Towards Task-Based Linguistic Modeling for designing GUIs

Iyad Khaddam, Nesrine Mezhoudi, Jean Vanderdonckt
Louvain Interaction Laboratory, Louvain School of Management (LSM) – Place des Doyens, 1
Université catholique de Louvain (UCL) – B-1348 Louvain-la-Neuve, Belgium
{iyad.khaddam, nesrine.mezhoudi, jean.vanderdonckt}@uclouvain.be

RÉSUMÉ
The linguistic perspective emphasizes the use of linguistic taxonomy to classify (partition) graphical user interface concepts and elements on several linguistic levels with clearly-defined interfaces between levels. This perspective is based on Nielsen's Virtual Protocol for Interaction that consists of several linguistic levels: goal, pragmatic (task), semantic, syntactical, lexical, alphabetical and physical.

A linguistic modeling is modeling the graphical interface by abstracting each linguistic level. The aim of the linguistic modeling is to enhance the maintainability quality of the graphical user interface model as defined in ISO-25010:2011, by enhancing sub-qualities of modularity, analyzability and modifiability.

Recent research reported on the linguistic perspective and the linguistic modeling requirements. In this paper, we elaborate more towards a linguistic modeling by modeling the task level; the high abstract level in the linguistic stack. Our contribution is an executable hierarchical task model that fulfils the specific needs towards linguistic modeling.

Mots Clés
Task model ; Architecture and formalism of interactive systems; Linguistic modeling.

ACM Classification Keywords
H.5.2: Graphical User Interfaces; H.5.3: Theory and Models; H.5.m: Miscellaneous.

INTRODUCTION

Developing usable User Interfaces (UIs) is a challenging and a complex task. This complexity mainly comes from the heterogeneity of contexts of use. Users interact using different devices (desktop, mobile...etc.), with different goals, cultures and capabilities and within different environments and situations. The perfect UI is hard (pragmatically impossible) to achieve. To cope with such continuously changing domain we need to prepare for changes that are expected at any time.

Enhancing maintainability of the GUI enhances the ability to implement changes. ISO/IEC 25010:2011 defines maintainability as [1] "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers". This quality is sub-divided into sub-qualities represented in table 1.

ISO 25010:2011: definitions of sub-quality
Modularity: The degree to which the system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
Reusability: The degree to which an asset can be used in more than one software system, or in building other assets.
Analyzability: The degree to which the software product can be diagnosed for deficiencies or causes of failure in the software, or for the parts to be modified to be identified.
Modifiability: Corrections, improvements or adaptations of the software to changes in environment and in requirements and functional specifications.
Testability: The degree to which the software product enables modified software to be validated.

Table 1. Maintainability sub-qualities in ISO-25010:2011.

Recently, researchers reported on a linguistic perspective to develop Graphical User Interfaces (GUIs) [2]. The aim of this perspective is to enhance the **maintainability** quality in the developed GUI. This perspective is based on Nielsen's Virtual Protocol for Interaction [3]. It employs a well-defined linguistic taxonomy to repartition GUI concepts and elements on several linguistic levels. These levels are mutually-exclusive: a GUI concept/element belongs to one and only one level. The resulting categories from the linguistic taxonomy (the linguistic levels) are: goal, task, semantic, syntax-time, syntax-space, widgets and widgets properties. The linguistic taxonomy does not only classify GUI concepts and elements, but also classifies changes on the GUI on different levels. Repartitioning GUI elements on levels leads to loosely-coupled modules on each level. This would enhance modularity on the GUI and consequently enhances the maintainability. A background on the linguistic perspective and how it enhances modularity is introduced in the next section. The linguistic perspective differentiates GUI input elements that change the task state (task input elements) from others and classifies them on the task level.

Model-Based User Interface (MB-UI) approaches gained a lot of interest from the Human-Computer Interaction (HCI) community due to their benefits and promises [4].

http://dx.doi.org/10.1145/2820619.2820636

The ultimate goal of these approaches is to address complexity of interaction in UIs. This complexity is addressed by mainly: (1) abstraction: to isolate the problem of interest and thus concentrate on the more important aspects, and (2) enforcing an engineering discipline by establishing systematic approaches to develop UIs.

A linguistic modeling approach aims at abstracting modules on linguistic levels defined in the linguistic perspective [12]. The marriage between modeling approaches and the linguistic perspective promises to have the benefits of both. A *linguistic model* of the GUI promises to follow an engineering discipline with enhanced maintainability in the GUI. But, is modeling from a linguistic perspective feasible?

The answer to the above question is to build a linguistic model by abstracting each linguistic level. The highest linguistic levels are the goal and the task. The contribution of this paper is to introduce a task model that fits the linguistic task level. This is a first step towards building a linguistic model. The difference between our model and other task models is mainly in: (1) Task identification criteria (adopted from the literature) to separate what is a task (on the task level) and what is refinement on how to carry out the task (on the semantic level, the lower level), (2) A modeling notation that helps in identifying needed task input elements in the GUI. Researchers and task modelers interested in addressing maintainability of GUIs from a linguistic perspective are the primary targeted audience of this paper.

In the next section, we give a background on the linguistic perspective to allow understanding how the linguistic perspective enhances maintainability. Our contribution and requirements for the linguistic task model are further explained at the end of section two which motivates the need for a linguistic task model. In section three, we review some existing task models to identify shortcomings in fulfilling linguistic task model requirements. Section four introduces our linguistic task model's notation and gives an example on how to use the notation. Section five presents the simulator and execution of the task model. Finally, section six states conclusion and future works.

