRoles[], UIModel)
2. foreach ur in UserRoles //Determine the Primary Role
3. tr ← ur.RoleRef
4. if tr = null then tr ← TaskRoles.GetRole(All-Roles)
5. ur.Priority ← tr.Priority;
6. UserRoles.OrderBy(Priority)
7. PrimaryRole ← UserRoles.First()
8. if PrimaryRole.RoleRef = All-Roles //Apply Concrete Operation to CUI
9. blkdAUI← GetBlkdAUI(TaskID, UIModel.TMToAUIMap)
10. blkdCUI← GetBlkdCUI(blkdAUI, UIModel.UIToCUIMap, UIModel.CUI)
11. foreach element in blkdCUI
12. switch PrimaryRole.ConcreteOperation
13. case Hide: element.Visible ← false; break;
14. case Disable: element ReadOnly ← true; break;
15. case Protect: element ReadOnly ← true;
16. element.MaskChar ← '*'; break;
17. case Fade: element.Opacity ← 30%; break;

Model Checking

Since the access rights are being allocated by humans, model checking is needed to ensure that critical constraints are not violated. This allows our tool to issue appropriate warnings and errors. Several techniques exist for defining and evaluating constraints on models. For example, the Object Constraint Language (OCL) could be used to define constraints on UML diagrams. Furthermore, there are numerous tools that could be used for model checking (Z3, Spec#, Formula, etc.). In our case we need to define constraints on task models represented by CTTs. Since our approach is based on the CEDAR architecture, all the models are being stored in a relational database. This allows the model checking to be performed using SQL, which is more familiar to many developers and I.T. personnel than constraint languages such as OCL. The following example shows a constraint and its SQL based solution in Listing 1.

Constraint: A sub-task should not be blocked for all the assigned roles because it will not be accessible by any user.

Listing 1. Task Model Constraint Example using SQL

```sql
With SetTasks as (Select TM.TaskModelID, TM.TaskModelName, TK(TaskID, TK.TaskName) TaskModel, TM INNER Join TaskModelTask @ TM On TM.TaskModelID = TK.TaskModelID Where TaskModelID in (@ModelIDs))
UserAccessOnTasks as (Select TaskModelID, TaskID, COUNT(case UR.CanExecuteTask when 1 then 1 else 0 end) as CanExecuteCount From SetTasks Cross Apply LoadSortedUserRoles(ToolStripMenuItemID, TaskID) Where UR.UserRolePriority = 1 Group By TaskModelID, TaskID)
Select SetTasks, Inner Join UserAccessOnTasks UAT On ST.TaskID = UAT.TaskID and ST.TaskModelID = UAT.TaskModelID Where CanExecuteCount = 0
```

Constraints are defined in Cedar Studio and associated with task models through a system variable (“@ModelIDs”). Predefined functions such as “LoadSortedUserRoles” could be used in model constraints and extended when necessary. In this case the function loads the users and their assigned roles sorted by the priority of execution according to a certain task. The SQL statement would return the tasks that are violating the constraint, to be displayed on the screen.

Feature-Set Minimization Example

Although enterprise applications contain many complex examples, a basic example has been purposefully chosen in order to accommodate screen shots in the paper. Complex real-life examples were considered in our evaluation.

The example illustrated in Figure 3 shows part of a task model built in Cedar Studio for a “Customer Maintenance” UI common in ERP systems. The lock shaped buttons allow the application of RBUIS on any task. In this case, the tasks called “Financial Information” and “Picture” circled in a dashed line are marked as simplified indicating that RBUIS has been applied. In the case of “Financial Info,” the access rights will get inherited by its sub-tasks. We considered a role called “Cashier” requiring a version of the UI showing only the “Name,” “Phone,” and “Address.” This allows users working as cashiers to enter the initial information for a new customer on the counter without having to handle other details that could be added later. The initial version of the Final UI (FUI) is illustrated in Figure 4(a), and the one simplified for the role “Cashier” is illustrated in Figure 4(b). In this example, the concrete operation in RBUIS was set to “Hide” hence the widgets became invisible.

Figure 3. Simplified Customer Maintenance Task Model

Figure 4. Feature-Set Minimization of Customer UI
OPTIMIZING THE LAYOUT

Providing users with an optimal layout could be based on various aspects (e.g., computer literacy, cognition, screen size, etc.). In this section we present our generic mechanism for devising adaptive behavior for such criteria. Enterprise applications require an approach that allows developers as well as I.T. personnel to implement adaptive behavior. Our feature-set minimization mechanism allows RBUIS to be applied visually and through code based rules. Similarly, our layout optimization mechanism allows the definition of adaptive behavior using a mix of visual and code constructs embedded in adaptation workflows. The meta-model for applying this mechanism on the CUI is shown in Figure 5.

