Automatically Testing Interactive Applications Using Extended Task Trees

Laya Madani

University of Grenoble
Université Joseph Fourier - Laboratoire d'Informatique de Grenoble
BP 53 - 38041 Grenoble Cedex 9 - France

Ioannis Parissis

University of Grenoble
Grenoble INP - Laboratoire de Conception et d'Intégration des Systèmes
BP 54 - 26902 Valence Cedex 9 - France

Abstract

Task trees are common notations used to describe the interaction between a user and an interactive application. They contain valuable information about the expected user behaviour as well on the expected software reactions and, thus, they can be used to support model-based testing. In this paper, a method for automatically generating test data from task trees is introduced. The task tree notation is extended to support operational profile specification. The user behaviour is automatically extracted from such extended trees as a probabilistic finite input-output state machine, thanks to formal semantics defined for this purpose for the task tree operators. The resulting probabilistic machine can then be used to generate test data simulating the user behaviour. This simulation can be performed using Lutess, a testing environment developed for synchronous software. The translation of the user interaction model into a Lutess description is explained and experimental results are reported.

1 Introduction

Interactive applications ensure the access to various commercial services (mobile phones, reservation systems or telecommunication services) and are increasingly involved in several critical domains such as flight or industrial pro-

Email addresses: laya.madani@imag.fr (Laya Madani), ioannis.parissis@esisar.grenoble-inp.fr (Ioannis Parissis).

cess control. Therefore, their correctness becomes a very important issue and their development requires thorough validation. Several automated methods have been proposed for verifying and validating interactive systems based on formal specifications such as the FSM (Formal System Modeling) analysis [9], the LIM approach (Lotos Interactor Model) [10], the ICO (Interactive Cooperative Object) formalism based on Petri Nets [26, 23] or model-checking [6] using the Lustre language [5]. In most of these approaches, the interactive application is formally described as an abstract model and various properties which must hold are verified on this model by means of traditional verification techniques (e.g. model checking). Using the B method has also been suggested [1] where proofs during the refinement process ensure that properties are preserved. Model-based testing methods focusing on the specification of the user behaviour have also been studied. For instance, in [13], a simple task model is used to exhaustively generate the interaction scenarios. The method presented in [27] relies on the specification of a finite state machine representing the behaviour of the system while in [20] the interface is modeled by means of hierarchical operators, preconditions and post-conditions.

A method to test interactive systems based on the synchronous approach has been recently proposed in [19, 4]. The synchronous approach has been successfully used to model and to implement reactive systems. Thanks to the underlying *synchronism hypothesis*, the program specification and verification becomes simpler and easier. The proposed testing approach suggests using the Lutess testing environment [8], which requires a partial formal specification of the software user behaviour, provided as a set of Lustre [5] expressions. This specification can be enhanced with operational profiles. Several test generation strategies can be applied to the resulting test model to automatically produce input data. Test input generation is carried out "on the fly" (test inputs are computed according to the previously produced inputs and outputs).

Although this approach seems promising, it requires a formal specification that is not easy to provide for interactive application designers who are not familiar with synchronous languages. This is a concern for all the verification methods for interactive applications based on formal notations. For this reason, test data generation methods based on task trees, more common in interactive application development, have been studied [18]. Task trees are built at early stages of the application design and describe the interactions between an application and the user and, hence, provide a model of the user behaviour. In this paper it is suggested to enhance task trees with occurrence probabilities in order to support operational profile definitions. Then, it is shown how such extended task trees can be transformed into a probabilistic input-output FSM modeling the user behavior (similarly to approaches on probabilistic modeling of reactive systems, e.g. [24, 14]). This model is used to automatically generate test data using the Lutess environment. A similar model is adopted in [2] as well as in [17, 16], to compute the probability (say p) that the user interacts

with an implementation under test (IUT) by means of a set of test sequences. If this set is applied to the IUT and the latter reacts as expected in the specification, then one can conclude that the IUT is correct with a probability at least equal to p (since non tested sequences can be also correct). Similarly, the upper bound of the probability to find errors in the IUT is $1 - p$. These investigations also suggest a criterion to assess the suitability of a set of test sequences, minimizing this upper bound and, hence, increasing the probability to find an error in the IUT.

The paper is organized as follows. Section 2 introduces task trees and the particular notation (CTT) used in this research work. Section 2.2 proposes an extension of this model including operational profile definition. Section 3 formally defines the transformation of this extended task tree into a probabilistic I/O machine and shows how this machine can be used to generate test sequences. Section 4 focuses on test data generation, in particular when using the Lutess environment.

2 Using Task Trees for Model-Based Testing

2.1 Task trees

Task models are often used in the design of interactive software applications and are usually built by human factors specialists. In such models [7], tasks are represented hierarchically: a task consists of subtasks combined by temporal operators. Therefore, the model describes the subtasks that must be executed to fulfill another, more complex, task.

A well known notation for task models is ConcurTaskTrees (CTT) [21]. CTT includes four kinds of tasks: **User tasks** (no interaction with the system, just an internal cognitive activity such as thinking about how to solve a problem), **application tasks** (system performance, such as generating the results of a query, no interaction with the user), **interaction tasks** (involving user actions with immediate feedback from the system, such as editing a document) and **abstract tasks** (tasks composed of other subtasks).

A CTT abstract task is composed of subtasks connected by means of temporal operators [21] described in Table 1.