A BACKGROUND OF THE LINGUISTIC PERSPECTIVE

Before explaining the linguistic perspective, we spot the light on the role of classification in modeling, which is usually implicitly considered. The linguistic perspective emphasizes the need for an explicit classification and a well-defined taxonomy to enhance maintainability of the GUI.

Modeling and classification

Each UI model has a specific point of view to the UI domain. This point of view guides abstraction efforts of the modeler and thus controls modeling decisions. For example, distribution of the domain concepts and elements in Cameleon Reference Framework (CRF) [13] depends on the point of view adopted. CRF defines four levels of abstraction for a UI: (1) the task model, (2) the Abstract UI model (AUI), that is modality and platform independent, (3) the Concrete UI model (CUI), that is platform-independent and finally (4) the Final UI model (FUI) that is platform-dependent. This point of view implies an implicit classification that classifies concepts like "the color" (a modality-dependent, platform independent concept) on the CUI level. The same applies on "layout" concepts: they are classified at the CUI level.

Classification and maintainability

The general procedure to modify a GUI passes through the following steps: (1) Locate the place of the change, (2) identify the element(s) to be changed, (3) Delineate the propagation of the change (all related and affected elements), (4) Modify the GUI and finally (5) Test the GUI.

The first step is related to identifying the module in the GUI. In a MB-UI approach that employs several levels of abstraction (the case of CRF), this step is related to locating the model/level concerned with the change. Locating the right module/level is related to the modularity sub-quality in the GUI. This sub-quality is better addressed in these approaches than in single-model approaches. A single-model approach employs only one model to generate the final UI (like from task model to the final UI).

The second and third steps are related to the analyzability sub-quality (see analyzability definition). The fourth step is related to the modifiability sub-quality and the last step is related to the testability sub-quality.

Classification impacts the modularity: what modules/levels should be defined in a GUI, and what concepts/elements should exist in each module. The example on how "the color concept" is classified in CRF shows this impact on MB-UI approaches. A mutually-exclusive classification of GUI concepts and elements enhances the modularity quality because it repartitions GUI concepts and elements on separate modules/levels (the classification categories). A concept/element can't repeat on two modules/levels. If such a repetition exists, locating the place of change is affected. In other words: repetition of concepts and elements increases coupling between modules/levels and thus affect maintainability.

The Linguistic Perspective

The basic idea of the linguistic perspective is to have a mutually-exclusive classification of GUI concepts and elements that is based on a well-defined taxonomy. The classification categories define the modules of the GUI. As GUI concepts and elements are repartitioned on these modules, we need also to define interfaces between them. The resulting modular GUI is expected to enhance

the maintainability because modules are loosely coupled with well-defined interfaces between them.

In 1986, Jacob Nielsen introduced his Virtual Communication Protocol of Interaction. He aims to use his protocol to analyze the interaction between the human and the machine. His protocol employs a linguistic classification of the interaction. The protocol consists of seven levels of interaction (ordered by level of abstraction): goal, task, semantic, syntax, lexical, alphabetical and physical. These levels form a stack of interaction, where each lower level implements required services for the upper level (realize the upper level). Therefore, the interaction is analyzed in a refined way from goals to physical level.

Researchers in [2] turned the protocol of Nielsen from an analysis tool to a taxonomy tool to classify GUI concepts and artifacts. This taxonomy also classifies activities of developing a GUI on linguistic levels, like: Where to identify a task? Where to define navigation between containers? Where to define placement on the screen? The output of their work is a classification of GUI concepts and elements on adapted levels from the original protocol. These levels are presented in table 2 in the first column. The second column shows the activities classified per level, while the third column shows the main GUI concepts and elements classified (repartitioned) per level, and grouped into key representative groups. The last column shows the communication interface between levels. Definitions of key terms introduced in the table 2 are below:

Detailed functions	Defined on the semantic level to realize a task. They detail how a task is carried out. They should identify all needed UI elements (input and output elements) on the GUI for performing the task.
UI elements	A UI element is either of type input or output. UI elements are concretized on the screen as widgets. They can be visible (like a label, a text box, a button or another widget) or non-visible (like a finger gesture, a mouse click, a key-press or others). Concretizing input elements is an activity on lower levels.
Task input element	is an input element that can exclusively change a task state (like complete a task or roll it back), but can't do anything else (like acquire data from the user of execute a detailed function). None-task input elements can't change a task state. Anyway, they can execute a detailed function or acquire data from the user.
Syntax-time container	is a logical group of UI elements that should be available together at the same time. Availability of a UI element on the screen here is not related to visibility of the concretizing widget. A UI element might be concretized as a non-visible input (keyboard shortcut or gesture or others). We may use the shorter term "time container" for these containers.
Navigation Elements	are responsible of moving on the time axe from one time-container to another. They can be concretized as concrete input widgets (like a button or a link). An example is the next or back buttons in a GUI wizard.
Syntax-space container	is a group of UI elements that belong to concurrent syntax-time containers (time containers that appear at the same time). A syntax-time container defines placement rules that control its UI elements placement on the screen. These rules define acceptable placement of UI elements on the screen. We may use the shorter term "space container" for them.

Level	Activities	Key GUI concepts & elements		Communication Interface
Goal	the goal of the user of the GUI	goal		-
Task	Define tasks needed to attain the goal	Task input elements	UI Elements	-Realize goals.
Semantic	Define detailed functions needed to carry out a task	Non-task input and output elements		-Realize tasks by defining needed detailed functions.
Syntax-time	Define distribution of UI elements on time by defining time containers. Define navigation	Time containers. Navigation elements.	Containers	-Realize distribution of UI elements on time.
Syntax-space	place UI elements in time containers on the screen	Space containers		-Realize placement of UI elements on the screen.
Widgets	map UI elements to GUI widgets	Concrete GUI widgets	GUI Widgets	-map UI elements and containers to widgets.
Widgets Properties	Set properties of GUI widgets	Properties of widgets		Implement widgets

Table 2. The linguistic classification of GUI activities and GUI concepts and elements.

