Derivation of a Dialog Model from a Task Model by Activity Chain Extraction

Kris Luyten[1], Tim Clerckx[1], Karin Coninx[1], and Jean Vanderdonckt[2]

[1] Limburgs Universitair Centrum – Expertise Centre for Digital Media
Universitaire Campus, B-3590 Diepenbeek, Belgium
{kris.luyten, karin.coninx}@luc.ac.be, tim.clerckx@skynet.be
[2] Université catholique de Louvain – Institut d'Administration et de Gestion
Place des Doyens 1, B-1384 Louvain-la-Neuve, Belgium
vanderdonckt@isys.ucl.ac.be

Abstract. Over the last few years, Model-Based User Interface Design has become an important tool for creating multi-device User Interfaces. By providing information about several aspects of the User Interface, such as the task for which it is being built, different User Interfaces can be generated for fulfilling the same needs although they have a different concrete appearance. In the process of making User Interfaces with a Model-Based Design approach, several models can be used: a task model, a dialog model, a user model, a data model,etc. Intuitively, using more models provides more (detailed) information and will create more appropriate User Interfaces. Nevertheless, the designer must take care to keep the different models consistent with respect to each other. This paper presents an algorithm to extract the dialog model (partially) from the task model. A task model and dialog model are closely related because the dialog model defines a sequence of user interactions, an activity chain, to reach the goal postulated in the task specification. We formalise the activity chain as a State Transition Network, and in addition this chain can be partially extracted out of the task specification. The designer benefits of this approach since the task and dialog model are consistent. This approach is useful in automatic User Interface generation where several different dialogs are involved: the transitions between dialogs can be handled smoothly without explicitely implementing them.

keywords: Model-Based User Interface Design, Task model, Dialog model, ConcurTaskTree, State Transition Networks

1 Introduction

The design of User Interfaces (UIs) has changed over the years. Several approaches for designing "good" UIs were developed, in particular Model-Based User Interface (MBUI) design has received a lot of attention. Especially for the

design of multi-device UIs MBUI design is very useful; it allows the designer to think about the underlying models instead of focusing on the appearance of the UI. This makes the UI more reusable and suitable for the tasks it is meant to support.

Among all possible models involved in User Interface (UI) development, the dialog model probably remains the least explored one and the hardest to edit and exploit [11]. In the past, four significant milestones have been followed in the quest for a better understanding of what a dialog could be in a UI:

1. Understand a dialog, its properties, and concepts: a dialog should describe the interaction between a user and a UI.
2. Modelling a dialog: dialog modeling is still an open question. In [3], the advantages and disadvantages of five dialog models (i.e. Backus-Naur-Form grammars, state transition networks, production rules, and Hoare's Communicating Sequential Processes (CSP), and Petri nets) are compared leading to a conclusion that none of them holds all the desired properties. Green [6] reported that event/responses languages are more expressive than grammars and state transition networks.
3. Acquiring design knowledge for producing a quality dialog from existing sources of information: for instance, expressiveness, parametrised modularization, and executability are properties of interest that should be captured in design knowledge.
4. Generating dialog by incorporating part of this design knowledge and by relying on modelling concepts: dialog is definitely governed by task configuration, although dialog and presentation usually work hand in hand.

The evolution from the first layer to the ultimate one happened progressively while obtaining more experience concerning the dialog model. Today, we have gained enough experience to embody it in a method and its supporting software to enable designers to develop the UI dialog in a more formal, systematic, and reusable way rather than intuitive like it was in the past [3]. Consequently it makes sense to attempt dialog generation by relying on the above steps. This was not possible in the past. The goal of this paper is to demonstrate that a dialog can be produced by starting from a task model, rather than a data or domain model. This goal subsumes the proof that a coarse grained dialog can be straightforwardly derived from a task model rather than merely being constrained by it. The benefit of this approach is that, if the task model changes, the dialog model could change accordingly.

The remainder of this paper is structured as follows: Sect. 2 reports on some significant steps of different dialog models used in methods and tools for UI development, from the less expressive and executable to the most ones. To illustrate the context of this work our framework Dygimes is introduced in Sect. 3. The Dygimes framework serves as a testbench to implement the algorithm and test its feasibility. This is followed by an introduction to the ConcurTaskTrees task specification in Sect. 4. Sect. 5 explains how an activity chain can be used for extracting a dialog model out of a task model. This is followed by an explanation of the actual algorithm in Sect. 6. How the transitions between different

windows are invoked, when the dialog model has to be rendered in a real User Interface, is explained in Sect. 7. Finally, Sect. 8 discusses the applicability and the obtained results, followed by an overview of the future work.