**Layout Optimization with RBUIS and Workflows**

The representation of adaptive behavior has a great impact on the extensibility of any adaptive system. Most adaptive UI state of the art systems tend to employ an arbitrary design that hardcodes adaptation behavior within the software application, severely minimizing its reusability and extensibility. A graphical tool is suggested for hiding the complexity of defining UI adaptation rules [19]. This tool might not be able to handle all possible scenarios due to the limited use of a high level visual representation.

To balance between ease of use and flexibility our approach combines high level adaptation operations and low level programming constructs by using both visual and code based representation. Workflows are not strange to enterprise applications due to their use for devising customizable and reusable business rules that could be separated from the software code. With appropriate tool support, workflows could also provide visual programming constructs (control structures, error handling, etc.). Additionally, it is possible to define code based adaptation operations that integrate within the visual workflow.

Our approach uses tool supported workflows, which could represent adaptive behavior with: (1) visual programming constructs, (2) compiled code libraries and dynamically interpreted scripts. The workflows are executed at runtime on the CUI models to perform the necessary adaptation.

To implement the workflows in practice we are using the Windows Workflow Foundation (WF), which is part of the .NET framework. WF provides a visual designer, which we integrated into Cedar Studio. This designer provides the ability to visually design activity workflows using a rich set of constructs, which could be saved in an XML based format then reloaded and executed when an adaptation is needed. Furthermore, the supported constructs could be extended through external compiled class libraries developed in C# or VB.NET and dynamically integrated with our tool. We have used this capability to develop a construct capable of integrating within a workflow and executing non-compiled script code. We currently support Iron Python but other scripting or transformation languages (e.g. XSLT) could be integrated in the future.

![Figure 5. Meta-Model for RBUIS and Workflows on CUI](image)

**Applying RBUIS with Workflows at Runtime**

Layout optimization is also based on our CEDAR architecture. After the feature-set is minimized, the workflows will be executed on the CUI by the adaptive engine. Afterwards, the FUI will be transferred to the client to be rendered on the screen. The process of selecting the workflows to be applied based on the user’s role is illustrated in Algorithm 2 through an excerpt of our algorithm assuming the priority is read from the “Roles” class. The running time of our algorithm is established to be polynomial: \( O(2m \log m + 2n \log n) \), where \( m \) = Num. of User Roles and \( n \) = Num. of Workflows to be Executed.

**Algorithm 2. Layout Optimization (Excerpt)**

1. \texttt{Optimize-Layout} (UserRoles[], Roles[], UIModel, LayoutID)
2. \texttt{foreach} ur \texttt{in} UserRoles // Determine the Primary Role
3. \texttt{tr} ← \texttt{Roles.GetRole(ur.RoleRef)}
4. \texttt{if} \texttt{tr} = \texttt{null} \texttt{then} \texttt{tr} ← \texttt{Roles.GetRole(All-Roles)}
5. \texttt{ur.Priority} ← \texttt{tr.Priority};
6. \texttt{UserRoles.OrderBy(Priority)}
7. \texttt{PrimaryRole} ← \texttt{UserRoles.First()}
8. \texttt{WorkflowsToExecute[]} ← \texttt{GetWorkflows(PrimaryRole, LayoutID)}
9. \texttt{WorkflowsToExecute.OrderBy(ExecutionOrder)}
10. \texttt{foreach} workflow \texttt{in} WorkflowsToExecute // Execute Workflows
11. \texttt{workflow.Execute(UIModel)} // Execution Time Depends on Content

**Layout Optimization Example**

This example builds on the previous one illustrated in the feature-set minimization section. We consider two roles “Sales Officer” and “Novice”. The “Sales Officer” requires the fully-featured UI illustrated in Figure 4 (a). The “Novice” requires layout optimizations that make functions accessible through on-screen buttons rather than a context-menu, and trading list boxes for radio buttons to fit more items on the screen. The workflow illustrated in Figure 6, represents the adaptive behavior by using three different techniques: (a) list boxes are substituted with radio button
groups using visual programming constructs, (b) function accessibility is set to high by calling an Iron Python script, and (c) the UI is refitted by calling a C# layout algorithm.

The optimized UI in Figure 7 shows the functions for the image (add, remove, etc.) and address text-area (bold, italic, etc.) on the screen. In contrast, the version in Figure 4 (a) provided these functions through a context-menu. Also, the payment terms list-box was substituted with a radio-button group that displays more items on the screen. Some factors (e.g. access. of functions) are set by “Adaptive Properties” on the “LayoutWidget” class in the meta-model (Figure 5). The implementation of such factors is done in the widget.