The linguistic perspective illustrates that concretization of concepts on the final GUI as widgets might lead to loss of their relation to the concept. For example: A button that completes a task on the GUI is not at the same level of abstraction as a button that validates data on the GUI. These two buttons are also different from a third button that simply moves to the next or previous screen. Each of these buttons should be related/defined to/at the right level of abstraction. This is further explained in the next example.

Take the example of a GUI for registration to a conference. The end-user needs to fill registration information and then pay the fees. Registration information include the user's personal information, registration type (regular, student or discounted fees), additional information if exists, and billing information. The goal of the user from using the GUI is: Register for a conference. This goal is further refined at the task level by performing two tasks: "Fill registration information" and "Pay conference fees". The task level should identify task input elements, which are in this case, input elements each completes the related task. In figure 1, we present the "Finalize Order" task input element that completes the first task.

On the semantic level (figure 1): For conciseness, we only refine the task "Fill registration information". At this level, we need to define detailed functions to carry out the task. These detailed functions in turn identify all UI elements needed to carry out the task.

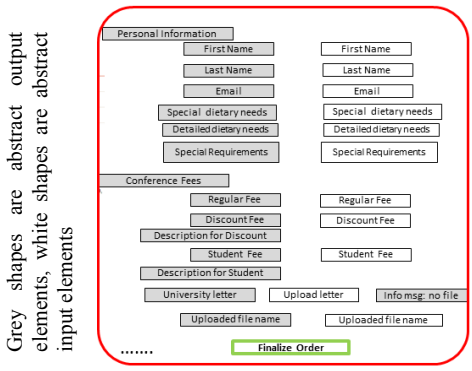


Figure 1. Outcome on the semantic level. "Finalize Order" is a task input element.

On the syntax-time: we define a correct distribution on time (that respects environment constraints). We may have two styles: (1) Display UI elements at the same time. (2) Define navigation as in figure 2.

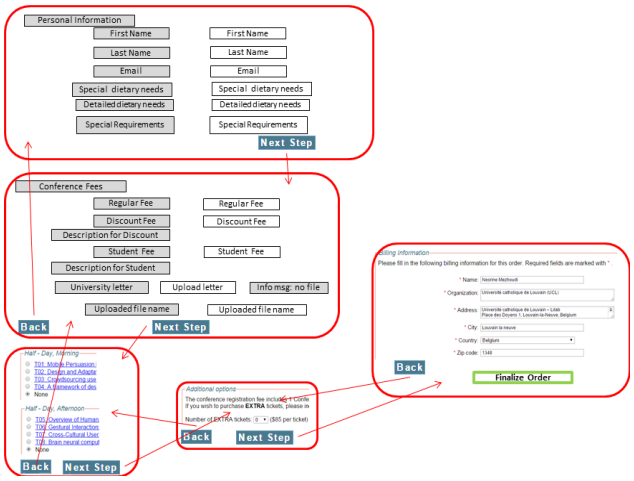


Figure 2. Distribute on time containers and define navigation elements for parts of the GUI.

On the syntax-space (figure 3): we refine only the syntax-time style 1 for conciseness: place elements on screen. Figure 3 depicts only the output of this level for the first time container: Personal Information according to syntax-time style 2 in figure 2.

On the widget level (figure 4): map UI elements to concrete GUI widgets. Finally, on widget Properties level (figure 5): Setting properties of widgets to get the final GUI.

On the final GUI, we note that every element is related to the level of abstraction that defines it. The "next step" button in figure 5 is defined at the syntax-time level, while other input elements (concretized as text boxes) are defined at the semantic level. The reader can foresee that the input element "Finalize Order" (see figure 2) shall be concretized on the screen as a widget (like a button).

Please notice that upper levels may impose constraints on the choices on lower levels (like selection of widgets).

Finally, the linguistic perspective enhances maintainability by enhancing sub-qualities as demonstrated in table 3.

Figure 3. Placement of elements on the screen.

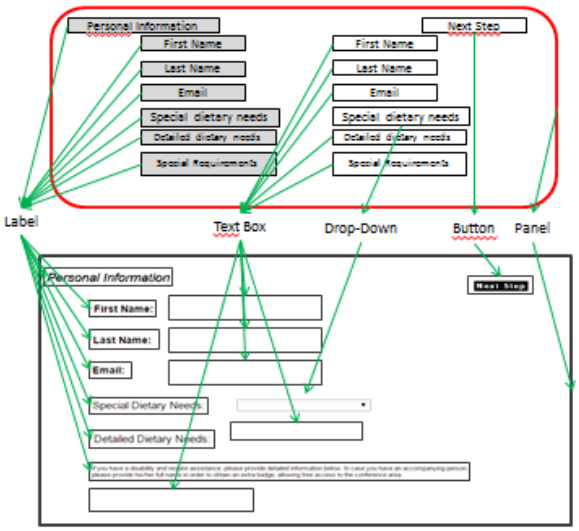


Figure 4. Mapping UI elements with concrete GUI widgets.