2 Related Work

The State Transition Diagram [12] was probably the first and the most frequently used formal method to model a dialog, as expanded in State Transition Networks (STN) [18]. Other formal methods have also been investigated, but there was no tangible proof of a far superiority of one of them over the other ones with respect to all the criteria defined in [3].

GENIUS [7] produced from a data model multiple Dialog Nets, a specific version of a Petri Net for specifying coarse grained dialog in terms of transitions (unspecified or fully specified) between views. The advantage of Petri Nets over previously explored formal methods was that they show the flow on undistinguishable tokens and places and a mark can be introduced one at a time.

TADEUS [15] takes envisioned task and object models to infer design of a dialog model expressed in Dialog Graphs, which are both graphically and formally expressed, thus leading to model checking while keeping visual usability.

The ICO formalism [1], based on Petri nets, allows more expressive and modular dialog specifications than the earlier attempts. In addition, any ICO specification of a dialog can be directly executed, which reduces the gap between specification time and execution time.

Windows Transitions [9] also extends STNs so as to create a visual and formal method to specify coarse grained dialog between windows. The STN is based on a series of window operation and transitions and can be drawn on screenshots. However, this model is not generated, but produced by hand. By consequence there is no guarantee to preserve constraints imposed by the task.

Therefore, there is a need to produce a dialog model in this way while maintaining a lightweight approach. The closest work is probably TERESA [10], which is aimed at producing multiple UIs for multiple computing platforms. In this paper, we also rely on the mechanisms introduced in TERESA and expand them in several ways that will be outlined throughout this paper. The main differences between the TERESA tool and the Dygimes framework introduced in the next section, are that the latter supports runtime creation of UIs and the possibility to use different widget libraries in addition to mark-up languages. Instead of focusing on tool support the Dygimes framework is focused on automatic creation of UIs. While the TERESA tool offers a design environment, the Dygimes framework used in this paper offers a runtime environment with little tool support.

3 The Dygimes framework

The work presented in this paper is being implemented as part of the research framework *Dygimes* (Dynamically Generating User Interfaces (UIs) for Mobile

Computing Devices and Embedded Systems) [4, 8]. Although the different aspects presented in this paper can be used independently of Dygimes, this framework is used as a test-bed for reassuring the feasability of the proposed approach.

Dygimes is a framework for generating multi-device UIs at runtime. High-level XML-based UI descriptions are used, in combination with a task specification, an interaction specification and spatial layout constraints. The high-level XML-based UI description contains the Abstract Interaction Objects that are included in the UI. These Abstract Interaction Objects are mapped to Concrete Interaction Objects[17]; the mapping process can be guided by providing customised mapping rules.

It suffices for the reader to have a global idea of what is included in the Dygimes framework. For this purpose, Fig. 1 gives an overview of a UI *design* cycle. The following steps are included in a design cycle:

1. A ConcurTaskTrees (CTT, see Sect. 4) task specification is provided . The leaves in the task specification are annotated with abstract UI descriptions (UI building blocks). Graphical tool support to attach the UI building blocks is provided.
2. The task specification and abstract UI descriptions are merged into one "annotated" task specification. Both the task specification and UI descriptions can be expressed using XML. This allows a smooth integration and results in a singe XML document the system can process.
3. The Enabled Task Sets (ETSs, see Sect. 4) are calculated (a custom algorithm to calculate these is provided in the Dygimes framework).
4. The initial Task Set is located (the first Task Set which is shown in a UI when the application is used, see Sect. 6.2).
5. The dialog model can be created using the temporal relations between tasks and the ETSs provided in the task model (see Sect. 6.3). The dialog model is expressed as a State Transition Network (STN).
6. The Abstract UI description is generated out of the ETSs and the STN. The STN provides the navigation between the different dialog windows and the ETS specifies the necessary content of a dialog window.
7. The transitions are integrated into the UI description.
8. The actual UI is generated and shown to the designer or the user.
9. The designer can test the UI and provide feedback by changing the Compound Task Sets (CTS, see Sect. 4) and Abstract Presentation.
10. The Compound Task Sets can be adapted by the designer.
11. Transitions are recalculated according to the new Compound Task Sets.

Although this is not a traditional design cycle as one can find in [5] for example, it is closely related to transformational development and software prototyping in Software Engineering [16].