The users can click the chameleon icon in the top right corner of the UI (Figure 4 (b), & Figure 7) to show a list of the applied adaptation operations as illustrated in Figure 8. Afterwards, the users can uncheck any reversible operation (feature-set minimization or layout optimization) and apply the changes for one time only or for future use as well. Furthermore, layout optimizations have another feature that allows the users to choose from possible alternatives. This is achieved by assigning workflows to groups as shown in the meta-model (Figure 5). Workflows in the same group could serve as alternatives. For example, a group could encompass several workflows for adapting the selection widget (combo box, list box, radio buttons, etc.). After the user applies the changes, a request will be sent to the server to re-simplify the UI and exclude the operations that he or she unclicked. In case the user decides to keep the changes for future use, based on the CEDAR architecture, the changes would get stored and he or she will gain access to an alternative version of the UI. The example operations illustrated in Figure 8 are related to the simplified UI in Figure 4 (b). The operations inform the user that the UI parts pertaining to the financial information and image are unused by the user’s role (Cashier) hence were eliminated. In this example, if the user unchecks both operations and applies the changes, the simplified UI in Figure 4 (b) will revert back to the original version in Figure 4 (a). If an operation is set as “irreversible by users” (e.g. due to security reasons) the check box would be disabled and a message would notify the user of the reason. If a feature depends on other disabled features, the user is informed that these features should be enabled as well. The dependency is determined from the CTT temporal operators and is defined on the meta-model (Figure 2) through the recursive relationship “Depends On” on the “Task” class.

Even though in the case of feedback the UI is changing while the user is working, the user’s initiation of the action reduces confusion due to the awareness and understating of the adaptation that is going to take place.
DEVELOPING APPLICATIONS WITH CEDAR STUDIO

Cedar Studio is our Integrated Development Environment (IDE) that supports the development of adaptive model-driven UIs for enterprise applications based on the CEDAR architecture. Due to space limits we will briefly describe its features in this paper. The tool could be observed in operation through an online demo video [35].

We created Cedar Studio in the form of an IDE to provide developers and I.T. personnel with an ease of access to all the visual designers and editors in one place. Currently, it supports visual designers for: (1) Task Model, (2) Domain Model, (3) AUI Model, (4) CUI Model, and (5) Workflows. Also, it supports a combination of visual and code designers for (1) Task Role Assignments and RBUIS Rules, (2) Model Constraints, and (3) Dynamic Scripts. Additionally, this tool supports automatic generation and synchronization between the various levels of abstraction (Task Model, AUI, and CUI) with the possibility to make manual changes at any of these levels.

Figure 9. Cedar Studio Supporting CEDAR Architecture

Cedar Studio was designed as a tool for supporting our CEDAR architecture through UI and adaptive behavior models that would get stored in a relational database to provide easier runtime management and interpretation. The implementation of CEDAR is provided as a service that is consumed by Cedar Studio and technology specific APIs that allow enterprise applications to integrate with our solution. An API would include the client components illustrated in Figure 1. To test our approach we developed an API and Toolkit in C# for the Windows Presentation Foundation. APIs for other presentation technologies (HTML, Java Swing, etc.) could be devised by anyone and used in combination with Cedar Studio for developing adaptive enterprise applications capable of benefiting from our simplification mechanism and any future extensions.

Adaptive UI behavior (widget hiding, substitution, etc.) could leave gaps and deformations in the layout, which are not esthetically desirable and could increase the navigation time (Fitts’ Law). We required a mechanism to preserve plasticity denoting the UI’s ability to adapt to the context-of-use while preserving its usability [9]. Hence, we devised an algorithm to refit the layout based on its initial manual design by filling the gaps and adjusting the widgets’ position based on their new size and initial location chosen by the developer. This technique creates a balance between fully automated approaches that generate the UI from an abstract model [14] and manual approaches that require developing and maintaining multiple CUI versions [34].

Cedar Studio is meant to be used during the development and post-development phases by software developers, deployment teams, and I.T. personnel. The UI models could be devised at the development phase and the simplification behavior could be added during the deployment phase according to the needs of each enterprise.

The following subsections illustrate the features of Cedar Studio in more details.

Task Models

The visual designer illustrated in Figure 10 supports the composition of task models using ConcurTaskTrees (CTT). The designer supports a tree layout algorithm that can automatically adjust the presentation of large task models. Visual and code based support is provided for allocating roles to tasks in order to simplify user interfaces.

Figure 10. Task Model Designer

The lock-shaped button on each task allows a visual allocation of access rights using the UI shown in Figure 11. A default policy (“All-Roles”) is implicitly assigned to grant access to all the roles on any given task. This policy could be overridden by explicitly assigning roles from different groups (Figure 11 a) to each task. The concrete operation and whether it is reversible by the user is specified for each role (Figure 11 b). Tasks could inherit or override roles assigned to their parent task (Figure 11 c). The order of the roles could be changed to indicate their priority. The priority source could be manually assigned (Figure 11 d).
The allocation of roles to tasks could also be done through SQL based rules. RBUIS rules are written in the form of an SQL condition conforming to our meta-model (Figure 2). This condition is then assigned roles and allocated to the task models on which it should be executed. Cedar Studio provides an editor for RBUIS rules (Figure 12) and the ability to validate the SQL syntax and display existing errors in the “Error List”.