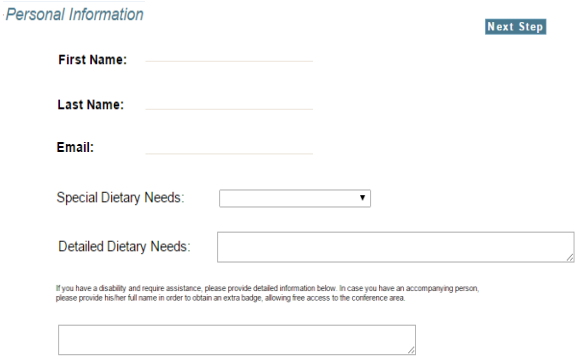


Figure 5. Setting widgets properties and the final GUI.

A Linguistic task model's requirements

The linguistic modeling approach aims at abstracting each linguistic level. The promise is to add the benefits of modeling approaches to maintainability enhancements in the linguistic perspective. The first step towards a linguistic model is to abstract the goal and task levels.

From the previous background on the linguistic perspective, we elicit the following requirements for a linguistic task model:

1- Well-defined criteria to separate between the goal and the task levels/models.

2- Well-defined criteria to separate between the task and the semantic model/level.

3- A notation that supports identifying task input elements.

Support for ISO-25010:2011 maintainability sub-qualities
Modularity: The perspective consists of 7 levels (modules), with defined interfacing between them. Changing a level will not affect the levels above, and may change impose changes to lower levels if the interfacing part is affected (replace a widget by another modifies only the widgets level and lower).
Reusability: If we want to create a GUI for registration to another conference that follows the same task model and provides same semantic but with a different GUI style, the task and semantic levels can be imported directly to the new system.
Analyzability: Every element at a level can be traced to upper level. For instance, a widget can be traced to the detailed function and task that needs it.
Modifiability: The perspective can classify a change based on required activities (modification) per level. Modifiability can be addresses per level instead of the complete GUI.
Testability: modifying a level doesn't affect upper levels. No testing for upper levels is required.

Table 3. The linguistic perspective and maintainability.

Our proposed task model follows the hierarchical task analysis approach. Unfortunately, hierarchical task analysis couples goals and tasks in an inseparable way. The hierarchy of tasks is a hierarchy of goals and sub-goals. This is further discussed in the next section. Anyway, we do not know of other approaches that separate goals and tasks clearly. Thus, the first requirement to separate between goals and tasks is impossible to satisfy.

The second requirement is fulfilled by reviewing the literature on task models. We investigate how reviewed task models identify tasks and adopt the most appropriate criteria. Lastly, the third requirement is addressed by introducing a task modeling notation that enables automatic identification of task input elements.

A REVIEW OF EXISTING TASK MODELS

The purpose of this literature review is to demonstrate that existing task models do not fulfill the linguistic task model requirements in section 2. This would justify why we need yet another task model. We do not provide an exhaustive analysis of all existing task models. We analyze a representative set of models only.

The models analyzed are: HTA [5], GOMS [8] , CTT [9] and K-MAD [14] and HAMSTERS [11]. Another analyzed model, the Guerrero model [10], employs task models in designing user interfaces for workflows.

Goals and tasks

In the late 1960s, Annett and Duncan offered a means to describing system in terms of goals and sub-goals, with feed-back loops in nested hierarchies [6]. This developed later into the specification of HTA (Hierarchical Task Analysis). The theory is based on a goal-directed behavior where we identify sub-goals in a hierarchy linked by plans. Plans describe how to perform sub-goal and determine conditions to trigger a sub-goal. The performance towards a goal can be achieved at multiple levels of analysis.

HTA is the bases of all existing task models. The theory is based on goals decomposition. Sub-goals are goals in turn. The hierarchy of tasks is a hierarchy of goals. Thus, separation between goals and tasks is impossible.

GOMS (Goals, Operators, Methods and Selection Rules) defines the concepts goals and tasks. Goals in GOMS are only high-level goals that are decomposed into tasks or sub-goals. The goal concept in GOMS is not meant to separate goals from tasks. GOMS describes the procedural knowledge in order for a user to carry out tasks on a device or system and is intended to use after a complete hierarchical task analysis.

In what concerns the linguistic perspective, hierarchical task analysis approaches can't separate tasks from goals and thus a task model will inherit the limitation of identifying what is a task and what is a goal. Anyway, we consider the main goal at the goal level, and the decomposition of this goal at the task level.

Task decomposition stopping Criteria

This is one of the critical aspects in HTA. The hierarchical decomposition stopping criterion in HTA is determined through the probability of failure (P) multiplied by the cost of failure (C) [7]. When the estimation of this formula is acceptable, the analyst can stop the decomposition. Although this formula is simple enough, applicability is hard.

GOMS does not introduce any changes to the HTA theory. It can be established after the complete HTA analysis. Thus it also inherits the problem of defining the decomposition stopping criteria. The pragmatic approach provided in GOMS is based on judgment calls [8]. The analyst needs to decide when to stop relying on a psychological theory or model for how people do the work. GOMS defines the following concepts: goals, tasks, methods, operators and selection rules. Methods in GOMS are a sequence of operators to carry out a task. In the linguistic perspective, GOMS methods fit at the semantic level.

Identification of tasks in CTT is based on identification of activities in the scenario. Tasks in the hierarchy can be added to represent a semantic grouping of identified activity tasks. Anyway, the decomposition may continue and stop at the granularity of identifying needed user input element. In the latter case, from a linguist perspective, CTT may identify non-task input elements, which should be identified at the semantic level. It may also identify navigation elements (like pagination on a grid of flights), which should be identified at the syntax-time linguistic level.

K-MAD is a hierarchical model of tasks from the most general one (root) to the most detailed ones (elementary actions) [14]. The stopping criterion is the elementary action which in the linguistic perspective might be at the semantic or even lower level.