Choice	T1[]T2	One task from T1 and T2 is chosen.
Independent Concurrency	T1|||T2	Actions belonging to two tasks can be performed in any order without any specific constraints.
Concurrency with information exchange	T1|[]|T2	In this case T1 and T2 exchange information other than the concurrent execution.
Order Independence	T1|=|T2	Both tasks have to be performed but when one is started then it has to be finished before starting the second one.
Deactivation	T1[>T2	The first task is definitively deactivated once the first action of the second task has been performed.
Enabling	T1> >T2	In this case one task enables a second one when it terminates
Enabling with information passing	T1[]> >T2	In this case task T1 provides some information to task T2 other than enabling it.
Suspend-resume	T1|>T2	This operator gives T2 the possibility of interrupting T1 and then when T2 is terminated, T1 can be reactivated from the state reached before the interruption.
Iteration	T*	This means that the task T is performed repetitively until the task is deactivated by another task.
Finite Iteration	T1(n)	It is used when designers know in advance how many times a task will be performed.
Optional tasks	[T]	This indicates that the performance of a task is optional.

Table 1
The CTT operators

2.2 Adding Operational Profiles to Task Trees

Operational profiles [22] provide information about the effective usage of an application. In particular, they can be used to guide the test process. For the particular case of interactive applications, it would be convenient to define such operational profiles on task trees. Indeed, assuming the latter to be models of the user behaviour and to exhaustively represent the interactions between the user and the application, operational profiles can be easily defined by assigning

occurrence probabilities to some of the described behaviours. It is suggested to extend the CTT notation to make possible to assign occurrence probabilities to the user actions involved in the tree operators, according to the following syntax:

$$
\begin{aligned}
T ::= t \quad & | \quad T[]_{pr_1,pr_2}T \quad | \quad T|||_{pr_1,pr_2}T \quad | \quad T[>_{pr}\ T \quad | \quad T^* \\
& | \quad T|>_{pr}\ T \quad | \quad [T]_{pr} \quad | \quad T >> T \quad | \quad T[]>> T \quad | \quad T(n) \\
& | \quad T|[]|_{pr_1,pr_2}T \quad | \quad T|=|T
\end{aligned}
$$

where t is an elementary task (that is, an application task or a user task). We assume that if t is an application task, then it is always followed by an "enabling" operator. In other words, application tasks are preconditions for other, interactive or abstract, tasks. This hypothesis simply means that the task tree provides a model of the user behaviour and not a specification of the application.

Occurrence probabilities are assigned to operators as follows:

Choice operator: $T = A[]_{pr_A,pr_B}B$ where $pr_A + pr_B = 1$. An execution probability is specified for every subtask (A, B).

Concurrency operators: $T = A|||_{pr_{ActA},pr_{ActB}}B$ (or $T = A|[]|_{pr_{ActA},pr_{ActB}}B$) where $pr_{ActA} + pr_{ActB} = 1$. To execute the task T, all the subtasks must be executed. However, only one action from these subtasks is executed at the same time. Occurrence probabilities can then be assigned to the actions of these sub-tasks. This means that for every state in the execution of T, the probability to execute an action from task A is pr_{ActA} and the probability to execute an action from task B is pr_{ActB}. This distribution of probabilities holds when it is possible to execute actions from A and B. If actions from A are no longer available, then the probability to execute an action from B is 1.

Deactivation operator: $T = A[>_{pr_{deac}}\ B$ where $pr_{deac} \leq 1$. In any state of A, the probability for A to be interrupted by B is pr_{deac}.

Suspend-resume operator: $T = A|\ >_{pr_{sus}}\ B$ where $pr_{sus} \leq 1$. In any state of A, the probability for A to be suspended by B is pr_{sus}.

Optional task: $[A]_{pr_A}$ where $pr_A \leq 1$. The probability to execute the task A is pr_A.

Enabling operators, Iteration and Finite iteration: The tasks involved in these operators are executed sequentially with no possible user choice. So, there are no probability assignments.

Remark 1 *The operator Order Independence between two tasks (*A/=/B*) means that both tasks must be executed and they can be executed in any order. So, we consider that* $(A \ /=/ \ B) = ((A>>B) \ []_{0.5,0.5} \ (B>>A))$.

Example: The interactive application "Memo" [3] makes possible to annotate physical locations with digital stickers ("post it"-like notes). Once a digital sticker has been set to a physical location, it can be read/carried/removed by other users. A Memo user is equipped with a GPS and a magnetometer enabling the system to compute his/her location and orientation. S/he is also wearing a head mounted semi-transparent display (HMD) enabling the fusion of computer data (the digital notes) with the real environment.

Memo provides three main tasks: (1) orientation and localization of the mobile user, so that the system is able to display the visible notes according to the current position and orientation of the mobile user (2) manipulation of a note (get, set and remove a note) and (3) exiting the system. So, the mobile user can get a note and carry it while moving. S/he can set a carried note to a specific place or delete a visible or carried note.

Consider the following operational profile of the user: The user doesn't have a preference between exploring the ground and handling notes; s/he prefers handing a displayed note than a carried one; if a note is displayed, s/he prefers getting it than removing it; if s/he carries a note, s/he prefers removing it than setting it.

Figure 1 shows an extended CTT for the Memo system including this profile. This tree states that the user can use the application repetitively (iteration operator *), activity that can be interrupted (deactivation operator [>) with a probability of 0.1 by the "exit" task. A memo application task is a choice (operator []) between two tasks: exploring the ground (probability 0.5) and handling notes (probability 0.5). Handling notes requires choosing between handling a displayed note (probability 0.6) or handling a carried note (probability 0.4). If the system displays a note (memoDisplayed task), the user can (enabling operator >>) get or remove this note. If the user is carrying a note (memoCarried task), s/he can (enabling operator >>) set or remove this note.