The second level of abstraction, namely AUI models, could be automatically generated from task models. It is possible to override the default mappings through the UI shown in Figure 14. A task is allocated one or more AUI elements.

AUI Model Designer

The generated AUI is easily modifiable through the visual AUI designer illustrated in Figure 15. Since AUI models are a modality independent representation, the designer shows each element as a box with a name, icon, and color. This designer allows AUI containers to be nested within one another and provides an easy to use flow style for visually manipulating the AUI elements. The properties box allows the modification of an element’s properties including its type. As suggested by other researchers [28], placeholder elements are used upon deletion to maintain the mapping between the models. The type of the placeholder could be switched to an AUI element type without affecting the mapping. It is also possible to delete an element without resorting to a placeholder. New elements could be added from the toolbar and manually mapped to their related tasks in the task model.
CUI models could be automatically generated from AUI models similarly to how AUI models are generated from task models. An interface is also provided to manually adjust the default mappings as shown in Figure 16.

Figure 16. Mapping AUI to CUI

The input of a human designer is highly desirable for achieving higher usability [28] through the manipulation of concrete objects rather than merely their abstraction [10]. Cedar Studio provides a highly developed CUI designer (Figure 17) by integrating and extending the “Windows Forms” designer of “Visual Studio .NET”. This designer has been time tested and is highly used for developing enterprise application user interfaces. It provides developers and I.T. personnel the ability to easily modify CUI models in a familiar and flexible environment. Similar to the AUI designer, the CUI designer supports placeholders upon deletion in addition to complete deletion of elements which could be manually replaced and mapped to the AUI model. A rich toolbar is provided including both basic (e.g. date-time picker) and complex (e.g. data grid) widgets required by enterprise applications.

Figure 17. CUI Model Designer

Workflow Designer

Workflows are common in enterprise applications for representing business rules. Our approach takes advantage of workflows for providing the ability to represent adaptive behavior both visually and through code. This provides developers and I.T. personnel with the ability to implement this behavior through a straightforward visual canvas (Figure 18 c). Workflows could be associated with roles and the CUI models to be executed on.

We integrated the “Windows Workflow” designer of “Visual Studio .NET”. This designer provides a rich set of standard visual programming constructs (Figure 18 b), which could be dynamically extended with custom activities (Figure 18 a) written in “C#” or “VB.NET”. One of the extensions we built supports calling external adaptive behavior written using the dynamic scripting language “Iron Python”. Cedar Studio stores the workflows in an XML format that allows them to be loaded and executed dynamically when needed.

Figure 18. Adaptive Behavior Workflow Designer
Cedar Studio supports an “Iron Python” script editor as illustrated in Figure 19. Scripts are created separately and could be called from within any workflow by selecting the script, specifying the method to call, and passing it the appropriate parameters. The entire process is done visually through the workflow designer.

Running the UI with a user allocated the role “Cashier”, the version illustrated Figure 21 in will be displayed.

Figure 19. Scripts Code Editor

Testing Output from Within Cedar Studio
Cedar Studio provides developers with the ability to run the devised user interfaces with and without adaptations using “Run” and “Run As” commands respectively. The “Run” command simply executes the initial least constrained version of the UI whereas the “Run As” operation prompts the developers to enter a user and executes the version of the UI corresponding that user’s roles. This functionality allows developers to easily test the developed UI and its associated adaptive behavior from within the IDE.

In the following example we shall illustrate one of the cases we considered for testing our approach and support tool. The UI illustrated in Figure 20 represents a fully-featured “Sales Invoice”. We considered a role called “Cashier” requiring a simplified version of this “Sales Invoice”.

When a user’s role is modified (Cashier to Manager, Novice to Expert, etc.), the adaptation will dynamically change according to the new role. This conforms to the concept of multi-layer interface design [30].

In the coming section we demonstrate an example of how adaptive behavior models could be devised, based on a user background aspect relevant to enterprise applications, in order to be used with our simplification mechanism.

BUILDING ADAPTIVE BEHAVIOR MODELS
One way to build adaptive behavior models for our system is to determine a user background aspect that influences enterprise application usability, statistically test its effect on UI alternatives, and implement the adaptive behavior for these alternatives using Cedar Studio. The outcome would be a general role-based adaptive model that could be refined for specific tasks and users using our feedback mechanism.

One such aspect discussed in the literature is “Computer Literacy”. We compiled a list of factors based on which the UI could be adapted, and tested the effect of computer literacy on users’ preferences. Although this list is not comprehensive, it allows us to test our system against factors discussed in the literature and relevant to enterprise applications. We grouped the factors under “Presentation” and “Navigation”, as shown below, due to the impact of these categories on enterprise application usability [31].