Guerrero's model [10] addresses developing user interfaces to workflow systems. The model contains three main entities: the workflow, the process and the task. The workflow is related to a hierarchy of processes. Leaves in the process hierarchy represent tasks. CTT model is employed to model tasks and model the UI. In that, Guerrero's model differs from CTT in the employment of CTT in the UI design as a sub activity in the workflow development.

The interest of this model to this paper is the definition of a set of task identification criteria. These criteria identify root tasks that should exist in the process model. The root tasks are further refined using CTT to design the UI. Task identification criteria are based on changes on the work environment as described in the scenario. These criteria are:

- Change of space: when the location of operations changes.
- Change of resource: when the scenario suggests that new or different resources are exploited. Resources are of types: user, material and immaterial
- Change of time: when the scenario indicates a different time period in which the task is performed. Three type exist: interruption, waiting point (decision or accumulation) and permanence of execution unit (synchronization point).
- Change of nature: tasks can have the following natures: manual, automatic, interactive or mechanical.

Task models that aim at generating the final user interface cover all linguistic levels. Thus, Guerrero's models can not be adopted as a linguistic task model. Anyway, the process model in Guerrero's model is at the task linguistic level while the task model in Guerrero's fits all the other linguistic levels. Thus, Guerrero's task identification criteria can be employed in the linguistic task model in order to identify leaf tasks.

Task notation and identification of task input elements

The widely used notation in tasks models is the CTT notations. Although several variations for this notation exist to enhance expressiveness power of the task model or mapping with system models, like HAMSTER [11], they all use temporal relations defined in CTT.

Although powerful, CTT temporal relations are not enough to identify task input elements in the linguistic perspective. We show a scenario on this limitation.

In order for a user to search for a flight, s/he fills in search parameters. The system searches for relevant flights and displays them on the screen. The user selects the preferred flight. This scenario can be modeled using CTT notation as in figure 6.

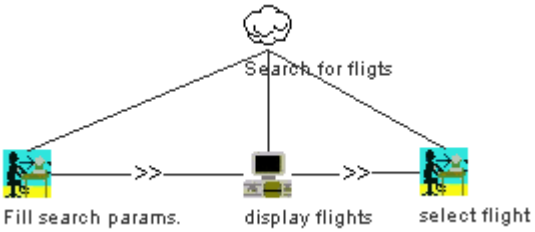


Figure 6: Search for flight using CTT notation

The tasks "display flights" and "select flight" in figure 6 might be implemented in the GUI as a table of flights with a "select" button on each row to select the flight. This select button is a task input element because it completes the task "select flight". The number of task input elements is equal to the number of flights in the search result. The linguistic perspective requires identifying all task input elements, provided that we know the number of flights to display. The notation doesn't reflect that **the select flight task will produce a number of task input elements related to the number of flights displayed**. This *dynamic aspect* in task execution is important in the linguistic perspective.

Note that another style is to display one flight and a select button at a time. This style differs from the above at the syntax-time level: defining time containers and navigation. Even in this style, we need to identify all produced task input elements.

K-MAD notation provides a solution to this dynamic aspect, but the notation introduced employs calling functions on the data model. From a linguistic perspective, these functions are at the semantic level. If such functions are to be used on the task level, they should take the form of a service to be implemented on the semantic level. Klug [15] introduced a task state and ports on tasks to exchange data to turn CTT model into executable task model. Data ports from the linguistic perspective are at the semantic level. Anyway, task state diagram is interesting and at the linguistic task level.

In the next section, we introduce a linguistic task model notation to overcome this limitation.

A LINGUISTIC TASK MODEL

The need for a new task model notation is justified by the linguistic requirement to identify task input elements. Task analysis and identification of task are already addressed in the literature and we adopted task identification criteria from Guerrero model. The task hierarchy is established by logically grouping identified tasks (like in CTT). A task can have one of the five categories: user, interactive, system, mechanical and abstract. These categories encompass categories in CTT and in Guerrero's model. The change we introduce in our

task model is a notation that supports identifying task input elements.

Identification of user input elements is based on the task category. User tasks are not considered in the GUI, because they are manual tasks. Thus no input elements are needed for these tasks. System tasks are performed by the system and require not interaction from the user to be started or completed. The same applies to mechanical tasks that are performed by machines (not computers), so they have no impact on the interaction. Abstract tasks are logical tasks in the hierarchy and do not have effect on identifying task input elements (they group input elements from children tasks). The only interesting category that affects identifying task input elements is the interactive category as they are performed by the user using the GUI.

An interactive task needs an input element so the user can denote the task as completed. Other task input elements might be needed to give the user control over the performance of the task. For instance, can the user rollback a completed task? This is important if the user can change his mind after completing the task (an example is selecting a flight and then deciding to select another).

The task-state diagram

Determining needed task input elements for a task depends on the task state. The user needs a complete input element on the GUI to complete the task in hand. If the task can be suspended, a suspend input is needed. Once it is suspended, we may need to resume it explicitly using a different task input element. Our task model defines a configurable task state diagram for tasks. Each task can configure its diagram. Identification of task input elements for an interactive task depends on its configured state diagram. The generic task state diagram is depicted in figure 7. The states are explained in table 4.

Transitions in a task's state diagram from state A to state B is defined as a condition. When the condition is satisfied, the transition occurs. There are three types of conditions:

1- Automatic: the condition is always true and the transition is automatic
2- User: the condition is true when the user provides a required explicit input.
3- Other Condition (or Condition for short): any other boolean expression. When this condition is true, the transition is performed.