4 ConcurTaskTrees formalism

The previous section introduced the Dygimes framework which uses a ConcurTaskTrees specification as one of its input documents. The ConcurTaskTrees task

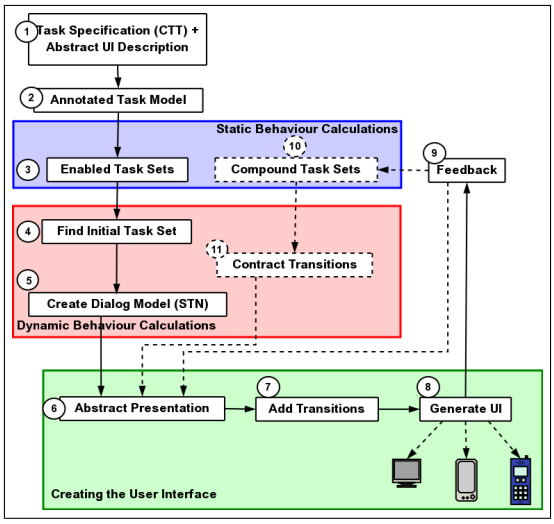


Fig. 1. The Dygimes User Interface design and generation process

model (CTT) is a method proposed by Paternò [13] for designing a task specification. This notation offers a graphical syntax, an hierarchical structure and a notation to specify the temporal relation between tasks. Four types of tasks are supported in the CTT notation: abstract tasks, interaction tasks, user tasks and application tasks. These tasks can be specified to be executed in several iterations. Sibling tasks, appearing in the same level in the hierarchy of decomposition, can be connected by temporal operators like choice ([]), independent concurrency (|||), concurrency with information exchange (|[]|), disabling ([>) , enabling (>>), enabling with information exchange ([]>>), suspend/resume (|>) and order independency (|=|). [14] specifies the following priority order among the temporal operators: *choice* > *parallel composition* > *disabling* > *enabling*.

For a good understanding of the rest of this paper, we introduce the following notation. Let $\mathcal{T}$ be an infinite set of tasks. By $\mathcal{O}$ we denote the set of temporal operators {[], |[]|, |=|, [>, >>, []>>, |>}.

Definition 1. A *CTT task model* M is a rooted directed tree where every node is a task in $\mathcal{T}$. In addition, there can be arcs between tasks carrying labels from $\mathcal{O}$ modeling connections by temporal operators. An arc labeled o from task t to t' is denoted by $t \xrightarrow{o}_M t'$.

For the remainder of the paper we fix a task model M. Using the introduced notation we can define a set of tasks of a task model M:

Definition 2. By $\mathcal{T}(M)$ we denote the set of tasks occurring in M.

We will define also other relations in this paper. These definitions have the sole purpose to support the development of the algorithm. A more precise way for defining semantics can be using Kripke semantics [2] for example. This would go beyond the scope of this paper, but is clearly an important approach when we want to prove its correctness.

We will make use of the "mobile phone task specification" for illustration purposes; a task model describing some functionalities offered by a regular mobile phone. It describes the steps to adjust the volume or read an SMS message. The task specification is shown in Fig. 2(a).

A very important advantage of the CTT formalism is the generation of *Enabled Task Sets* (ETS) out of the specification. [13] defines an ETS as:

> a set of tasks that are logically enabled to start their performance during the same period of time.

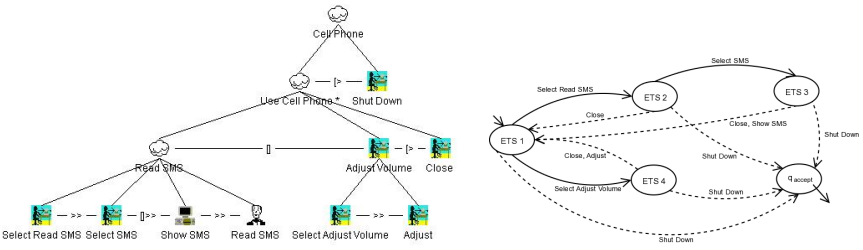
An Enabled Task Collection (ETC) E is a set of sets of tasks ($E \subseteq 2^{\mathcal{T}}$). In [13] an algorithm is given to compute a specific ETS of a given task model M, we denote the latter by $E(M)$. Usually there are several ETSs which can be calculated out of a task model.

In our framework, the ETSs are calculated by transforming the CTT specification into a priority tree and applying the predefined rules (somewhat modified) described in [13]. A priority tree is a CTT specification, where all the temporal relations of the same level in the task hierarchy have the same priority according to their defined order. Such a tree can be obtained by recursively descending into the CTT specification inserting a new level with abstract tasks where the temporal operators on the same level do not have the same priority. This does not change the semantic meaning.