Presentation: Layout Grouping (Tab Page, Sub-Window, Group Box), Multi-Record Visualization (Grid, Carousel, Detail Form), Simple Selection Widget (Combo, Slider, Radios), Multi-Record Input (Scrolling Grid, Non-Scroll Grid, Form), Accessibility of Functions (High, Medium, Low), Information Density (High, Medium, Low), Text versus Graphics (Text Only, Image Only, Image & Text)
Navigation: Multi-Doc UI (New Window, New Page, New Tab), Search the UI (Go to Widget, Filter, Filter & Re-layout) Navigation Structure (Menu, Tree, Panel)
The factors “Accessibility of Functions”, “Information Density”, “Text versus Graphics”, and “Navigation Structure” are tackled from a cultural perspective [29]. Another research work addresses “UI Layout Grouping” and “Data Selection Widget” from the perspective of each user’s motor abilities [14]. Search is emphasized as being an important part in enterprise application navigation [31]. Even though it usually indicates the ability to search for data we consider the factor “Searching the UI” could help in enhancing navigation. Since enterprise applications are known to be highly data intensive we decided to consider “Multi-Record Data Visualization” and “Multi-Record Data Input” in our study. Also, we included the “Multi-Document Interface” factor to consider various navigation schemes between user interfaces already displayed on the screen.

To validate whether there is a variance in the preference of users from different computer literacy levels, we devised an online interactive survey [36] to test our factors. The survey had one independent variable namely “Computer Literacy” with three values: “Novice”, “Intermediate”, and “Expert”. The dependent variables are the previously discussed set of factors with their possible values in addition to an open ended value “Other” that allows participants to specify any possible value, which was not included in the list.

One limitation of surveys inquiring about UI preferences is the order in which the participant sees the various options. Participants generally tend to like the first option that they see hence creating a certain level of bias in the survey’s outcome [29]. By considering this potential bias we designed our survey to display the different UI options as small randomized snippets all on one page. One example is illustrated in Figure 22. The options are interactive to allow the participant to provide better assessment. Participants were asked to rate each of the options on a seven point Likert scale indicating their satisfaction.

In order to classify participants under one of the three computer literacy categories, we have to inquire about their computer skills without giving the impression that we are subjecting them to an intelligence test. Hence the best option was to allow the participants to evaluate themselves through a series of questions. We think that they provided honest answers since the survey is completely anonymous. We selected a set of eight questions from a computer literacy test that was validated in the literature [16].

**How would you rate your ability to (1-7)?**
- Use a word processor to create documents
- Learn a software package that you never used before
- Use an operating system (Windows, Mac OS, Linux, etc.)
- Discuss strength and weaknesses of software packages

**To what extent would you agree with the following (1-7)?**
- I do not need someone to tell me the best way to use a computer.
- I would prefer to learn a new computer software package on my own.

The given ratings on the computer literacy questions were averaged to determine the participant’s level as follows: (Novice: 1,2,3), (Intermediate: 4,5), (Expert: 6,7).

We classified participants under the following groups: Novice (n=22), intermediate (n=22), and expert (n=45). A two-way ANOVA was performed to examine the effect of computer literacy on user interface preferences. There was homogeneity of variance between groups as assessed by Levene's test for equality of error variances. We report measures that were significant (p < .05) and partial eta-squared ($\eta^2$) due to its significance in human-computer interaction research [18]. Partial eta-squared could be interpreted as a small (.01), medium (.06), or large (.14) effect size [8]. We highlight the following factors: Multi-Document UI (F (4,288) = 4.507, $p = .002$, $\eta^2 = .059$), Navigation Structure (F (4,228) = 4.526, $p = .002$, $\eta^2 = .074$), UI Layout Grouping (F (4,234) = 3.824, $p = .005$, $\eta^2 = .061$). Upon observing the Quantile-Quantile plots we found the data to be normally distributed with some occasional exceptions. Therefore, we also report the outcome of the Kruskal-Wallis (non-parametric) ANOVA as a confirmation to our results: Multi-Document UI (H (2) = 14.587, P = 0.01), Navigation Structure (H (2) = 8.662, P = 0.013), UI Layout Grouping (H (2) = 6.447, P = 0.04).

One should note that from a technical perspective adaptive behavior for all the factors is deurable using our platform. Also, different factors could have a statistically significant variability affected by criteria other than computer literacy. For example “Accessibility of Functions” and “Information Density” were shown to be impacted by culture [29].
ASSESSING CEDAR STUDIO
User interface development tools should take into consideration the following criteria [24]:

- **Threshold and Ceiling**: The “threshold” represents the difficulty in learning and using the tool, and the “ceiling” relates to how advanced the tool’s outcome can be. An ideal tool would have low threshold and high ceiling.

- **Path of Least Resistance**: Developers should be guided to construct the UI in an appropriate manner by making the right approach easier to follow than the wrong one.