A transition type is the type of its condition. Transitions for system tasks can be of type "Automatic" or "Condition". User and mechanical tasks has no impact on the GUI. Anyway, for consistency, a state diagram with "Automatic" transitions is attached to them. Transitions for interactive tasks are richer.

Employing task states can be noticed in other researches, like in [16], where researchers enrich CTT by adding states to tasks to support run-time execution of the task mode. Task states in our model enable identifying task input elements. Besides, researchers in [16] use CTT temporal relations to define transitions among tasks, while states in our model play a major role in defining these transitions (in the coming section).

Properties to configure an interactive task state diagram

Every interactive task must define the condition for the complete transition state. This condition must be of type "User" or "Condition", with use as a default. Setting the condition type to "User" means the task requires an input element on the GUI to allow the user to complete it. Setting the condition type to "Condition" means the task depends on the state of another task(s) to complete. We elaborate more on this case after introducing relations between tasks.

State	Type
Created	The task is created by the system. Task initialization can happen here.
Offered	The system offers the task to the user to start
Started	The task is in the course of running.
Suspended	The task is in the course of running. For interactive tasks, an explicit input from the user is needed.
Completed	The task has completed execution and the sub-goal is satisfied.
Destroyed	Resources reserved by the task can be released
Errored	Something prevents running the task.

Table 4. Definition of task states

An interactive task sets, by default, all transitions to "Automatic", with the exception for the complete transition. Transition types can be configured by setting the task properties. These properties and their impact on the state diagram are discussed in the following.

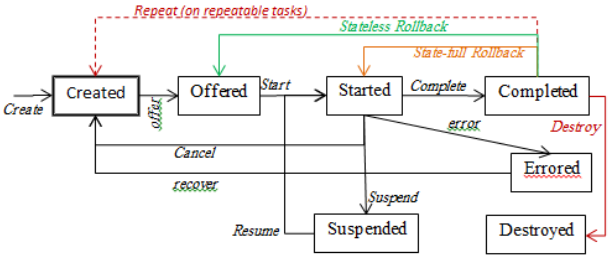


Figure 7. The Task State Diagram and transitions.

Assume the user performing a task would like to withdraw already filed in information and restart from the beginning. Such tasks are cancellable tasks. Setting the canCancel task's property to true sets the cancel transition type to "User". This impacts the GUI by adding a task input element to cancel the task. Setting the canCancel property to false removes the cancel transition from the task's state diagram.

Assume the user can rollback a task. If the task can be rolled back, a task input element to rollback is needed on

the GUI. An example is when the user selects a flight then changes her/his mind. Not every task can be rolled back. A payment for a flight task can't be rolled back by simply changing the task state. Rolling back a payment usually requires a different process. Rolling back a task can be of two types: (1) Stateless: re-create the task and discard all previous task information. (2) State-full: restore the task information previously entered before completion. The task property canRollBack can have one of three values: false, stateless and statefull. The value of this property controls the rollback transition. The false value removes the rollback transition. The stateless and statefull values set the rollback translation type to user, although it can be modified to a condition (a relation to another task).

The property "mayError" denotes a task might be prevented from execution due to various circumstances. If these circumstances are detectable by the system, we can define a condition on the error transition. If these circumstances are not detectable by the system, the user should be given this means to move the task to the error state by a task input element on the GUI. The "mayError" property can have one of the values: false (removes the error transition), condition (define a condition transition) and User (define a "User" error transition). Another related property is the "canRecover" which denotes how to recover from the error if possible. It accepts the same values of the property "mayError".

The property "canSuspend" on a task controls the suspend transition in a similar way to canCancel property. The resume transition is influenced by this property. A task might be suspended by the user action, but resumed based on the state of another task. In this case, the suspend transition is set to user but the resume transition is a condition. The resume transition initial type is set to user when the suspend transition type is set to user.

The destroy transition is a special case and is defined when a task is dynamically created by another. Dynamic tasks are discussed later in this section. For dynamic tasks, destroy transition can have one of two types: user or condition. Offer and start are condition transitions. The user can't control to start a task by an input. The system offers and starts the task once conditions are satisfied. The create transition is a special transition that is related to dynamic tasks (discussed later section).

Tasks relations

Relations between tasks control the task execution sequence towards attaining the goal of the parent task. Relations take the form of ECA (Event, Condition and Action) rules:

Event	ON TS.State	TS is the source task
Condition	TD.State= "value"	TD is the destination task
Action	TD.Transition	

An example on relations is:

```
ON Fill_Flight_Search_Info.Completed
        If (Display_Flight.State="Created")
                Display_Flight.offer
```

Because transitions are un-ambiguously defined in the task's state diagram, the condition part can be automatically verified when executed in the condition. Thus, we can omit the condition part from the relation.

The action part is ensured to be executed in the relation, but the destination task may not change its state. Changing the destination task state depends on the transition condition. If the transition's condition is evaluated to true, the state is changed.

Task relations can be seen as adding an "AND" condition (the event and condition) to the task's transition defined in action part. In the example above, the task Display_Flights changes its state to "Started" if the offer transition condition is evaluated to true and the Fill_Flight_Search_Info state is "Completed".

Relations can be defined on sibling tasks in the hierarchy. Children tasks can also be related to direct parents. An example on defining relations between tasks is depicted in figure 8. In this figure, we note that a child task has a relation with the parent to complete it upon completion.

Optional tasks

Optional tasks are tasks that always look to parent tasks as completed, although their state diagram indicates a different state. An optional task has the property "isOptional" set to true.

Task Repetition

Repetition can be implemented on a task T using relations like: On T.Completed, T.offer. Anyway, this can't be executed at run time, because the "offer" transition is not possible from the "Completed" state. The rule should be updated to: On T.Completed, T.create. The latter rule means a new instance of the task T is created once the task T is completed. We call the newly created task instance a dynamic task.