The ETSs calculated from the model in Fig. 2(a) are:

$$\begin{aligned} ETS_1 &= \{Select\ Read\ SMS, Select\ Adjust\ Volume, Close, Shut\ Down\} \\ ETS_2 &= \{Select\ SMS, Close, Shut\ Down\} \\ ETS_3 &= \{Show\ SMS, Close, Shut\ Down\} \\ ETS_4 &= \{Select\ Adjust\ Volume, Close, Shut\ Down\} \end{aligned} \tag{1}$$

Based on the heuristics given in [14], adhesive and cohesive tasks can be defined as follows. Two tasks $t, t' \in \mathcal{T}(M)$ are **cohesive** w.r.t. a task model M if there is a set $S \in E(M)$ such that $\{t, t'\} \subseteq S$. Two tasks, t and t', are called **adhesive** if they are not cohesive, but they are semantically related to some extent. [14] also introduces some rules for merging ETSs, which can be very useful when there are a lot of ETSs. These heuristics can be used to identify adhesive tasks: two tasks which do not belong to the same ETS, but their respective ETSs can be merged into one ETS. On the other hand, merging ETSs can cause problems when a UI for a mobile phone has to be generated starting from the task model. Typicallly, a mobile phone can only show one task at the same time due to the screen space constraints. Consequently, a fine-grained set of ETSs can ease the automatic generation of UIs subject to a (very) small screen space.



(a) A CTT specification for using some functionalities offered by a mobile phone.

(b) State Transition Network describing the behaviour of Fig. 2(a)

Fig. 2. A CTT diagram for a mobile phone (a) and its State Transition Network (b).

In addition, we define a Compound Task Set (CTS) based on the definition of a task set:

Definition 3. A **Compound Task Set** of a task model M is a collection of tasks $C \subseteq \mathcal{T}(M)$ such that

- every two distinct tasks in C are cohesive or adhesive; and,
- for every $t \in C$ there is an $S_t \in E(M)$ such that $t \in S_t$; in addition, $S_t \subseteq C$.

The different CTSs indicate which UI building blocks (attached to the individual leaf tasks) can be presented as a group to the user. Notice the composition of a CTS depends on the heuristics the designer applies. In step 9 of Fig. 1, the designer can choose to group certain ETSs. Our system relies on the designer, instead of using heuristic rules to contract the ETS as proposed in [14]. The heuristic rules can be used to guide the designer to make better decisions in a next implementation of the Dygimes framework.

The next step is to discover for every $S \in E(M)$ the set $R \subseteq E(M)$ of ETSs where every ETS in R can replace S when that ETS is finished. So, we will try to discover transitions that allow the user to go from one ETS to another according to the temporal relations defined in the task specification.

5 Activity Chain Extraction

We define an *Activity Chain* as a path of different dialogs to reach a certain goal. A dialog is uniquely defined by the ETS which it presents. Each dialog is considered a step in the usage of the application, so a graph of ETSs can be built representing the flow of the dialogs the user sees. Each ETS is a node in this graph and can have a directed edge to other ETSs, which represents the transition of one dialog to another dialog. In addition, a start task set can be

identified presenting the initial dialog. Given these properties, the activity chain can be specified as a State Transition Network (STN).

A STN is defined as a *connected graph* with *labeled directed edges*. Edges are labeled with tasks. Nodes are sets in $E(M)$. In addition, there is an *initial node* and a *finishing node*.

For example, Fig. 2(b) shows the STN for the task model shown in Fig. 2(a). Intuitively, a STN seems insufficient to describe the behaviour because of the concurrency supported in the task model. However, tasks which are executing in the same period of time belong *to the same ETS*, which makes concurrent ETSs unnecessary. We will only need one active state in the generated dialog specification; STN are sufficient for this purpose. This condition may not hold when collaborative UIs are considered; this paper only discusses the applicability for single user applications.

In Fig. 2(b), the transitions between states are labeled with the task names, where *Shut Down* is the exit transition. The goal is to find and resolve these transitions automatically, so the generated UI is fully active and the system offers support for several related dialogs, without the need to implement the transitions explicitely. By using automatically detected transitions, the UI designer can make an appropriate dialog model without burdening the developer. It suffices to describe the "transition conditions" when a transition is triggered in the program code. Three different level of dialogs can be identified: intra-widget (how widgets behave), intra-window (what is the dialog inside a window) and inter-window (how the different windows behave w.r.t each other). The focus in this paper lies on the inter-window level.

