Towards Web Browsing Assistance Using Task Modeling Based on Observed Usages

Benoît Encelle[1](✉) and Karim Sehaba[2]

[1] Université de Lyon, CNRS, Université de Lyon 1. LIRIS, UMR5205, 69622 Lyon, France
benoit.encelle@liris.cnrs.fr

[2] Université de Lyon, CNRS, Université de Lyon 2. LIRIS, UMR5205, 69676 Lyon, France
karim.sehaba@liris.cnrs.fr

Abstract. This article deals with knowledge extracted from observed usages of Web sites/applications for assistance purposes. The extracted knowledge is used to develop assistance systems in order to help a) users in carrying out Web browsing tasks, or b) designers to adapt/redesign Web applications. The suggested approach involves the generation of task models from interaction traces, which are then used to perform assistance. Task metamodel characteristics for assistance purposes are firstly identified and then used to develop a comparative study of some well-known task metamodels, resulting in the selection of the ConcurTaskTrees (CTT) metamodel. In order to generate CTT task models, a set of algorithms that identify CTT operators from interaction traces - represented as deterministic finite state automata - are presented. We also expose an approach for performing assistance, for users and designers, based on task models and finally conducted unit testing and validation based on two real web browsing scenarios.

Keywords: Knowledge extraction · Interaction traces · Task model · ConcurTaskTrees (CTT) · Web browsing/redesign assistance

1 Introduction

This article is an extended version of [1] and deals with knowledge extraction from observed usages. This work is situated in the field of Web browsing assistance/Web automation, and Web application redesign assistance. The question we address is how to develop a system able to assist Web users in carrying out Web browsing tasks and Web designers in the adaptation and/or the redesign of Web applications.

In many current assistance systems, the helps typically provided and, on the whole, the assistance knowledge, are predefined during the design phase and correspond to the uses envisaged by the designers. However, it is usually difficult to anticipate the entire spectrum of the real uses for a given system, and for an online application more particularly. Indeed, the Web users can have very diverse profiles, their requirements can be in continuous evolution, they may have various conditions of use, etc. Even if tools and methodologies exist to predict particular uses (including requirements analysis, rapid prototyping and assessments in ecological situation), these will remain intended uses and could sometimes not cover/correspond to all real uses. This may be due to several difficulties related to: the analysis of all the contexts of use, the representativeness of the observed sample of users, the achievement of truly ecological conditions in assessments, etc. Thus, the design of an assistance system with a complete representation of the needs of upcoming users and their evolution is extremely difficult, if not impossible.

To solve these kinds of difficulties, we propose to base the assistance on the real uses, observed right after system deployment. The approach developed in this work aims to produce task models based on these real uses. As an input in the task model generation process, we start from interaction traces (i.e. logs). Here, a trace represents the history of the actions of a given user on a Web application. At the output, a task model represents the different possibilities of performing a given task. Formally, a task model is a graphical or textual representation resulting from an analysis process, making it possible to logically describe the activities to be carried out by one or more users to achieve a given objective, such as booking a flight or hotel room.

The task models obtained by the proposed process will then be used, in an assistance system, to guide the users in the accomplishment of their tasks and the designers in the analysis of their applications, to carry out redesigns of them.

In this article, we studied the characteristics that should be supported by task models to ensure that they can be used for providing assistance. Two main properties have been identified, namely:

1. The intelligibility of the task model for the user; and
2. The expressiveness of the task model and its ability to be manipulated by a software, more specifically an assistance system.

Several metamodels have been suggested to represent task models in the literature. Therefore, we challenge these metamodels with the previously identified characteristics to identify the most suitable ones for our assistance need. This study leads us to choose the ConcurTaskTrees (CTT) metamodel. Then, we establish a process for generating task models, representing real uses, based on interaction traces. Finally, we then suggest an approach for providing Web task assistance based on these task models.

In summary, the work presented in this paper aims at four contributions:

1. The specification of the characteristics of the task metamodels for assistance purposes;
2. The confrontation of existing metamodels regarding the characteristics previously identified (cf. 1). The study we conducted allowed us to choose the CTT metamodel;
3. A process for generating CTT task models from interaction traces;
4. An assistance approach based on these task models.

This article is organized as follows. Section 2 presents a state of the art on web browsing assistance based on traces and task models. Section 3 details the target characteristics of a task metamodel for assistance purposes. Section 4 presents a comparison of existing task metamodels against our target characteristics, leading to the choice of the CTT metamodel that we present next. Section 5 describes our approaches for a) generating task model from interaction traces and b) providing assistance based upon these task models. Section 6 is dedicated to the validation process and results. Finally, we conclude and state some perspectives in Sect. 7.

This article, in comparison with [1] mainly a) gives more details on the process for generating CTT task models from interaction traces (third contribution) and b) provides an approach for task model-based assistance (fourth contribution).