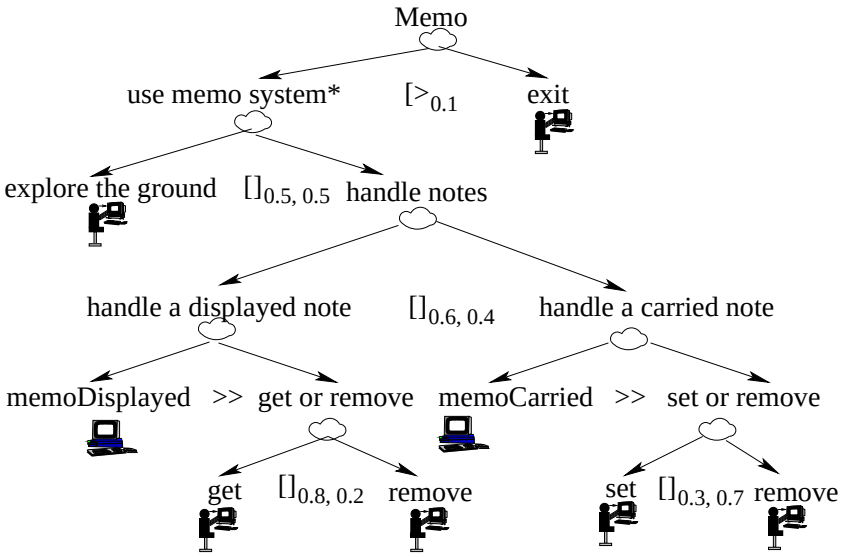

Figure 1. Example of extended CTT

The task "get or remove" chooses between two interactive tasks ("get", "remove"), with a probability of 0.8 for "get" and 0.2 for "remove". Similarly, "set or remove" chooses between "set" (probability 0.3) and "remove" (probability 0.7).

On the other hand, the task ”handle notes” specifies a choice between "handle a displayed note" (probability 0.6) and "handle a carried note" (probability 0.4). Each of these two abstract tasks is composed of an application task which enables an abstract task. So, "handle a displayed note" cannot be executed if the application task "memoDisplayed" is not performed (similarly, for the task "handle a carried note"). As a result, four scenarios are possible for the task "handle notes":

- If both the application tasks ”memoDisplayed” and ”memoCarried” are available, then the user chooses between "handle a displayed note" and "handle a carried note" (with a probability of 0.6 and 0.4 respectively).
- If only the application task "memoDisplayed" is available, then the user must choose "handle a displayed note" and execute the task "get or remove".
- Similarly, if only the application task "memoCarried" is available, then the user must choose "handle a carried note" and execute the task "set or remove".
- If none of the two tasks ”memoDisplayed” and ”memoCarried” is available, then the user cannot perform the task "handle notes".

3 Extracting a formal model of the user behaviour

To make possible the automatic test data generation from task trees with operational profiles, formal semantics must be associated with the extended CTT syntax. This can be done in several ways, in particular using probabilistic extensions of Lotos [24, 14]. In this research work, the target model is a probabilistic input-output FSM, which is also a common model for reactive or interactive systems (see for example [2, 17, 16]). Such a model is exploitable in the Lutess testing environment as it is explained in section 4.

3.1 Preliminary definitions

For any task T, a probabilistic I/O machine

$M_T = (Q_T, qi_T, qf_T, I_T, O_T, trans_T, P_T)$

is defined, where:

- Q_T is a set of states
- qi_T is the initial state, the state where the task T starts. It is a source state (there is no transition from any state of the task T to its initial state).
- qf_T is the final state, the state where the task T ends. It is a sink state (there is no transition from the final state of a task to any state of this task).
- I_T is a set of application inputs for the task T.
- O_T is a set of application outputs for the task T.
- $trans_T \subseteq Q_T \times (I_T \cup \{\mu\}) \times O_T \times Q_T$ is the set of transitions corresponding to the task T. The input μ is an empty input (no user action). If $(q_T, a, b, s_T) \in trans_T$, we write $q_T \xrightarrow{a/b} s_T$. Sometimes, the input and the output of the transition are omitted: $q_T \xrightarrow{c} s_T$ (stands for $c = a/b$).
- P_T is the probability function: $P_T : trans_T \rightarrow [0..1]$ where the following property holds:

Property: For every state, the sum of the probabilities of the transitions leaving this state is equal to 1: $\forall q \in Q_T \setminus \{qf_T\} : \sum_{c,q'} P_T(q \xrightarrow{c} q') = 1$

As it has been mentioned in section 2, a task in the CTT notation can be a user task, an abstract task, an application task or an interactive task. User tasks are of no interest for test data generation, since they correspond to cognitive activities with no input sent to the system.

An application task o is supposed to be modeled by the machine $M_o =$

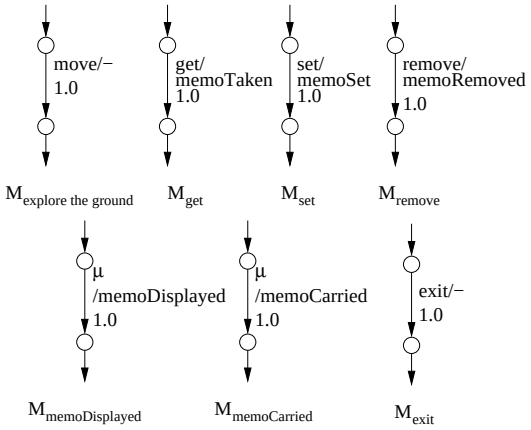

Figure 2. PFSM of tasks: *"explore the ground", "get", "set", "remove", "memoDisplayed", "memoCarried", "exit"*

$(Q_o, qi_o, qf_o, I_o, O_o, trans_o, P_o)$ where: $Q_o = \{qi_o, qf_o\}$, $I_o = \{\mu\}$, $O_o = \{o\}$, $trans_o = \{qi_o \xrightarrow{\mu/o} qf_o\}$, $P_o(qi_o \xrightarrow{\mu/o} qf_o) = 1$. In other words, an application task is considered as an elementary machine with two states, the unique transition of which consists in issuing an output.