Transitions between states in the STN can be identified by investigating the temporal operators that exist between tasks in the task specification. Usually, one proceeds from one ETS to another when an enabling or disabling operator is detected. More formally (based on definition 1), this can be expressed by introducing the following definition:

Definition 4. Let $S_1, S_2 \in E(M)$, t_1 is a **candidate transition** in one of the following cases:

- $t_1 \overset{>>}{\rightarrow}_M t_2$, $t_1 \in S_1$ and $t_2 \in S_2$
- $t_2 \overset{[>}{\rightarrow}_M t_1$, $\{first(t_1), t_2\} \subseteq S_1$ and $body(t_1) \subseteq S_2$

Here, *first(t)* is the first subtask of t that is to be performed, and *body(t)* are the subtasks of t not included in *first(t)*. These two functions are defined in [13].

6 Dynamic Behaviour of the User Interface

In short, building the STN to guide the activity chain consists of:

1. A set of states; every ETS is mapped on a state;
2. A set of transitions; every task involved in a disabling or enabling relation can be a candidate transition as described in definition 4;

3. An initial state; this is the unique initial ETS shown to the user;
4. A set of finishing states; the set of ETSs that can terminate the application;
5. An accept state; arriving in the accept state will terminate the application. The accept state can be reached out of a finishing state.

The rules we will show here are obtained based on *empirical results*, not by mathematical deduction. This means we can not prove they are correct in every situation, but only know that they work in many situations. We plan to generalize the rules so they can be checked more formally. For now; the algorithm is implemented and tested succesfully on several examples.

In the next section, we will show some example rules to extract the STN out of the task specification. This is done in four steps: finding the states of the STN, locating the start state, collecting the transitions between states and finally locating the finishing or "accept" state. The most challenging part is collecting the transitions of the STN: this requires investigating the temporal relations in the task model and deciding which task will invoke another ETS.

Before we continue, we define two functions: *firstTasks* and *lastTasks*:

firstTasks : takes a node n of the CTT task model and returns the left-most group of leaves that are descendants of n and are all elements of the same ETS. This function will return a single ETS if no ancestor of n is involved in a concurrent relation.

lastTasks : takes a node n of the CTT and returns the right-most group of leaves that are descendants of n. When a concurrency or a choice relation is specified between siblings on the right hand-side, these are processed recursively and joined together.

6.1 Mapping Sets on States

The easiest part is finding which states to use in the STN. This is a one-to-one mapping of all ETSs which can be retrieved from the Task Model. So every $s \in E(M)$ is a state in the STN. For example; in the STN for the mobile phone example (Fig. 2(a)), each ETS is mapped on a state resulting in 4 different states.

6.2 Finding the Initial State

The initial state can be found by mapping the first ETS that will be presented to the user onto this state. This ETS is referred to as the *start ETS* or S_s. To find this ETS we first seek the left-most leaf t_l in the CTT specification that is not a user task. This task appears before every enabling temporal operator so it must belong to the start task set.

However, t_l can belong to different ETSs when it has an ancestor involved in a concurrent temporal relation. If t_l only belongs to one ETS, the start ETS is found. To find S_s when t_l belongs to more than one ETS we check which tasks must belong to S_s by a recursive calculation of the first of the root. The ETS

containing all the tasks of *firstTasks(root)* is selected. This ETS is unique, because the root node can not have ancestors involved in a concurrent relationship with its siblings.

Consider the example in Fig. 3(a). Taking the *firstTasks(root)* gives us $S_s = \{Task_{1,1}, Task_{2.1}\}$.

6.3 Detecting Transitions

Once all ETSs are mapped onto states of the STN, transitions between the different states have to be detected. Transitions are regular tasks; in [14] transition tasks between *Task Sets* are also defined but without letting the system detect them automatically. Our approach detects the transitions *automatically* relying on the temporal operators in the task model.

To detect candidate transition tasks, the task model has to be scanned for all candidate transitions according to definition 4. First, the task model is transformed into a priority tree before further processing. For every candidate task $t_1 \in \mathcal{T}(M)$ where $t_2 \overset{[>}{\rightarrow}_M t_1$ or $t_1 \overset{>>}{\rightarrow}_M t_2$ and t_1 and t_2 belong to different ETSs[1], the selection of transition tasks out of the canditate transitions can be done as follows:

If the temporal operator is ***enabling****:* $t_1 \overset{>>}{\rightarrow}_M t_2$, one of the following four steps is taken:

1. t_1 and t_2 are leaves, one of the two following two steps is valid:
   (a) t_2 belongs to just one task set: all ETSs containing t_1 trigger the ETS that contains t_2.
   (b) t_2 belongs to several ETSs: for every e in $E(M), t_1 \in e$, there is a transition T and a task set T_l where T_l is the same task set as e except for t_1 is replaced by t_2 in the task set. Such a task set exists due to the presence of a concurrency temporal operator between ancestors of t_1 and t_2. Fig. 3(b) shows the enabling transitions of the task model in Fig. 3(a). Consider the enabling relation between *Task 1.1* and *Task 1.2*. Three enabled task sets contain *Task 1.1* namely ETS_1, ETS_2 and ETS_3 and three other enabled task sets contain *Task 1.2* namely ETS_4, ETS_5 and ETS_6. ETS_1 and ETS_4 differ only by one task. Here *Task 1.1* is replaced by *Task 1.2* so we introduce this transition: $ETS_1 \xrightarrow{Task1.1} ETS_4$. All other transitions in this example can be found in the same way (e.g. ETS_2 and ETS_5 differ by the same tasks).
2. t_1 is a leaf, t_2 is not: t_1 triggers the ETS of the *firstTasks(t_2)* if t_1 and t_2 have no ancestor involved in a concurrent relation. Fig. 4 shows how this situation maps on a STN. Even if one of the descendants of t_2 is involved in a concurrency or choice relation this does not change the process: they would belong to the same ETS by definition. If there is an ancestor of t_1 and t_2 which has a concurrent temporal relation with another task, detecting the correct transition is more difficult, and an approach similar as 1b is applied.

[1] Notice t_1 and t_2 have the same ancestors, since they are siblings

3. t_2 is a leaf, t_1 is not: in this case the triggering task is found by taking the right-most leaf of the descendants of t_1. This can be done by using the function *lastTasks*: the tasks returned by *lastTasks(t_1)* are the tasks that trigger t_2. Fig. 5 shows an example. If there is an ancestor of t_1 and t_2 which has a concurrent temporal relation with another task, the same approach as 1b has to be applied.
4. Neither t_1 nor t_2 are leaves: *lastTasks(t_1)* collects the "last" tasks of t_1 as in 3. Now apply 2 on each last task and t_2 as if they had an enabling temporal operator between each other. Fig. 6 shows an example.

If the temporal operator is ***disabling****:* $t_1 \overset{[>}{\rightarrow}_M t_2$, then $first(t_2)$ is a *disabling task* (*first* is defined in Sec. 5) and one of the following steps is taken:

1. If t_2 has an ancestor that is an iterative task t_i: for each enabled task set E containing t_2 add a transition $E \overset{t_2}{\rightarrow} startTaskSet(t_i)$, where *startTaskSet($t_i$)* is the first task set of the subtree with root t_i.
2. If t_2 has an ancestor a with an enabling operator on the right hand side: let r be the right hand side sibling of a and add transitions as if there is an enabling operator between t_2 and r.
3. In all other cases; for each enabled task set E containing t_2 add a transition to the accept state: $E \overset{t_2}{\rightarrow} q_{accept}$. q_{accept} is a finishing state and will be further explained in Sect. 6.4.

Fig. 7 shows a CTT specification with a disabling relation and the extracted STN. Notice how the task *Quit* is responsible for both transitions to the accept state.

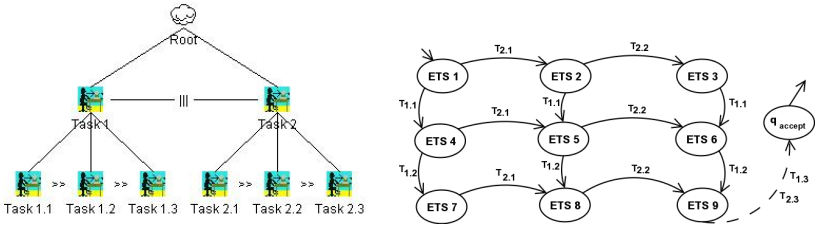
6.4 Mapping the Finishing States

To complete the STN, the "last" ETSs with which the user interacts have to be located. For this reason we introduce a new definition for *expiring task*.

Definition 5. A task $t \in \mathcal{T}(M)$ is an **expiring task**, when t is a leaf and there is no $t' \in \mathcal{T}(M)$ such that $t \overset{o}{\rightarrow}_M t'$ with $o \in \mathcal{O}$.