2 Related Works

The design and/or the generation of task models has been the subject of several studies in the literature. In this work, we are interested in generating task models from traces. In this context, the methods proposed in the literature can be classified into two main approaches, namely: the generation of task models from a) one task instance and b) from multiple task instances.

2.1 From an Instance to a Task Model

The first approach is to start from one instance (i.e. a particular way of performing a task) to generate a task model. In this way, let us mention the CoScripter system [2] which makes it possible to automate web tasks via a scripting language, used among other in the Trailblazer Web assistant [3].

```
1. goto "http://www.mycompany.com/timecard/"
2. enter "8" into the "Hours worked" textbox
3. click the "Submit" button
4. click the "Verify" button.
```

Fig. 1. Example of a CoScript (from [3]).

Figure 1 shows an example of a CoScript representing the actions for performing a "book purchase" task on Amazon. As this example shows, a CoScript is very close to a trace and possibly generalizes it to a minimum. Indeed, the scripting language integrates only one element of generalization (with the notion of personal database [2]). Therefore, a CoScript is more an instance than a task model itself. In addition, control structures such as conditional (e.g. if A then B else C) or iterative ones do not exist in CoScripter [2]. Therefore, to generate a task model from a CoScript, the challenges of a generalization process from an instance remain open as pointed out in [4].

To answer these challenges, namely the elicitation of a task model from an instance, the approach used in PLOW (Procedural Learning on the Web) [4] increases the description of a task instance by knowledge provided during its performance. This knowledge makes it possible to represent conditions, iterations, etc. This "expert" knowledge is, within the framework of PLOW, provided by a certain kind of user named demonstrator. The latter teaches the system new task models by providing this expert knowledge orally, while he/she is performing the task. This knowledge is then interpreted using Natural Language Processing technologies. This approach requires 1) that the demonstrator keeps in mind to bring a maximum of knowledge and 2) that the tool is able to correctly interpret this knowledge to modify the model accordingly. Thus, the more a demonstrator brings knowledge and the more a model can be generic. Overall, with respect to this knowledge, the more it is expressed in language similar to that used by the system to model the task, the less likely it is that it will be misinterpreted.

2.2 From Instances to a Task Model

The second approach attempts to eliminate the need of expert knowledge by using multiple instances to generate a task model, as in LiveAction [5]. The latter focuses on the identification and modeling of repetitive tasks, tasks being represented as CoScripts. In LiveAction, a task model is generated using a set of CoScripts and Machine Learning techniques. With this kind of approach, task models are represented as finite state automata (FSAs) [5, 6]. However, to our knowledge, an assistance system based on such automata has not been developed yet.

With this kind of approach, the level of genericity of the models obtained depends on the quantity of instances as well as their quality (variability in particular). It should be noted that, as suggested in [5] and with the aim of quickly reaching satisfactory models, users should be able to add knowledge to generated models by manipulating them directly.

2.3 Synthesis

The research works on generating task models from interaction traces can be classified into two distinct approaches. The *instance to model* approach produces more specific than generic task models. This approach requires a lot of expert knowledge and reaches its limits in a Web context with a large amount of "*open and massive*" applications. The approach *instances to model* requires a large number of different instances to generate sufficiently complete and generic models.

To fulfill our needs, we base our work on the generation of task models from several instances to reduce the expert knowledge that has to be initially provided to obtain sufficiently complete and generic models. In order to minimize the number of instances needed and the necessary variability among them, we will opt for "*user-friendly*" task models. By user-friendly task models, we mean task models that could be quickly and easily understandable and handleable by users (users can easily modify these models).

These modifications that add knowledge to a generated model, will have to be carried out directly by handling the model itself, thus evacuating the risks of misinterpretations which can lead to erroneous models (risks existing in PLOW for example). Therefore, reusing an approach such as the one described in LiveAction seems interesting to us, except that the generated models are represented by finite state automata (FSAs). These FSAs are indeed difficult to understand and manipulated by end users:

- There is no decomposition or hierarchical relationships between the tasks and their subtasks, which does not facilitate model reading and understanding.
- FSAs generally have a large number of states and transitions, even for representing simple tasks, which causes difficulties in reading and understanding models.

To remedy these problems, we argue for the use of task meta-models that are more user-friendly than FSAs. To achieve this objective, we will first identify all the characteristics that have to be supported by a metamodel dedicated to browsing assistance. Then, we will confront some well-known existing task metamodels with the characteristics identified in order to guide our choice. The metamodels studied were designed to be quickly understandable and handleable by "novice" (i.e. user-friendly metamodels).

3 Metamodel Characteristics Study

As mentioned above, task models generated from traces must be user-friendly, easily understandable, and even directly handleable by the user. In addition, these models must also be machine-friendly to be automatically used by an assistance system to guide the user in performing his/her task. This characteristic implies that a machine can a) be able to identify models that do not conform to the metamodels (i.e. models that can lead to misinterpretations) and b) understand and interpret these models at a certain level. This requires a certain level of formality, a precise semantics of the elements constituting these metamodels.