Interactive tasks involve user actions and immediate feedback from the system. Test data generation is mainly concerned with those tasks. Interactive tasks are assumed to be modeled as I/O machines that must be provided at the beginning of the validation process.

The CTT of Figure 1 contains five interactive tasks: "get", "set", "remove", "explore the ground", "exit" and two application tasks: "memoDisplayed", "memoCarried". These tasks are modeled by probabilistic I/O machines, illustrated in Figure 2, where the interactive tasks are elementary interactions modeled by single transitions, the probability of which is *1.0*. For the interactive task "get", when the user gets a displayed note, the system displays the message "memo is taken" and the note disappears from the ground. Similarly, when the user removes a (carried or displayed) note, the system displays "memo is removed". When the user carries a note and issues the "set" command, the system displays "memo is set" and the carried note is dropped to the ground. Finally, when the user moves in order to explore the ground, there is no particular expected reaction of the system.

Remark 2 *The symbol "-" means "any reaction of the system".*

Finally, an abstract task is composed of other tasks combined by the various CTT operators. Probabilistic I/O machines (PFSM) can be associated with abstract tasks, resulting from the composition of their subtask PFSM, as it is explained in section 3.2.

The following notation is used:

- $(q \xrightarrow{c}_T q')_{p_T}$ for $(q \xrightarrow{c} q') \in trans_T$ and $P_T(q \xrightarrow{c} q') = p_T$
- $Q_T^{-fin} = Q_T \setminus \{qf_T\}$, $Q_T^{-init} = Q_T \setminus \{qi_T\}$, $Q_T^{-init-fin} = Q_T \setminus \{qi_T, qf_T\}$

Enabling operators ">>" and "[]>>" Consider three tasks *A, B* and *T*, such as : $T = A >> B$. Keeping in mind that the operator >> denotes that task A enables task B, while the operator [] >> means that task A provides some information to task B while it enables it, the same semantics are defined for the two operators >> , []>>: indeed, the information passing from A to B, when the operator []>> is used, is of no interest from the test data generation point of view, since this communication is internal to the application.

Since B starts when A terminates, the final state of A, qf_A, will be merged with the initial state of B, qi_B to a new state (qf_A, qi_B). The formal definition of the composition is:

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T, P_T) =$$
$$M(Q_A, qi_A, qf_A, I_A, O_A, trans_A, P_A) >> M(Q_B, qi_B, qf_B, I_B, O_B, trans_B, P_B)$$

$$Q_T = Q_A^{-fin} \cup Q_B^{-init} \cup \{(qf_A, qi_B)\}, qi_T = qi_A, qf_T = qf_B, I_T = I_A \cup I_B,$$
$$O_T = O_A \cup O_B$$

The transition relation $trans_T$ and the probability function p_T are defined as follows:

$(q \xrightarrow{c}_T s)_{p_T}$ if and only if one of the following holds:

- $(q \xrightarrow{c}_A s)_{p_A}, q, s \in Q_A^{-fin}, p_T = p_A$
- $(q \xrightarrow{c}_A qf_A)_{p_A}, q \in Q_A^{-fin}, s = (qf_A, qi_B), p_T = p_A$
- $(qi_B \xrightarrow{c}_B s)_{p_B}, q = (qf_A, qi_B), s \in Q_B^{-init}, p_T = p_B$
- $(q \xrightarrow{c}_B s)_{p_B}, q, s \in Q_B^{-init}, p_T = p_B$

Example: Consider the PFSM of the task : *"get or remove"* (or *"set or remove"*) illustrated in Figure 3. Applying the previous definition, this machine is composed with the machine of the task *"memoDisplayed"* (*"memoCarried"*) (c.f. Figure 2) to get the PFSM of task *handle a displayed note* =

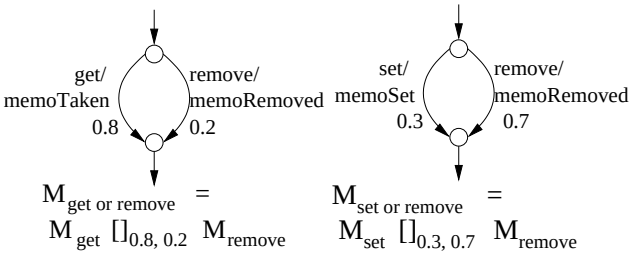

Figure 3. PFSM of "*get or remove*", "*set or remove*"

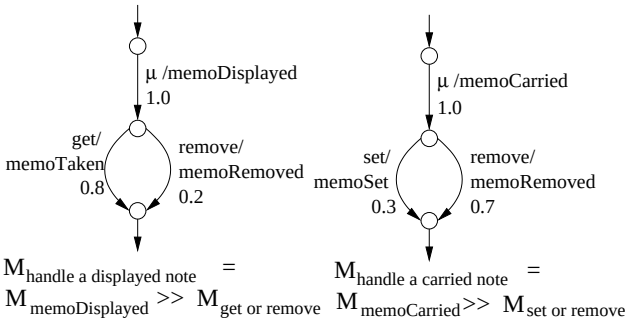

Figure 4. PFSM of tasks: *"handle a displayed note", "handle a carried note"*

$memoDisplayed >> get\,or\,remove$ ($handle\,a\,carried\,note = memoCarried >> set\,or\,remove$), provided in Figure 4.