- **Predictability**: Any automated approach provided by the tool should be predictable to the developers using it.

- **Moving Targets**: The tool should be able to keep up with the rapid developments in user interface technology.

Upon designing and developing Cedar Studio we tried to meet the abovementioned criteria as much as possible. It might not be feasible to achieve low threshold and high ceiling in all cases. This is due to the learning curve created by any additional features that would allow the tool to produce a more advanced outcome. Yet, we aimed towards achieving a proper balance between the two. This is done by integrating automated generation and synchronization between models (low threshold), alongside the possibility of conducting manual adjustments (high ceiling). Furthermore, the visual designers provide the ability to produce an advanced outcome by simply understanding the semantics of the model (low threshold / high ceiling). In the cases where coding could be used, a visual designer alternative was provided (e.g. Visual Workflows instead of Scripts, Visual Role Assignments instead of RBUIS Rules) and the language the most familiar to developers was chosen (e.g. SQL instead of OCL for Model Verification).

The path of least resistance is maintained by allowing developers to easily apply the model-driven approach. The automated generation of models representing the different levels of abstraction and the mapping between them saves the time of having to perform this manually. This automatic generation preserves predictability by allowing the developer to customize the default mappings between the different model elements (e.g. abstract input to text box). Furthermore, the support for visual adjustment and resynchronization provides an easy way to customize what was automatically generated.

Concerning the Moving Targets criterion, the model driven approach supported by Cedar Studio was initially created as a mechanism to absorb the effects of changes in technology. This approach allows our IDE to be independent from presentations technologies and to evolve more easily alongside them. If new techniques for building user interfaces or even new UI types emerge in the future models are a good approach to cope with such change since relying on the existing abstract representations allows the regeneration of different concrete user interfaces types.

EVALUATING ROLE-BASED UI SIMPLIFICATION
Our simplification mechanism was evaluated using an online interactive survey [37] with a user interface pair composed of an initial and a simplified UI. We selected the “Customer Maintenance” form of the SAP ERP system. The initial version of this UI contains numerous nested tab pages and dozens of fields. Yet, users with different roles in the enterprise require a simpler version for managing basic customer records.

We developed a copy of SAP’s UI alongside a simplified version containing the fields used to create a basic customer record. The fields were selected based on the variability in SAP’s user needs [34]. The concrete operation was set to “Hide” with some fields being reversible by the user, and the widgets were regrouped accordingly.

Participants were asked to fill a set of fields required for creating a basic customer record using both UI versions. In the case of the simplified UI some of the fields had to be retrieved through the user feedback screen, allowing us to test how participants react to this feature.

In some cases, participants tend to like the first UI option they see hence creating certain bias in a study’s outcome. To avoid this potential bias we presented half of the participants with the initial UI first and the other half with the simplified one first. After each task, participants were asked to answer the System Usability Scale (SUS) questions, which allow us to detect usability differences between the two UI versions. Also, the time taken to complete each task was recorded.

We hypothesize that: Simplifying enterprise application UIs based on roles improves user satisfaction and efficiency.

The participants (n=25) never used the selected UI before. A Wilcoxon Signed Ranks Test showed that simplifying the user interface based on roles elicited a statistically significant improvement in both SUS usability score (Z = -3.530, P = 0.0004) and task completion time (Z = -2.644, P = 0.008) hence confirming our hypothesis. The median SUS score was 50 for the initial UI and 67 for the simplified one. The median time taken to complete the task (seconds per input field) was 19 for the initial UI and 11 for the simplified one. The results are illustrated as box plots in Figure 23.

The results were also reflected in the comments of some participants about the simplified version being more efficient whereas the initial UI made it complicated to locate fields. Also, the ease of use of the feedback mechanism was reflected by the fact that 80% of the participants were able to use it by only referring to a few words of instruction on its purpose.
CONCLUSIONS AND FUTURE WORK
We presented our Role-Based UI Simplification (RBUIS) approach, comprising feature-set minimization and layout optimization. RBUIS is based on our CEDAR architecture that is offered as a generic extensible service allowing the addition of adaptive behavior as needed. The scalability of our mechanism was shown by our complexity analysis.

Additionally, we introduced Cedar Studio our IDE that provides tool support for developing and maintaining adaptive enterprise UIs. We described how it can be used to represent role-based adaptive behavior visually (role assignment, constructs in workflows) and through code (RBUIS rules, compiled code and scripts in workflows).

Furthermore, we provided an example on building adaptive behavior models for our mechanism, based on statistically significant differences in user interface preferences.

Finally, we conducted a user study to evaluate RBUIS. The study showed a statistically significant improvement in user satisfaction and efficiency for simplified UIs. The outcome of the study also reflects the importance of a model-based approach that preserves designer input, made on the CUI, during adaptation. Also, by offering the UI as a role-based alternative our approach reduces confusion created by adaptations conducted while the user is working.