We complement these two main characteristics with other secondary characteristics. The first is related to the set of tasks we want to model and their intrinsic properties. We want to model web browsing tasks, essentially sequential tasks (actions to be carried out one after another) and single-user tasks (collective tasks are not considered). This characteristic therefore corresponds to the expressiveness of the metamodels and more specifically to their ability to represent browsing tasks. In this perspective, elements of a metamodel must make it possible to specify the component of the user interface to be manipulated. Another important thing is that the metamodels must also allow the expression of optionality (to model the fact that a sub-task is optional to carry out a given task), for a sufficiently generic modeling of Web browsing tasks. For example, the modeling of a "ticket reservation" task must be able to represent optional steps, such as the possibility of entering a discount card id.

The second secondary characteristic is related to the adaptability and extension capabilities of metamodels: these are initially designed to be used in a set of software engineering phases [7], but they must be adaptable to meet our assistance goal. However, these adaptations may require complements (e.g. concepts), i.e. new elements added to the metamodels. As a result, metamodels have to be extensible if necessary (extensibility characteristic).

The third is related to the plurality of devices that can be used to browse the Web (smartphone, tablet, computer, etc.) and their respective characteristics (screen sizes, proposed interaction modalities, etc.). Indeed, several variants of user interfaces or process of accomplishing tasks may exist for the same Web application ("responsive" aspect). These variations of the context of use must be able to be supported by the chosen metamodel: the same task has to be described in several ways according to the context of use.

To sum up, a "candidate" metamodel that could integrate the assistance process we wish to develop, must have key characteristics (user-friendly and machine-friendly) and secondary ones (expressivity, adaptability/extensibility, support variations in the context of use). The following section presents a study comparing existing metamodels with the above characteristics.

4 Task Models for Assistance Purposes

4.1 Comparison of Metamodel with Target Characteristics

Comparative studies of the most well-known metamodels have been proposed in the literature [7–9], including:

- HTA: Hierarchical Task Analysis,
- GOMS: Goals, Operators, Methods and Selection rules,
- CTT: Concur Task Trees,
- MAD: "Méthode Analytique de Description".

These studies suggest analysis frameworks to compare metamodels between them in order to guide the choice of one or more specific metamodels according to a given objective. These analyses are based on a set of characteristics, including those we target (see previous section).

Concerning the user-friendly aspect, the authors of [7] refer to it through the "*usability axis in communication*" and specify in particular that, in relation to textual or formal metamodels, the graphic metamodels are more suitable. The author of [10] also approve this position. For example, the highly textual GOMS metamodel is moderately user-friendly [7]. Being able to break down tasks into sub-tasks (decomposition characteristic [7]), how to break down tasks and thus describe the relationships between tasks and sub-tasks are also important. For example, a tree-like representation of tasks/sub-tasks appears intuitive [11], as in MAD or CTT, and offers several levels of detail, including the ability to unfold/fold branches of the tree. Similarly, the ability of the metamodel to allow the reuse of elements helps to minimize the number of elements present and improve readability.

Concerning the machine-friendly characteristic, the degree of formality of a metamodel is related to what must be generated [7]: a formal metamodel can be used for the automatic generation of code while a semi-formal model can be used to generate user documentation for example. The authors of [8] confirm the need for a certain level of formality of the metamodels to be machine-readable: for example, a plan in HTA described informally (textual descriptions) the logic of execution of the sub-tasks that make up a task, which can lead to interpretation ambiguities. On the contrary, CTT has a set of formal operators, based on the LOTOS language [12], to describe this same logic, which guarantees an unambiguous automatic interpretation.

Regarding the secondary characteristics, concerning the expressivity of the metamodels for Web browsing tasks, in addition to sequentiality and single-user criteria, models such as MAD or GOMS do not allow the expression of the optionality [7], unlike CTT. In addition, some models, such as HTA for example, are not intended to indicate the user interface components that must be manipulated to perform the tasks/subtasks while others, such as CTT, allow it. We also have to mention that the W3C Working Group "*Model-Based User Interfaces*" has chosen CTT to model web tasks.

Concerning adaptability/extensibility, graphical metamodels are more easily extensible to express relationships or concepts that were not initially planned [7]. For example, several extensions have been proposed for models of this type, such as MAD or CTT (e.g. MAD*, CCTT).

Regarding the characteristic "context of use variations support", few metamodels integrate this dimension as underlined by [8]. Nevertheless, CTT integrates this dimension through the platform concept [10].

Table 1. Comparison of metamodels with target characteristics [1].