Choice operator "[]" Consider three tasks A, B and T, such as: $T = A[]_{pr_A,pr_B}B$, where $pr_A + pr_B = 1$. Since the operator $[\,]$ denotes that task T is performed by choosing one task among A and B, T starts when either A or B begins, and it ends when the chosen task ends. So, the initial state of T, qi_T will be the combination of the two initial states qi_A and qi_B ($qi_T = (qi_A, qi_B)$). The final state of T, qf_T, will be also the combination of the two final states ($qf_T = (qf_A, qf_B)$). The transition probabilities of the resulting machine are computed as follows: for every state of the automaton resulting from the composition, if this state is not the initial state qi_T, then the probability value of the origin transition is preserved. Else, this probability is multiplied with the probability of the task automaton to which the transition belongs. The formal definition of the composition is:

$$
\begin{aligned}
&M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T, P_T) \\
&= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A, P_A)\ []_{pr_A,pr_B} \\
&\quad M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B, P_B)
\end{aligned}
$$

$$
\begin{aligned}
&Q_T = Q_A^{-init-fin} \cup Q_B^{-init-fin} \cup \{(qi_A, qi_B), (qf_A, qf_B)\}, \\
&qi_T = (qi_A, qi_B), qf_T = (qf_A, qf_B), I_T = I_A \cup I_B, O_T = O_A \cup O_B
\end{aligned}
$$

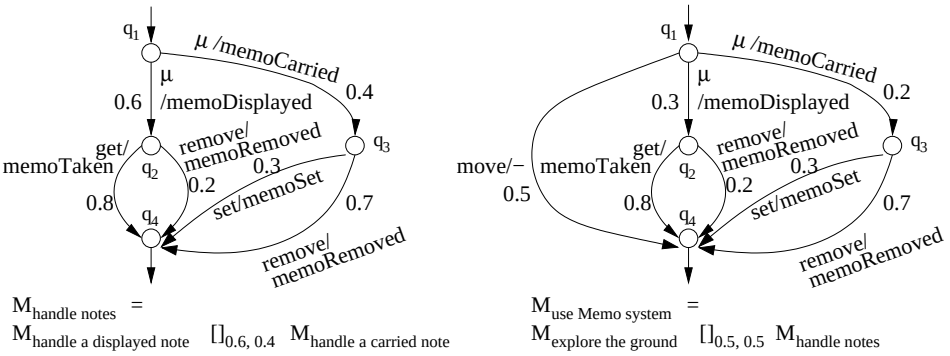

Figure 5. PFSM of *"handle notes"*, *"use memo system"*

The transition relation $trans_T$ and the probability function P_T are defined as follows:

$(q \xrightarrow{c}_T s)_{p_T}$ if and only if one of the following holds:

- $(qi_A \xrightarrow{c}_A s)_{p_A}$, $q = (qi_A, qi_B)$, $s \in Q_A^{-init-fin}$, $p_T = pr_A.p_A$
- $(q \xrightarrow{c}_A s)_{p_A}$, $q, s \in Q_A^{-init-fin}$, $p_T = p_A$
- $(q \xrightarrow{c}_A qf_A)_{p_A}$, $q \in Q_A^{-init-fin}$, $s = (qf_A, qf_B)$, $p_T = p_A$
- $(qi_A \xrightarrow{c}_A qf_A)_{p_A}$, $q = (qi_A, qi_B)$, $s = (qf_A, qf_B)$, $p_T = pr_A.p_A$
- $(qi_B \xrightarrow{c}_B s)_{p_B}$, $q = (qi_A, qi_B)$, $s \in Q_B^{-init-fin}$, $p_T = pr_B.p_B$
- $(q \xrightarrow{c}_B s)_{p_B}$, $q, s \in Q_B^{-init-fin}$, $p_T = p_B$
- $(q \xrightarrow{c}_B qf_B)_{p_B}$, $q \in Q_B^{-init-fin}$, $s = (qf_A, qf_B)$, $p_T = p_B$
- $(qi_B \xrightarrow{c}_B qf_B)_{p_B}$, $q = (qi_A, qi_B)$, $s = (qf_A, qf_B)$, $p_T = pr_B.p_B$

Examples: In the Memo example, applying the previous definition of the composition results in the PFSMs of $get \,[]_{0.8\,,0.2}\, remove$, $set \,[]_{0.3\,,0.7}\, remove$ (see Figure 3) from the PFSMs of the tasks "get", "set", "remove" illustrated in Figure 2.

Consider the PFSMs of tasks: *handle a displayed note*, *handle a carried note* in Figure 4. From these machines, applying the previous definition, the PFSM of the task $handle\,notes = handle\,a\,displayed\,note \;[]_{0.6,\,0.4} handle\,a\,carried\,note$ is built (see Figure 5-left). This machine is composed with the machine of the task *explore the ground* (c.f. Figure 2) ($use\,Memo\,system = explore\,the\,ground \;[]_{0.5,\,0.5} handle\,notes$) to get the PFSM illustrated in Figure 5 (right).

Iteration operator "$*$" Consider two tasks A and T, such as : $T = A^*$. In the probabilistic I/O machine corresponding to the repetitive task T, there are two types of transitions:

- $trans_{A'}$: transitions of M_A where the final state has been replaced by the initial state (iteration).

- $trans_{T'}$: transitions from the initial state qi_T, added to preserve the initial state of a task as a source state. In fact, because of the iteration, there are transitions to the state qi_A. The added state qi_T is a source state and the same actions, that can be executed from the state qi_A, can be also executed from the state qi_T.