In the future we will extend our mechanism to support UI simplification in scenarios that require the use of multiple user interfaces for fulfilling a task. Additionally, more user studies will be conducted using eye-tracking in addition to measuring user satisfaction and efficiency.

REFERENCES


APPENDIX

Algorithm A.1: Feature-Set Minimization Running Time

Variables:

\[ m = \text{Number of Task Models} \]
\[ n = \text{Number of Tasks in a given Task Model} \]
\[ j = \text{Number of User Roles} \]
\[ k = \text{Number of Blocked CUI Elements for a given Task} \]
\[ p = \text{Number of Parent Tasks for a given Task} \]
\[ l = \text{Number of Task Roles} \]

Legend:

CON = Constant
LOG = Logarithmic
POL = Polynomial

Total Running Time: \( O \big( m \times (n \times l \times p \times (2 \ j \times \log j + k) + n) \big) \)

<table>
<thead>
<tr>
<th>Type</th>
<th>Cost</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>POL</td>
<td>c5</td>
<td>O(n * l * p)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>//Re-enabled Read-Only Containers with Non-Read-Only Children</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DisabledCUIContainers[] -- Select c From UIModel.CUIElements[]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Where c.ReadOnly = false and child.ElementID = c.ElementID).Count() &gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c6 .</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c7 .</td>
</tr>
<tr>
<td></td>
<td></td>
<td>foreach cuiEl in DisabledCUIContainers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cuiEl.IsReadOnly = false</td>
</tr>
</tbody>
</table>
|      |      | }
| POL  | c4   | O(1) |
|      |      | //Get the Task Model’s Associated Tasks, Mappings, and CUI |
|      |      | Tasks[] -- Select t From UIModel.Task |
|      |      | Where t.TaskModelID = TaskModelID |
|      |      | . |
|      |      | . |
| POL  | c3   | O() |
|      |      | //Simplify All the Task Model’s Tasks |
|      |      | foreach task in Tasks[] |
|      |      | if TasksUnblockedByTheUser.Contains(task) = false |
|      |      | TaskRoles[] -- Get-Task-Roles(task, UIModel) |
|      |      | Where m.TaskModelID = TaskModelID |
|      |      | . |
|      |      | . |
| POL  | c2   | O(1) |
|      |      | TMToAUIMapping[] -- Select m From UIModel.MappingTaskModelToAUI[] |
|      |      | Where m.TaskModelID = TaskModelID |
|      |      | . |
|      |      | . |
| POL  | c1   | O() |
|      |      | //Load UI Related Information From Database |
|      |      | UserRoles[] -- GetUserRolesFromDB(UserRef) |
|      |      | UIModel -- GetUIModelFromDB(UserInterfaceID) |
|      |      | . |
|      |      | . |
| POL   | c7   | O(1) |
|      |      | foreach cuiEl in DisabledCUIContainers |
|      |      | cuiEl.IsReadOnly = false |
|      |      | . |
|      |      | . |
| POL   | c6   | O() |
|      |      | //Simplify All Associated Task Models |
|      |      | tasksToSimplify[] -- Select t From UIModel.Task |
|      |      | Where t.TaskModelID = TaskModelID |
|      |      | . |
|      |      | . |
| POL   | c5   | O(n) |
|      |      | //Minimize-Feature-Set (UserRef, UserInterfaceID) |
|      |      | foreach t in UIModel.TaskModel[] |
|      |      | Simplify-Task-Model(tm.TaskModelID, UserRoles, UIModel) |
|      |      | . |
|      |      | . |
| POL   | c4   | O(1) |
|      |      | //Get the Task Model’s Tasks |
|      |      | Tasks[] -- Select t From UIModel.Task |
|      |      | Where t.TaskModelID = TaskModelID |
|      |      | . |
|      |      | . |
| POL   | c3   | O() |
|      |      | //Simplify-Task-Model (TaskModelID, UserRoles[], UIModel, TasksUnblockedByTheUser[]) |
|      |      | foreach tm in UIModel.TaskModel[] |
|      |      | Simplify-Task-Model(tm.TaskModelID, UserRoles, UIModel) |
|      |      | . |
|      |      | . |
| POL   | c2   | O(1) |
|      |      | //Get the Task Model’s Associated Tasks, Mappings, and CUI |
|      |      | Tasks[] -- Select t From UIModel.Task |
|      |      | Where t.TaskModelID = TaskModelID |
|      |      | . |
|      |      | . |
| POL   | c1   | O() |
|      |      | //Load UI Related Information From Database |
|      |      | UserRoles[] -- GetUserRolesFromDB(UserRef) |
|      |      | UIModel -- GetUIModelFromDB(UserInterfaceID) |
|      |      | . |
|      |      | . |
Running Time: $O(2j \times \log j + k)$

Simplify-Task (TaskID, UserRoles[], TaskRoles[], TMToAUIMapping[], AUIToCUIMapping[], CUIElements[])

{  //Order Assigned Roles by Task Role Priority
  foreach ur in UserRoles
    tm - Select From TaskRoles
    Where t.RoleRef = ur.RoleRef
  .
  CON c3 O() if tr = null
    tr - Select t From TaskRoles
    Where t.RoleRef = All-Roles
  .
  .
  CON c5 O() ur.Priority - tr.Priority;
  .