	User-friendly	Machine friendly	Expressivity	Adaptability/extensibility	Variations to the context of use
CTT	+	+	+	+	+
HTA	+	−	−	−	−
MAD	+	+	−	+	−
GOMS	−	+	−	−	−

Table 1 gives a summary of our confrontation of metamodels with regard to our target characteristics. It thus appears that the CTT metamodel, among the metamodels studied, is the only one that meets all the characteristics previously identified. The following section provides a short introduction to this model.

4.2 CTT Metamodel Overview

A Concur Task Tree (CTT) model exposes a hierarchical structure of tasks as a tree. Each tree node represents a task or a subtask. The node icon identifies the category of the task or subtask:

- cloud: abstract task, decomposable;
- user and keyboard: interaction task;
- computer: task performed by the system.

Logical or temporal relationships between tasks are indicated by operators. For example, the operator "[]" represents the choice and "|=|" represents the order independency operator. In Fig. 2, the room booking task can only be done if the user has specified the type of the room he wants to reserve (single or double). Thus, the "Make reservation" sub-task is only activated if the "Select room type" task has been performed.

These operators are formally defined (mainly from the LOTOS language [12]). CTT allows to add features to tasks as needed: iteration (indefinite or finite) and optionality. Finally, the tasks can be correlated to the application domain objects (the type of room for example) and to their representation(s) on a given user interface (for the selection of a type of room for instance, radio buttons or other).

5 Task Model Generation and Assistance

Remember that our goal is to propose a task model generation approach based on observed usages. These task models will assist users in performing web browsing tasks and the designers in the adaptation and redesign of web applications. To represent the tasks, we propose to use CTT and, for the observed usages, we rely on the traces/logs resulting from the actions of users on Web applications.

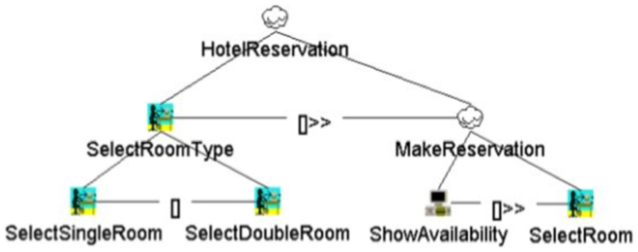


Fig. 2. CTT modeling of a room booking task [10].

Figure 3 presents the process workflow involved in our approach. At first, the logs/traces are transformed (T1) into a finite state automaton (FSA) or, more precisely, a deterministic finite state automaton (DFSA). We already mentioned that a bunch of work related to web browsing assistance generate FSAs from the traces, such as LiveAction. If these automata have the advantage of being relatively simple to process automatically (machine-friendly), they remain difficult to understand for the general public given: a) the large number of states and transitions they can contain, even for represent a simple browsing task, and b) their linear reading as there is no hierarchy of states to represent task/subtask relations. Therefore, the second step of our approach is to transform these automata into CTT models (T2).

In this section, we are interested in 1) the transformation T2 (FSAs -> CTT models), since T1 (Traces -> FSAs) has already been treated by other works - notably [5], and 2) the assistance process based on task models.

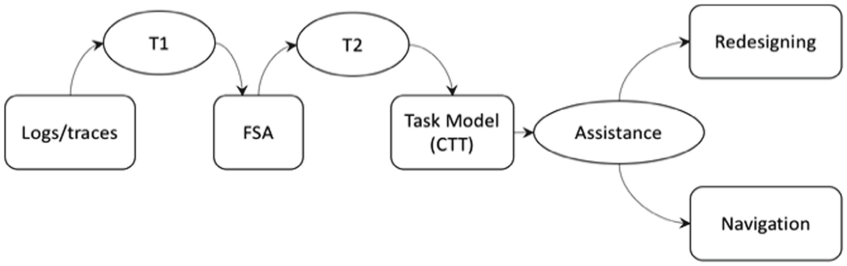


Fig. 3. Task model generation process and assistance [1].

5.1 Task Model Generation from Deterministic Finite State Automata

Formally, a deterministic finite state automaton (DFSA) is a 5-tuple (Q, Σ, R, qi, F) consisting of:

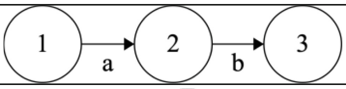
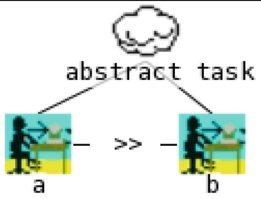
- Q: a finite set of states (Web resources/pages)
- Σ: an input alphabet, which in our case represents all the events applied to web resources (e.g. button click, typed text, etc.)
- R: a part of Q × Σ × Q called the set of transitions
- qi: the initial state. qi ∊ Q
- F: a part of Q called the set of final states

Since the automaton is deterministic, the relationship R is functional in the following sense: If (p, a, q) ∊ R and (p, a, q′) ∊ R then q = q′.