The probability of a transition in the resulting machine is the same than in the original machine. The formal definition of the composition is:

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T, P_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_B, trans_A, P_A)^*$$

$$Q_T = Q_A^{-fin} \cup \{qi_T, qf_T\}, I_T = I_A, O_T = O_A, trans_T = trans_{A'} \cup trans_{T'}$$

where $trans_{A'} \subseteq Q_A^{-fin} \times I_A^{\mu} \times O_A \times Q_A^{-fin}$ and the values of the associated probabilities are defined as follows:

$(q_A \xrightarrow{a}_{A'} s_A)_{p_T}$ if and only if one of the following holds:

- $(q_A \xrightarrow{a}_A s_A)_{p_A}$, $p_T = p_A$
- $(q_A \xrightarrow{a}_A qf_A)_{p_A}$, $s_A = qi_A$, $p_T = p_A$

And $trans_{T'} \subseteq \{qi_T\} \times I_A^{\mu} \times O_A \times Q_A^{-fin}$ and the values of the associated probabilities are defined as follows:

$(qi_T \xrightarrow{a}_{T'} s_A)_{p_T}$ iff $(qi_A \xrightarrow{a}_{A'} s_A)_{p_T}$

Note that the final state qf_T of the repetitive task is not reachable because the iteration continues until the task is deactivated by another task.

Example: Figure 6 provides the PFSM of task $use\,Memo\,system*$.

Deactivation operator "[>" Consider three tasks A, B and T, such as: $T = A[>_{pr_{deac}} B$ where $pr_{deac} \leq 1$ which means that in every state of A, the probability for A to be interrupted by B is pr_{deac}. The operator [> denotes that B can deactivate A when the first action of B occurs. So, in the machine of T, M_T, from each state of M_A (excepted the final state qf_A) there is a transition labeled by the first action of task B towards the corresponding state in the machine M_B (when task B can start by one action among several actions, there will be a transition for every action). Since T ends when A has finished without

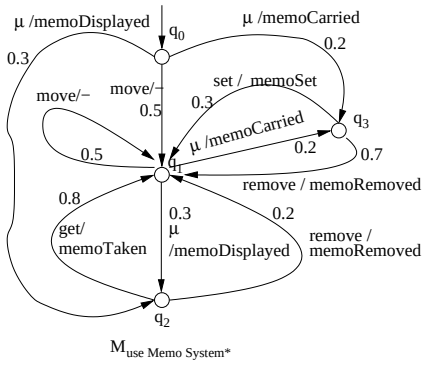

Figure 6. PFSM of task "use Memo system *"

interruption or when B has interrupted A and has finished, the final state of T, qf_T, will be the combination of the two final states ($qf_T = (qf_A, qf_B)$).

In the machine M_T, there are three sets of transitions:

- The set $trans_{A'}$ containing the transitions of M_A.
- The set $trans_{AB}$ containing the transitions corresponding to first actions of the machine M_B which can interrupt A.
- The set $trans_{B'}$ containing the transitions corresponding to the continuation of B after A interruption.

The transition probabilities of the resulting machine and the formal definition of the composition are computed as follows:

$$
\begin{aligned}
&M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T, P_T) \\
&= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A, P_A)[>_{pr_{deac}} \\
&\quad M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B, P_B)
\end{aligned}
$$

$$
\begin{aligned}
&Q_T = Q_A^{-fin} \cup Q_B^{-init-fin} \cup \{(qf_A, qf_B)\}, qi_T = qi_A, qf_T = (qf_A, qf_B), \\
&I_T = I_A \cup I_B, O_T = O_A \cup O_B, trans_T = trans_{A'} \cup trans_{AB} \cup trans_{B'}
\end{aligned}
$$

where $trans_{A'} \subseteq Q_A^{-fin} \times I_A^{\mu} \times O_A \times (Q_A^{-fin} \cup (qf_A, qf_B))$ and the associated values of P_T are defined as follows:

$(q_A \xrightarrow{a}_{A'} q)_{p_T}$ if and only if one of the following holds:

- $(q_A \xrightarrow{a}_A q)_{p_A}$, $q \in Q_A^{-fin}$, $p_T = (1 - pr_{deac}).p_A$
- $(q_A \xrightarrow{a}_A qf_A)_{p_A}$, $q = (qf_A, qf_B)$, $p_T = (1 - pr_{deac}).p_A$

$trans_{AB} \subseteq Q_A^{-fin} \times I_B^{\mu} \times O_B \times (Q_B^{-init-fin} \cup (qf_A, qf_B))$ and the associated values of P_T are defined as follows:

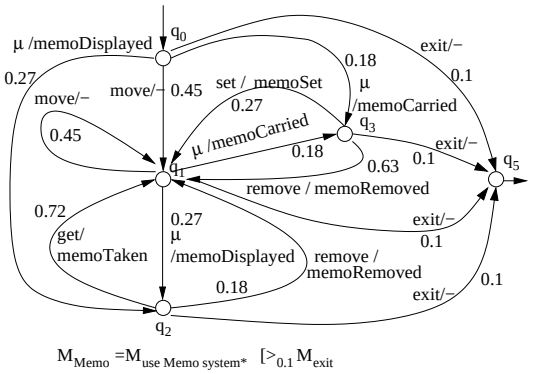

Figure 7. PFSM of task "*memo*"

$(q_A \xrightarrow{b}_{AB} q)_{p_T}$ if and only if one of the following holds:

- $(qi_B \xrightarrow{b}_B q)_{p_B}$, $q \in Q_B^{-init-fin}$, $p_T = pr_{deac}.p_B$
- $(qi_B \xrightarrow{b}_B qf_B)_{p_B}$, $q = (qf_A, qf_B)$, $p_T = pr_{deac}.p_B$