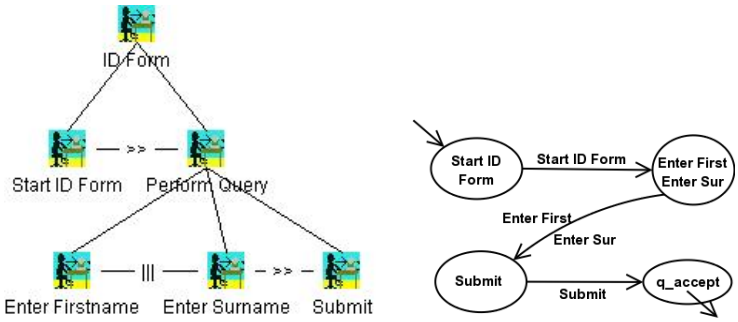
If an expiring task t_e has an ancestor with an enabling operator on the right hand side, we have already taken care of this leaf by detecting enabling transitions (see Sect. 6.3). If this is not the case, further examination of the task is required:

- If t_e has an ancestor that is an iterative task t_i: for each enabled task set e containing t_e add a transition: $e \overset{t_e}{\rightarrow} startTaskSet(t_i)$.
- Else: for each enabled task set e containing t_i add a transition: $e \overset{t_e}{\rightarrow} q_{accept}$.



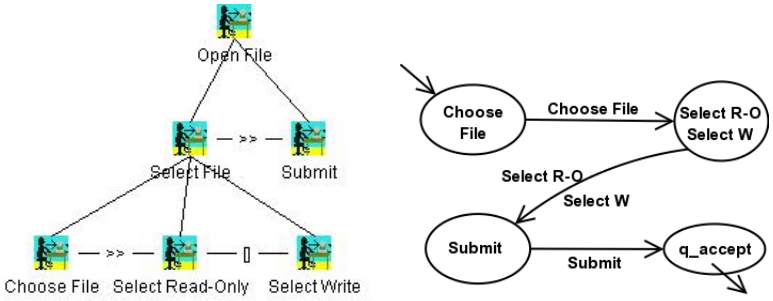
(a) ConcurTaskTrees diagram (b) State Transition Network

Fig. 3. CTT with concurrency (3(a)) and the resulting State Network Diagram (3(b))



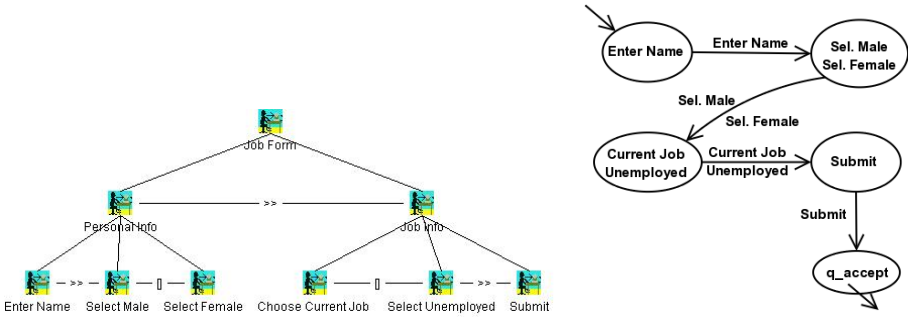
(a) ConcurTaskTrees diagram (b) State Transition Network

Fig. 4. $Task_{Start\ ID\ Form}$ is a leaf and $Task_{Perform\ Query}$ is no leaf.



(a) ConcurTaskTrees diagram (b) State Transition Network

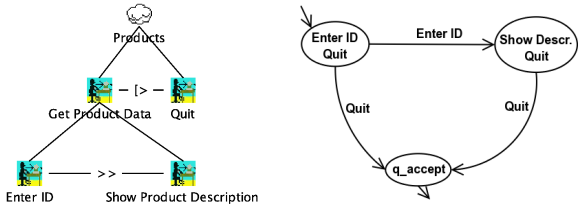
Fig. 5. $Task_{Submit}$ is a leaf and $Task_{Select\ File}$ is no leaf.



(a) ConcurTaskTrees diagram

(b) State Transition Network

Fig. 6. Neither $Task_{Personal\ Info}$ or $Task_{Job\ Info}$ are leaves.



(a) CTT with disabling task

(b) The resulting STN for (a)

Fig. 7. Extracting the STN when a disabling relation is involved

6.5 The resulting State Transition Network

Once the system has processed the steps described in the previous sections, a complete STN has been built. This STN describes a dialog model that is correct w.r.t. the task specification: the order of tasks that can be executed (the order between the ETS) is now also expressed in the STN. This is a powerful tool for designers to check whether their task specification meets the requirements before the working system has to be built. The designer can rely on the STN to produce a usable prototype supporting navigation through the UI.

7 Actual transitions between dialogs

Once the STN is completely defined, the system still lacks a way of detecting the actual conditions when the transition takes place. In the Dygimes framework,

where high-level XML-based UI descriptions are attached as UI building blocks to leafs in the task model, a specialised action handling mechanism [4] is implemented to take care of the state transitions. For now, widgets playing a key role for a dialog (e.g. a "next" button in an installation wizard) are identified by the designer by introducing a special "action" and attaching it to the presentation model. The specialised action contains the preconditions and will be executed only when the preconditions are fullfilled.