In the following subsections we detail the algorithms for converting DFSAs to CTT models, focusing on CTT operator identification on DFSAs.

Enabling Operators ("≫", "[]≫"). The CTT enabling operators (without "≫" or with information passing "[]≫") indicates that a subtask B cannot start until a subtask A is performed. From the DFSA point of view, if there is an *endState* state for which there is only one previous *startState* state, we can deduce that there is an enabling operator (see Table 2 for an example of conversion from DFSA to CTT).

Table 2. Example of an DFSA to CTT conversion, two enabling operators (between states 1, 2 and 2, 3).

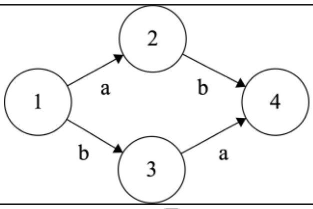
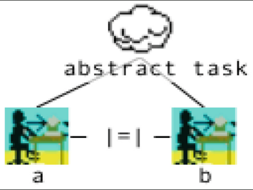
DFSA	
CTT	

DFSA to CTT model conversion algorithm (pseudocode) - enabling operator identification:

```
Input: State startState, State endState
Output: Boolean (true if enabling op. between startState
and endState, false otherwise)
If((endState.getPreviousStates.size>1) or
  (endState.getPreviousStates[0]!=startState))
  Return false
EndIf
Return true
```

Order Independency Operator ("|=|"). There is an order independency operator between two or more subtasks if they can be performed in any order. From the DFSA point of view, if there are for a couple of states (*startState*, *endState*) n paths (n > 1) that link them and that each of these paths has the same k transition labels (which then will be in a different order) and n = k!, then we can conclude that these are subtasks that can be performed in any order, expressed in CTT using independence operators (see Table 3 for an example of conversion from DFSA to CTT).

Table 3. Example of an DFSA to CTT conversion, one order independency operator (between states 1 and 4, two equivalent paths (through states 2 and 3)).

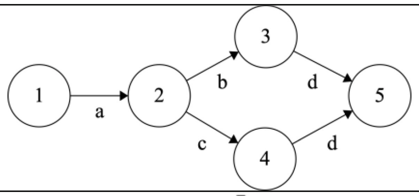
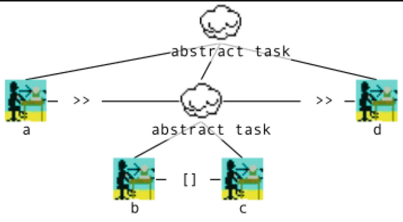
DFSA	
CTT	

DFSA to CTT model conversion algorithm (pseudocode) - order independency operator identification:

```
Input: State startState, State endState
Output: Boolean (true if order independancy op. exist,
false otherwise)
paths=getAllPossiblePaths(startState, endState)
If(paths.size<=1)
  Return false
EndIf
j=paths[0]
If(factorial(j.transitions.size)!= paths.size)
  Return false
EndIf
For i=1 to paths.size-1
  If(!hasSameTransitionLabels(j,paths[i]))
    Return false
  EndIf
EndFor
Return true
```

Choice operator ("[]"). The CTT choice operator indicates that a task T can be performed either performing subtask A or B. From the DFSA point of view, if - from a *startState* state to an *endState* state - there is two or more paths and at least two of these paths start with a different state, we can conclude that there is a choice operator (see Table 4 for an example of conversion from DFSA to CTT).

Table 4. Example of an DFSA to CTT conversion, choice operator (between states 2 and 5).

DFSA	
CTT	 abstract task a >> abstract task >> d b [] c

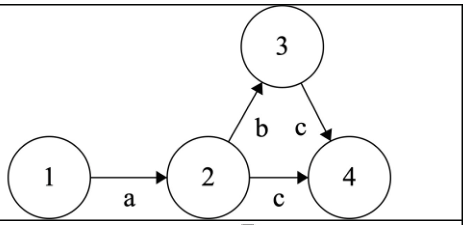
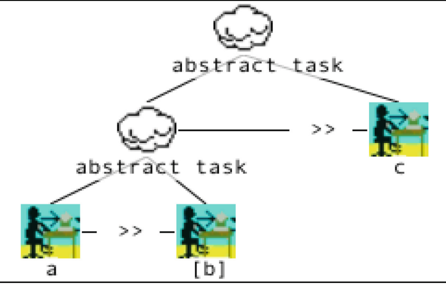
DFSA to CTT model conversion algorithm (pseudocode) - choice operator identification:

```
Output: Boolean(true if choice exist, false otherwise)
paths=getAllPossiblePaths(startState, endState)
If(paths.size<2)
  Return false
EndIf
ts=paths[0].getStates[0]
For i=1 to paths.size-1
  If(paths[i].getStates[0]!=ts)
    Return true
  EndIf
EndFor
Return false
```

Optional Subtask Detection ("[ST]"). Given a task T = A»[B]»C meaning that T is done by performing either subtasks (A, B and C), or (A and C).