$trans_{B'} \subseteq Q_B^{-init-fin} \times I_B^{\mu} \times O_B \times (Q_B^{-init-fin} \cup (qf_A, qf_B))$ and the associated values of P_T are defined as follows:

$(q_B \xrightarrow{b}_{B'} q)_{p_T}$ if and only if one of the following holds:

- $(q_B \xrightarrow{b}_B q)_{p_B}$, $q \in Q_B^{-init-fin}$, $p_T = p_B$
- $(q_B \xrightarrow{b}_B qf_B)_{p_B}$, $q = (qf_A, qf_B)$, $p_T = p_B$

Example: Figure 7 provides the PFSM of task $Memo = use\,Memo\,system*\,[>_{0.1}\,exit$. In this machine, from every state of the PFSM of task $use\,Memo\,system*$ there is a transition labeled by "$exit/-$" (the first and the only action of task $exit$) to the final state with an interruption probability equal to *0.1*. The probabilities of the transitions of task $use\,Memo\,system*$ are multiplied by $0.9\,(= 1 - 0.1)$.

Independent concurrency operator "|||"

$$
\begin{aligned}
&M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T, P_T) \\
&= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A, P_A)|||_{pr_{ActA}, pr_{ActB}} \\
&\quad M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B, P_B)
\end{aligned}
$$

$$Q_T = Q_A \times Q_B, qi_T = (qi_A, qi_B), qf_T = (qf_A, qf_B),$$
$$I_T = I_A \cup I_B, O_T = O_A \cup O_B$$

The relation $trans_T$ and the probabilities of the transitions are defined as follows:

$((q_A, q_B) \xrightarrow{c}_T (s_A, s_B))_{p_T}$ if and only if one of the following holds:

- $(q_A \xrightarrow{c}_A s_A)_{p_A}$, $q_B = s_B \neq qf_B$, $p_T = pr_{ActA}.p_A$
- $(q_A \xrightarrow{c}_A s_A)_{p_A}$, $q_B = s_B = qf_B$, $p_T = p_A$
- $(q_B \xrightarrow{c}_B s_B)_{p_B}$, $q_A = s_A \neq qf_A$, $p_T = pr_{ActB}.p_B$
- $(q_B \xrightarrow{c}_B s_B)_{p_B}$, $q_A = s_A = qf_A$, $p_T = p_B$

Remark 3 *For the operator Concurrency with information exchange ($|[]|$), the information exchanged between the two tasks is of no interest from the test data generation point of view, so the same formal definition is adopted for the two operators $|||$ and $|[]|$.*

Finite iteration operator "(n)"

$$M_T(Q_T, qi_T, qf_T, I, O, trans_T) = M_A(Q_A, qi_A, qf_A, I, O, trans_A)(n)$$

$$Q_T = \{(q_A, i) \mid q_A \in Q_A^{-fin}, i \in [1..n]\} \cup \{qf_T\}$$
$$qi_T = (qi_A, 1)$$

where the relation $trans_T$ and the transition probabilities are defined as follows:

$((q_A, i) \xrightarrow{a}_T (s_A, j))_{p_T}$ if and only if one of the following holds:

- $(q_A \xrightarrow{a}_A s_A)_{p_A}$, $i = j$, $p_T = p_A$
- $(q_A \xrightarrow{a}_A qf_A)_{p_A}$, $j = i + 1$, $s_A = qi_A$, $p_T = p_A$

$((q_A, i) \xrightarrow{a}_T qf_T)_{p_T}$ if and only if $(q_A \xrightarrow{a}_A qf_A)_{p_A}$, $i = n$, $p_T = p_A$

Suspend-resume operator "$|>$"
$A|>_{pr_{sus}} B$, means that, in every state of A, the probability for B to suspend

A is pr_{sus}. We assume that A can be suspended several times by B. For example, the task "editing a text" can be suspended several times by the task "printing".

$$M_T(Q_T, qi_T, qf_T, I_T, O_T, trans_T, P_T)$$
$$= M_A(Q_A, qi_A, qf_A, I_A, O_A, trans_A, P_A)| >_{pr_{sus}}$$
$$M_B(Q_B, qi_B, qf_B, I_B, O_B, trans_B, P_B)$$

$$Q_T = \{qi_T\} \cup Q_A \cup (Q_A^{-fin} \times Q_B^{-init-fin}),$$
$$qf_T = qf_A, I_T = I_A \cup I_B, O_T = O_A \cup O_B$$
$$trans_T = trans_{A'} \cup trans_{AB} \cup trans_{B'} \cup trans_{T'}$$

where $trans_{A'} = trans_A$ and if $t \in trans_{A'}$ then $P_T(t) = (1 - pr_{sus}).P_A(t)$.

The transition relation $trans_{AB} \subseteq Q_A^{-fin} \times I_B^{\mu} \times O_B \times (Q_A^{-fin} \cup (Q_A^{-fin} \times Q_B^{-init-fin}))$ and the associated values of P_T are defined as follows (the transitions of this set correspond to the first actions of B that can suspend A):

$(q_A \xrightarrow{b}_{AB} (s_A, s_B))_{p_T}$ if and only if
$(qi_B \xrightarrow{b}_B s_B)_{p_B}$, $q_A = s_A$, $p_T = pr_{sus}.p_B$

$(q_A \xrightarrow{b}_{AB} s_A)_{p_T}$ if and only if
$(qi_B \xrightarrow{b}_B qf_B)_{p_B}$, $q_A = s_A$, $p_T = pr_{sus}.p_B$