LOG O(\lceil j \times \log j \rceil) UserRoles.OrderBy(Priority)

CON c6 O() primaryRole = UserRoles.First()
CON c7 O() if primaryRole.RoleRef # All-Roles
  //Simplify CUI
  Simplify-CUI(PrimaryRole, TaskID, TMToAUIMapping,
  AUIToCUIMapping, CUIElements)
  .

Running Time: $O(k)$

Simplify-CUI (PrimaryRole, TaskID,
TMToAUIMapping[], AUIToCUIMapping[], CUIElements[])

{  blockedAUIElementIDs[] - Select el.AUIElementID
    From TMToAUIMapping
    Where el.TaskID = TaskID
  .
  CON c2 O() blockedCUIElementIDs[] - Select el.CUIElementID
    From AUIToCUIMapping[]
    Where blockedAUIElementIDs.Contains(el.AUIElementID) = true
  .
  CON c3 O() blockedCUIElementIDs[] - Select el.CUIElements
    Where blockedCUIElementIDs.Contains(el.WidgetID) = true
  .

Apply Concrete Operation to CUI

POL c4 O() switch PrimaryRole.ConcreteOperation
  .
  POL c5 O() case Hide
    element.Visible = false; break;
  .
  POL c5 O() case Disable
    element.Enabled = false; true; break;
  .
  POL c5 O() case Protect
    element.Enabled = false; true;
  .
  POL c5 O() case Fade
    element.Opacity = 30%; break;
  .

Running Time: $O(\lceil l \times p \rceil)$

Get-Task-Roles (Task, UIModel)

{  TaskRoles[] - Select tr From UIModel.TaskRoles
    Where tr.TaskID = Task.TaskID
  .
  CON c2 O() if Task.ParentRoleInheritance = Merge
    ParentTask - (Select t From UIModel.Task[]
    Where t.TaskID = Task.ParentTaskID).First()
  .
  .
  CON c4 O() if ParentTask = null {return TaskRoles;}
  POL c5 O(\lceil l \times p \rceil) TaskRoles.Merge(Get-Task-Roles(ParentTask, UIModel))
  CON c6 O() return TaskRoles
  .
}
Algorithm A.2: Layout Optimization Running Time

**Variables:**

- \( m \) = Number of User Roles
- \( n \) = Number of Workflows Assigned to the Primary Role

**Legend:**

- CON = Constant
- LOG = Logarithmic
- POL = Polynomial

**Total Running Time:** \( O(2m \log m + 2n \log n) \)

<table>
<thead>
<tr>
<th>Type</th>
<th>Cost</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>POL</td>
<td>( c_1 )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_2 )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_3 )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_4 )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_5 )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>LOG</td>
<td>( c_6 )</td>
<td>( O(m \log m) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_7 )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_8 )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_9 )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_{10} )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>CON</td>
<td>( c_{11} )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

**Pseudocode**

```plaintext
Running Time: \( O(2m \log m + 2n \log n) \)

Optimize-Layout ([UserRoles[], Roles[], UIModel, LayoutID, WorkflowsCancelledByUser[], AlternativeWorkflows[]])

//Order Assigned Roles by Workflow Role Priorities
PO
POL c1 \( O(m) \)
foreach ur in UserRoles
    tr ← Roles[].GetRole(ur.RoleRef)

CON c2 \( O(1) \)
if tr = null

CON c3 \( O(1) \)
    tr ← Roles[].GetRole(All-Roles)

CON c4 \( O(1) \)
    ur.Priority ← tr.Priority;

CON c5 \( O(1) \)

LOG c6 \( O(m \log m) \)
UserRoles.OrderBy(Priority)

CON c7 \( O(1) \)
primaryRole ← UserRoles.First()

CON c8 \( O(1) \)
Workflows[] ← Get-Workflows(primaryRole, LayoutID, AlternativeWorkflows)

CON c9 \( O(1) \)
Workflows = Select w From Workflows
    Where WorkflowsCancelledByUser.Contains(w) = false

CON c10 \( O(n \log n) \)
Workflows.OrderBy(ExecutionOrder)

CON c11 \( O(n) \)
    foreach workflow in Workflows[]
    workflow.Execute(UIModel)

//Execution Time depends on the Workflow’s Content
```

//Execution Time depends on the Workflow’s Content