B is an optional subtask that can or cannot be performed. To identify an optional task in an DFSA, all paths from a *startState* state to an *endState* state are collected before checking if each path last transition label is the same. After that, if a direct path (one transition) between *startState* and *endState* exists, we can conclude that there is (at least) an optional subtask (see Table 5 for an example of conversion from DFSA to CTT).

Table 5. Example of an DFSA to CTT conversion, one order independency operator (between states 1 and 4, two equivalent paths (through states 2 and 3).

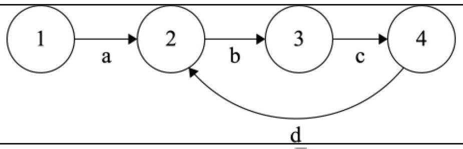
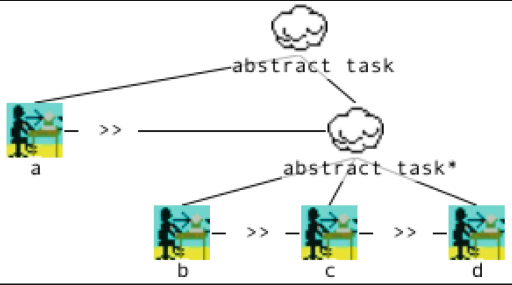
DFSA	
CTT	

DFSA to CTT model conversion algorithm (pseudocode) - optional subtask detection:

```
Input: State startState, State endState
Output: Boolean (true if optional task exists, false oth-
erwise)
paths=getAllPossiblePaths(startState, endState)
lastTransitionLabel=paths[0].getLastTransition().label
For i=1 to paths.size()-1
  If(paths.get(i).getLastTransition().label<>lastTransi-
tionLabel)
    Return false
  EndIf
EndFor
For i=0 to paths.size()
  If(paths[i].transitions.size=1)
    Return true
  EndIf
EndFor
Return false
```

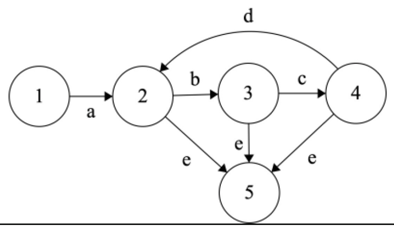
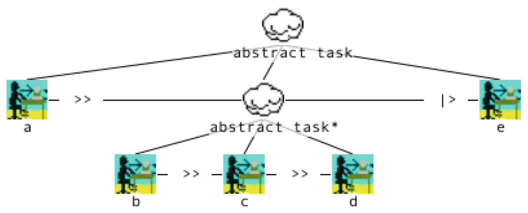
Iterative Subtask Detection ("ST*"). An iterative subtask ST corresponds from the DFSA point of view to a cycle. i.e. a *startState* state that has a path to an *endState* state, that also has a path to *startState* (see Table 6 for an example of conversion from DFSA to CTT). As a comment, an iterative subtask is usually linked with a disabling operator (not represented in Table 6).

Table 6. Example of an DFSA to CTT conversion, including an iterative subtask (states 2, 3, 4).

DFSA	
CTT	 abstract task >> abstract task* a >> >> b c d

Disabling Operator (“[>”). The CTT disabling operator indicates that a first subtask (usually an iterative one) is completely interrupted by a second subtask. From the DFSA point of view, if there exist, for each state of a subtask *st*, transitions to states that are out of the task with a same label t, we can conclude that there is a disabling operator (see Table 7 for an example of conversion from DFSA to CTT).

Table 7. Example of an DFSA to CTT conversion, disabling operator (for subtask containing states 2, 3, 4).

DFSA	
CTT	

DFSA to CTT model conversion algorithm (pseudocode) - disabling operator identification:

```
Input: SubTask st, TransitionLabel tl
states=st.getStates()
For i=0 to states.size-1
  If(!states[i].hasTransitionLabel(tl))
    Return False
  EndIf
EndFor
Return true
```

5.2 Task Model-Based Assistance

Once task models are generated from traces, they will be used to assist:

- the user in his task performance by showing him the actions that must be performed to achieve his objective, or

- the designer in the adaptation of his Web application by highlighting the observed usages of his application.

In both cases, we assume that tasks are organized by categories and that each task model is associated with metadata specifying the name of the task, its purpose and, possibly, the traces that generate it.

User Assistance. User assistance can be provided in three modes: manual search, search through the declaration of the purpose of the task and automatic search.

Manual Search. In this first mode, as a classic help, the user is supposed to select the task by navigating in a task tree. Tasks are structured hierarchically, thereby representing the different functionalities of the web application. For instance, for a booking site such as booking.com, the first level of the hierarchy will contain main tasks: accommodation, flights, car rental and Airport taxis. The second level will contain the different tasks of each functionality, for example for accommodation, the tasks to "search hotel", to "book hotel", to "pay" can be associated with it.