When several concurrent tasks are included in the same ETS, which are all labeled as a transition in the STN, the system has to wait untill all these tasks are finished. This can not be detected in the STN, in this case knowledge about the temporal relations is necessary. One possible solution is to group the concurrent tasks and handle them as if they were one task. For desktop systems concurrent tasks are not unusual, but for small devices (like PDA, mobile phone) the concurrency will be limited by the constraints of the device.

8 Conclusions and Future Work

We have presented a method for automating the creation of the dialog specification using the task specification. The extraction of a dialog model out of the task specification using the temporal relations between tasks offers some useful possibilities. First of all, using an algorithm rather than relying on the designer results in a correct dialog model w.r.t the task specification. It also helps the designer to make the design cycle shorter by automating some things which had to be done (partially) manually.

The following step is to prove the correctness of our statements. Therefor we first need a proof of the correctness of the ETS calculation algorithm. We will look into providing a formal proof to strengthen our case in the future. This paper applies an empirical approach for extracting the dialog model using the task model. The rules provided here were obtained by practical experience rather then by mathematical deduction. The correctness of the algorithm will be emphasized during the further development of the Dygimes framework. Still, we think the method presented in this paper can improve consistency between the dialog and the task model in Model-Based UI design.

Acknowledgments Tim Clerckx is a student at the Limburgs Universitair Centrum. In the context of his master's thesis, Tim has contributed significantly in the design of the algorithm and the test implementation. The authors would like to thank Frank Neven and Quentin Limbourg for their useful comments and contributions to this text. The research at the Expertise Centre for Digital Media (EDM/LUC) is partly funded by the Flemish government and EFRO (European Fund for Regional Development).

References

1. Rémi Bastide and Philippe Palanque. A Visual and Formal Glue Between Application and Interaction. *Visual Language and Computing*, 10(3), 1999.

2. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 1999.
3. Gilbert Cockton. Interaction Ergonomics, Control and Separation: Open Problems in User Interface Management. *Information and Software Technology*, 29(4):176–191, 1987.
4. Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Bert Creemers. Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems. In *Mobile HCI*, 2003. accepted for publication.
5. Alan Dix, Janet Finlay, Gregory Abowd, and Russel Beale. *Human-Computer Interaction (second edition)*. Prentice Hall, 1998.
6. Mark Green. A Survey of Three Dialog Models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.
7. Christian Janssen, Anetee Weisbecker, and Jürgen Ziegler. Generating User Interfaces from Data Models and Dialog Net Specifications. In *ACM Conf. on Human Aspects in Computing Systems InterCHI'93*, pages 418–423, Amsterdam, April 24–28 1993. Addison-Wesley.
8. Kris Luyten, Chris Vandervelpen, and Karin Coninx. Migratable User Interface Descriptions in Component-Based Development. In Peter Forbrig, Quentin Limbourg, Bodo Urban, and Jean Vanderdonckt, editors, *Interactive Systems: Design, Specification, and Verification*, volume 2545 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2002.
9. Efrem Mbaki and Jean Vanderdonckt. Window Transitions: A Graphical Notation for Specifying Mid-level Dialogue. In *First International Workshop on Task Models and Diagrams for User Interface Design TAMODIA2002*, pages 55–63, July 18–19 2002.
10. Giullio Mori, Fabio Paternò, and Carmen Santoro. Tool Support for Designing Nomadic Applications. In *Intelligent User Interfaces*, pages 141–148, January 12–15 2003.
11. Dan Olsen. *User Interface Management Systems: Models and Algorithms*. Morgan Kaufman, 1992.
12. David L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 1969 24th national conference*, pages 379–385, 1969.
13. Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2000.
14. Fabio Paternò and Carmen Santoro. One model, many interfaces. In Christophe Kolski and Jean Vanderdonckt, editors, *CADUI 2002*, volume 3, pages 143–154. Kluwer Academic, 2002.
15. Egbert Schlungbaum and Thomas Elwert. Dialogue Graphs - a formal and visual specification technique for dialogue modelling. In *Formal Aspects of the Human Computer Interface*, 1996.
16. Ian Sommerville. *Software Engineering*. Addison-Wesley, 5th edition, 1996.
17. Jean Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *ACM Conference on Human Aspects in Computing Systems InterCHI'93*, pages 424–429. Addison Wesley, 1993.
18. Anthony Wasserman. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, 11:699–713, 1985.