A dynamic task is a task that is created by another task at run-time. Creation of these tasks (repetition decision) could be initiated by an explicit action from the user or by the system. The need for a decision for repetition justifies the need for a different task (the justification for identifying this task is a change of time::decision point). Thus, to repeat a task "T", we need a task for the decision "R". The relation between these tasks to enable repetition can be defined like: On R.Completed, T.create. R is called the pumping task (it pumps dynamic tasks in the model).

If the repetition decision is taken by the user, the pumping task must be interactive and dynamic. If the repetition decision is taken by the system, the pumping task needs to have the property "repeatable" set to "true". This is important because it has implications on the state diagram: an automatic transition is defined from the

completed state to the created state. The repetition condition in a system pumping task is the condition of its offer transition.

Demonstration and implementation

Building a convincing example on using the notation would take a large space. A pragmatic, and still convincing, technique is to demonstrate expressiveness of the approach in comparison to other notations.

The comparison with CTT notation is demonstrated in figure 8. This figure shows how temporal relations can be represented using our notation. Thus, the notation introduced in the linguistic task model is at least equivalent to CTT notation.

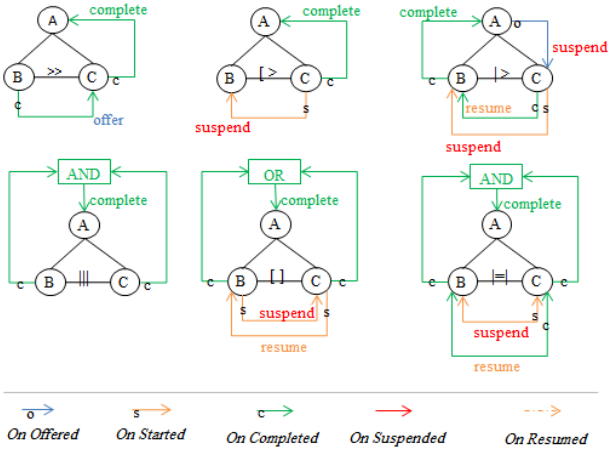


Figure 8. Representation of CTT temporal relations using the linguistic task model notation.

Id	Parent	Task	Justification	Configuration
1	/	Search for a flight	Grouping	Abstract, Stateless rollback,
2	1	Fill Search params.	-	Interactive, State-full rollback, canCancel=true
3	1	Display Flights	Grouping	Abstract, Stateless rollback,
4	3	Repeat on flights	Repetition.	System, Stateless rollback, repetitionTask=true
5	3	Display a Flight	From 2: Change in nature: interactive->system	System, Stateless rollback, canCancel=false
6	3	Select the Flight	From 4: Decision point	Interactive, Stateless rollback, canCancel=false

Table 5. Decomposition of the goal "Search for a flight".

The linguistic task model notation addresses also the dynamic aspect of task execution. We give an example on this **dynamic aspect**, which is **not supported** in CTT notation to the same level of details (see explanation on this limitation on page 6, column 2).

The example we present has the goal "Search for a flight". This goal is decomposed into the following tasks with justifications and configurations in table 5. The scenario works as follows: *The user fills in search parameters. The system searches for relevant flights and displays them on the screen. The user selects the preferred task.*

The task "Display Flights" is a repetitive task, because for each flight the system displays, the user has the choice to select it or select another one. Thus, the notation enforces adding a system pumping task that pumps the tasks "Display a Flight" and "Select the flight". That pumping task repetition condition is the number of flights in the search result. Table 5 shows tasks, their justification and configuration values.

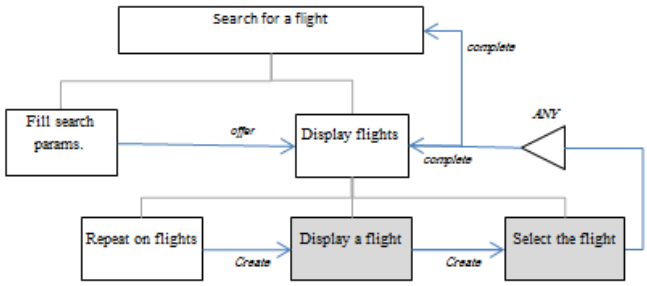


Figure 9. Search for flight example.

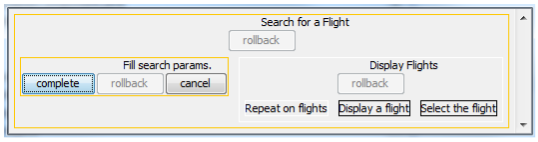
From a linguistic perspective, Figure 9 provides the means to identify task input elements. Each dynamic interactive task "select the flight" requires a task input element for completion. The total identified task input elements at run-time is equivalent to the number of flights in the search result. Besides, other task input elements can be identified for each task from its state diagram and transitions. Every "User" transition requires a task input element to enable the transition.

The current implementation for designing the task model uses a java API. A graphical designer is still under work. The Java API allows creating tasks and configuring properties. It also allows defining relations between tasks. The critical and essential part in the implementation is the run-time simulator for a linguistic task model that generates task input elements. This simulator is already implemented and tested.