Search through the Declaration of Purpose. The second mode is a keyword search, entered by the user, specifying his objective. It is a question of searching in the task database those whose objective matches the user's keywords. This search is based on a similarity measure. Several metrics can be used, particularly those used in text mining, like Cosine Similarity between the words in the task database and the words introduced by the user describing its task.

The word-vector cosine metric can consider each set of words as a vector in which each dimension corresponds to a different term, and in which the length is set to the frequency with which each word has been observed. For the calculation, a list of stopwords is removed. The similarity between the task description word vector A and the potential suggestion word vector B is calculated as follows:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

where A_i and B_i are components of vectors A and B respectively. The similarity value ranges from −1 (meaning exactly opposite) to 1 (meaning exactly the same).

Automatic Search. In automatic mode, as the user performs his task, the assistance system displays the tasks that match the actions performed by the user. If several tasks match, the system will order them according to two possible options, namely the popularity of the tasks and the browsing history of the target user. Regarding the popularity of tasks: the tasks most requested by users will be displayed in the first positions followed by the least requested tasks. For browsing history, tasks that have never been requested by the target user will be displayed in the first positions, followed by tasks already performed by the target user.

Designer Assistance. Re-design assistance simply consists of displaying some usage indicators allowing the designer to become aware of the Web application uses and eventually adapt his Web application to these uses through CHI re-design for instance. The following classes of indicators are adapted versions of those defined in [13].

1. Stickiness class of indicators: this class of indicators specifies the ability of a task to attract and retain users' interest. Stickiness is evaluated by using stats (basic indicators) related to the "popularity" of a task, for instance: the count of the task accomplishments, the number of unique users that performed the task, etc.
2. Performance class of indicators: this class of indicators is related to the ease/difficulty encountered by users while performing a task. Performance is based on stats related to the time needed for accomplishing a task, the count of back and forth during the task accomplishment, etc.
3. Navigation class of indicators: this class of indicators analyses the way users performed tasks on an application (the order of tasks performed, task paths). Navigation is based on the analysis of transitions between tasks (tasks that follow/precede a given task).
4. Stop & resume class of indicators: this class of indicators analyses the distribution of interruptions in task accomplishments and seeks to explain how - and ultimately why - users interrupt and resume a task. In general, these interruptions are correlated to a decrease in users' understanding or motivation (too much steps to perform, data to provide). Some interruptions are final (final stops – drops out of ongoing tasks), others are followed by resumes.

6 Tests and Results

In order to validate our approach, we implemented the previously mentioned CTT operator identification from DFSAs algorithms, performed unit testing first and then tested our approach on two real scenarios. The Java language was used for coding the algorithms and the DFSAs were stored in XML files.

6.1 Unit Testing

We checked our algorithms for generating CTT operators on a set of 24 XML files, each containing a DFSA that correspond to a particular CTT operator. The identification of CTT operators was successful for 21 out of 24 files, that leads to an average success rate of 87.5% (see Table 8).

Table 8. Results of unit testing. Each row represents a DFSA (an XML file) that corresponds to a CTT operator [1].

Operator	Nb. of states	Result	Rate
Enabling	5	Success	100%
	7	Success	
	13	Success	
	5	Success	
	3	Success	
Disabling	9	Success	100%
	4	Success	
	3	Success	
Choice	4	Success	100%
	12	Success	
	14	Success	
	48	Success	
Order independency	4	Success	100%
	12	Success	
	14	Success	
	48	Success	
Optionality	6	Failure	50%
	4	Success	
Iteration	5	Failure	33%
	6	Success	
	4	Failure	
Finite iteration	9	Success	100%
	9	Success	
	13	Success	

6.2 Real Scenarios Testing

We have also checked our algorithms on two real scenarios that represent respectively the booking of a flight on "www.airfrance.fr" and the search for an itinerary on "www.google.fr/maps". For testing these scenarios and thus evaluating our algorithms, we studied algorithm capacities to correctly identify CTT operators.

The Air France scenario DFSA is composed of 265 states and mainly contains CTT operators that are often encountered in web browsing tasks (choice, order independency, enabling, disabling, iteration). We obtained an average operator identification accuracy rate of 76.58% (see Table 9). Some identification errors are related to each other, especially in the case of the disabling operator which very often take place in iterations. Therefore, if the identification of an iteration operator that contains a disabling operator fails, then the disabling operator will not be identified either.

Table 9. Results of the Air France scenario CTT operator identification [1].

Operator	Instances	Identified instances	Rate
Enabling	104	80	76,92%
Disabling	1	0	0%
Choice	49	38	77,55%
Order independency	3	3	100%
Iteration	1	0	0%
Total	**158**	**121**	**76,58%**

For the Google Maps scenario, the DFSA is composed of 64 states and the average operator identification accuracy rate is 71.73% (see Table 10). Most operators are correctly identified except the optional one (i.e. optionality of tasks).

Table 10. Results of the Google maps scenario CTT operator identification [1].

Operator	Instances	Identified instances	Rate
Enabling	30	21	70%
Choice	13	10	76,92%
Order independency	2	2	100%
Optionality	1	0	0%
Total	**46**	**33**	**71,73%**

7 Conclusion

This paper deals with the development of assistance systems, based on knowledge extracted from observed usages. With our approach, such assistance systems are based on task models, generated using interaction traces (logs) represented as finite state automata. A task represents all the user's actions performed on a device to achieve a given objective. A trace represents the history of the user's actions on the digital environment. The idea is to consider the traces left by users as sources of knowledge that an assistance system can employ to perform user-specific assistance, and thus overcome the limitations of assistance based on intended uses, sealed during the design phase of a system.

Task models generated from traces could thus be used to guide a user in performing his task or a designer in adapting the digital environment to the observed uses.

Our contributions are the following 1) the specification of the characteristics of task metamodels for user assistance, 2) the comparison of well-known existing task metamodels, that lead us to the selection of the CTT metamodel, 3) the development of a set of algorithms for the generation of CTT task models from traces and 4), the development of an assistance approach, based on task models.

In future work, we plan to deepen the study of existing task metamodels - being aware that not all of them have been covered. To our mind, UML statecharts support of identified characteristics strongly deserve to be studied. A comparative study of the compliant metamodels, assessing their ability to be quickly understood and manipulated by “novice” - “first time users”, could also be carried out to strengthen the selection of a given metamodel.

An interesting study about Web task model generation using CTT could be performed in order to compare our approach against another one based on the analysis of web sites code [14], and for investigating to what extend both of them could be employed in a complementary manner. We also plan to continue the development and evaluation of our conversion process and assistance system in using a large corpus of Web browsing data.

References

1. Sehaba, K., Encelle, B.: Generation of task models from observed usage - application to web browsing assistance. In Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, KMIS, vol. 3, pp. 74–82 (2019). https://doi.org/10.5220/0008068300740082. ISBN 978-989-758-382-7
2. Leshed, G., Haber, E.xM., Matthews, T., Lau, T.: CoScripter: automating & sharing how-to knowledge in the enterprise. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1719–1728. ACM, April 2008
3. Bigham, J.P., Lau, T., Nichols, J.: Trailblazer: enabling blind users to blaze trails through the web. In: Proceedings of the 14th International Conference on Intelligent User Interfaces, pp. 177–186. ACM, February 2009
4. Allen, J., et al.: Plow: a collaborative task learning agent. In: AAAI, vol. 7, pp. 1514–1519, July 2007
5. Amershi, S., Mahmud, J., Nichols, J., Lau, T., Ruiz, G.A.: LiveAction: automating web task model generation. ACM Trans. Interact. Intell. Syst. (TiiS) **3**(3), 14 (2013)
6. Mahmud, J., Borodin, Y., Ramakrishnan, I.V., Ramakrishnan, C.R.: Automated construction of web accessibility models from transaction click-streams. In: Proceedings of the 18th International Conference on World Wide Web, pp. 871–880. ACM, April 2009
7. Balbo, S., Ozkan, N., Paris, C.: Choosing the right task-modeling notation: a taxonomy. In: The Handbook of Task Analysis for Human-Computer Interaction, pp. 445–465 (2004)
8. Limbourg, Q., Vanderdonckt, J.: Comparing task models for user interface design. The Handbook of Task Analysis for Human-Computer Interaction **6**, 135–154 (2004)
9. Jourde, F., Laurillau, Y., Nigay, L.: Description of tasks with multi-user multimodal interactive systems: existing notations. Journal d’Interaction Personne-Système (JIPS) **3**(3), 1–33 (2014)
10. Paternò, F.: ConcurTaskTrees: an engineered notation for task models. In: The Handbook of Task Analysis for Human-Computer Interaction, pp. 483–503 (2004)
11. Paternò, F.: Task models in interactive software systems. In Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals, pp. 817–836 (2001)
12. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Comput. Netw. ISDN Syst. **14**(1), 25–59 (1987)
13. Sadallah, M., Encelle, B., Maredj, A., Prié, Y.: Leveraging learners’ activity logs for course reading analytics using session-based indicators. Int. J. Technol. Enhanc. Learn. **12**(1), 53–78 (2020). https://doi.org/10.1504/IJTEL.2020.103815
14. Paganelli, L., Paterno, F.: A tool for creating design models from web site code. Int. J. Softw. Eng. Knowl. Eng. **13**(02), 169–189 (2003)