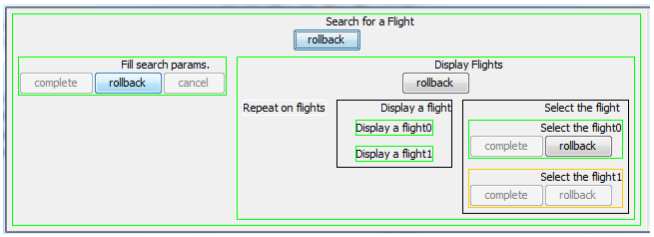
Figure 10 shows the result of executing the task model in the simulator at two moments: the first is at the start and the second after the selection of a flight. Every task is displayed in a rectangle. The border color of the rectangle reflects the task state (see color caption in figure 8). Grey color denotes the "Created" state (the Repeat on flights task), while "Black" color denotes undefined state (the case of a dynamic task).

Figure 10 demonstrates the creation of dynamic tasks after completing the "Fill search param." task. Tasks are refined at the semantic level by defining detailed functions that define how to carry out the task. This includes the needed non-task input elements for every task, and also transition conditions for the system tasks.

The repetition condition in the model must be realized at the semantic level based on the search result. In the simulator on the linguistic task level, system repetition conditions are realized as repetition for two times only. This is why we see only two instances of the dynamic tasks “Display a flight” and “select the flight”.



Starting execution of the task model



After selecting a flight

Figure 10. Executing the “search for a flight” task model in the simulator.

Figure 10 also demonstrated the identification of task input elements for each task. For example, we can see four rollback elements. Each rollback has a different purpose on the GUI. The rollback on the “select flight0” task is to allow selecting a different flight. The rollback on the “Display Flight” task is to allow re-executing children, which includes: re-execute the query and re-create dynamic tasks. It might look like a refresh button. The rollback on the “Fill search params.” allows modifying current search parameters (stateful rollback). The rollback on the root task is to rollback everything including current search parameters. The GUI designer at a later stage can decide how to manage these different rollback, but he already knows that ignoring any is limiting a functionality in the model.

CONCLUSION AND FUTURE WORKS

In this paper, we presented a linguistic task model and notation. The presented linguistic task model clearly separates the task and the semantic levels by adopting a well-defined set of task identification criteria proposed by Guerrero. We also discussed the impossibility to separate goals from tasks in hierarchical task approaches.

A new task modeling notation is introduced. This notation enables identification of task input elements. This identification is based on the task state diagram that is configured on each task.

The notation also addressed the dynamic aspect of modeling by introducing dynamic tasks and pumping tasks. It also presented relations on tasks based on ECA rules on tasks states. Anyway, further research is needed to enhance the understandability of the notation.

The task model is the first brick towards modeling GUI from a linguistic perspective. In the future, we will keep working on modeling other linguistic levels. Tool support should be worked on too. We needs to work on creating a tool to support the notation introduced.

BIBLIOGRAPHIE

1. ISO/IEC, “ISO/IEC 25010 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models,” International Organization for Standardization, Tech. Rep., 2010.
2. Khaddam, I., Mezhoudi, N., Vanderdonckt , J. A Linguistic Perspective to Develop Graphical User Interfaces. 3ed International Conference on Control, Engineering & Information Technology (CEIT). Tlemcen, Al-geria, 25-27 May, 2015 (accepted).
3. Nielsen, J. A. Virtual Protocol Model for Computer-Human Interaction. International Journal of Man-Machine Studies, March 1986, Vol.24, Is-sue.3 , pp. 301-312.
4. http://www.w3.org/TR/mbui-intro/
5. Annett, J. (2003). Hierarchical task analysis. Handbook of cognitive task design, 17-35.
6. Annett, J., Duncan, K.D., 1967. Task analysis and training design.
7. Annett, J., Duncan, K.D., Stammers, R.B., Gray, M.J., 1971. Task Analysis. Department of Employment Training Information paper 6. HMSO, London.
8. Kieras, D. GOMS Models for Task Analysis. The Handbook of Task Analysis for Human-Computer Interaction, Ed. Dan Diaper, Neville A.Stanton, Lawrence Erlbaum Associates, pp. 83-116. 2004.
9. Paternò , F., Model-Based Design and Evaluation of Interactive Applications. Applied Computing, Springer-Verlag London , 2000
10. Guerrero, J. G., Vanderdonckt, J., Lemaigre, G. Identification Criteria in Task Modeling. Human-Computer Interaction Symposium IFIP International Federation for Information Processing Volume 272, 2008, pp 7-20
11. Barboni, E., Ladry, J., Navarre, D., Palanque, P., Winckler, M.: Beyond modelling: an integrated environment supporting co-execution of tasks and systems models. In: Proceedings of the 2nd ACM SIGCHI Symposium on Engineering interactive Computing Systems. EICS 2010, pp. 165–174. ACM, New York (2010).
12. Khaddam, I., Mezhoudi, N., Vanderdonckt , J. Towards a Linguistic Modeling of Graphical User Interfaces: Eliciting Modeling Requirements. 3ed International Conference on Control, Engineering & Information Technology (CEIT). Tlemcen, Al-geria, 25-27 May, 2015.
13. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., & Vanderdonckt, J. (2003). A Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computers , 15(3): 289–308.
14. Caffiau, S., et al. Increasing the expressive power of task analysis: Systematic comparison and empirical assessment of tool-supported task models. Interact. Comput. (2010), doi:10.1016/j.intcom.2010.06.003.
15. Klug, T., Kangasharju, J. Executable Task Models. TAMODIA’05, September 26–27, 2005, Gdansk, Poland. Copyright 2005 ACM 1-59593-220-8/00.
16. Gharsellaoui, A., Bellik, Y., Jacquet, C. Un système d'aide et de suivi des tâches utilisateur dans un environnement ambiant. IHM 2014, Oct 2014, Lille, France. pp.130-138, Actes de la 26e conférence francophone sur l'Interaction Homme-Machine. <hal-01080244>