The transition relation $trans_{B'} \subseteq (Q_A^{-fin} \times Q_B^{-init-fin}) \times I_B^{\mu} \times O_B \times (Q_A^{-fin} \cup (Q_A^{-fin} \times Q_B^{-init-fin}))$ and the associated values of P_T are defined as follows (the transitions of this set correspond to the continuation of B after A has been suspended):

$((q_A, q_B) \xrightarrow{b}_{B'} (s_A, s_B))_{p_T}$ if and only if
$(q_B \xrightarrow{b}_B s_B)_{p_B}$, $q_A = s_A$, $p_T = p_B$

$((q_A, q_B) \xrightarrow{b}_{B'} s_A)_{p_T}$ if and only if
$(q_B \xrightarrow{b}_B qf_B)_{p_B}$, $q_A = s_A$, $p_T = p_B$

The transition relation $trans_{T'} \subseteq \{qi_T\} \times I_T^{\mu} \times (Q_A \cup (\{qi_A\} \times Q_B^{-init-fin}))$ and the associated values of P_T are defined as follows (the transitions of this set start from the new initial state qi_T, which is added because the initial state of a task must be a source state. In fact, because of the suspend-resume operator,

there are transitions to the state qi_A. The added state qi_T is a source state and the same actions, that can be executed from the state qi_A, can be also executed from the state qi_T):

$(qi_T \xrightarrow{c}_{T'} s_A)_{p_T}$ if and only if one of the following holds:

- $(qi_A \xrightarrow{c}_{A'} s_A)_{p_T}$
- $(qi_A \xrightarrow{c}_{AB} qi_A)_{p_T}$, $s_A = qi_A$

$(qi_T \xrightarrow{b}_{T'} (qi_A, s_B))_{p_T}$ iff $(qi_A \xrightarrow{b}_{AB} (qi_A, s_B))_{p_T}$

Optional task The optional tasks must be used with the activation or the independent concurrency operators [15]:

Activation operator:

$$[A]_{pr_A} >> B = (A >> B)[]_{pr_A,(1-pr_A)}B$$

Independent concurrency operator:

$$[A]_{pr_A}|||_{pr_{Act[A]},pr_{ActB}}B$$
$$= (A|||_{pr_{Act[A]},pr_{ActB}}B)[]_{pr_A,(1-pr_A)}B$$

4 Model-based test generation

The PFSM obtained from the extended task tree is a model of the user behaviour. It expresses what are the possible user actions and how probable these actions are. Simulating this model results in generating test data for the interactive application. The main idea is to use this model to produce inputs "on the fly", while the interactive application is executed.

In section 4.1 it is shown how the PFSM should be simulated in theory for such an on-the-fly test generation dealing with the specified operational profiles while in section 4.2 it is shown that this model can be translated in an equivalent representation exploited by the Lutess testing environment, which has been used for a preliminary experimental evaluation of the approach.

The PFSM obtained in section 3, which is actually the test model, describes the interacting user behaviour (i.e. it is not a model of the application). We suppose that this model verifies the following properties:

- $\forall q$ such that $q \xrightarrow{i_1/o'} q'$, $q \xrightarrow{i_2/o"} q"$, $i_1 \neq \mu$, $i_2 \neq \mu$ this implies that $i_1 \neq i_2$.

- $\forall q$ such that $q \xrightarrow{\mu/o_1} q'$, $q \xrightarrow{\mu/o_2} q$", this implies that $o_1 \neq o_2$.

4.1 Generating tests from the test model

It is assumed that the PFSM is simulated while the interactive application under test is executed and that inputs and outputs are exchanged between them on-the-fly. During the simulation, assuming the PFSM to be in a given state, an input is chosen according to the probabilities of the outgoing transitions of this state. The chosen input is then sent to the interactive application, the resulting application outputs are read and the set of possible following states is computed (the current state may have several succesor states in only one case: The input is empty (μ) and the outputs of the application enable more than one transition). The next state is randomly chosen in this set according to the specified probabilities, and so on.

Formally, the simulation of the PFSM associated with an extended task tree is carried out by means of the following functions:

Definition: $beh_T : Q_T \longrightarrow 2^{I_T^\mu}$ where
$beh_T(q) = \{i \in I_T^\mu \,|\, \exists o, p,\ q \xrightarrow{i/o}_T p\}$ is the set of all valid inputs of the application in the state q.

Definition: $pTrans_T : Q_T \times I_T^\mu \times 2^{O_T} \longrightarrow 2^{Q_T}$ where
$pTrans_T(q, i, os) = \{p \,|\, q \xrightarrow{i/o}_T p\,, o \in os\}$ is the set of arrival states of transitions leaving the state q having an input i and of which the output is in os.

Remark 4 *When applying this function on the test model described in section 3, if the input i is not empty, there is at most one possible transition.*

Definition: A distribution of probabilities on a set A of elements, denoted by $ProbDist_A$ is a set of pairs $< el, pr >$ such that $el \in A$, pr is a real $\in [0..1]$, such as:

$$\sum_{<el,pr> \in ProbDist_A} pr = 1$$

If A is empty, then $ProbDist_A$ is also empty.

Algorithm 1

1.***var***
2. $preq, q \in Q_T$, $pFollowingq \in 2^{Q_T}$,
3. $i \in (I_T \cup \{\mu\})$, $oset \in 2^{O_T}$
4. ***begin***
5. $q \leftarrow qi_T$
6. ***while*** $(beh_T(q) \neq \emptyset)$
7. $oset \leftarrow \emptyset$
8. $i \leftarrow draw(ProbDistIn(q))$
9. ***if*** $(i \neq \mu)$ ***then*** $write(i)$
10. $wait(C)$
11. $read(oset)$
12. $preq \leftarrow q$
13. $pFollowingq \leftarrow pTrans_T(preq, i, oset)$
14. ***if*** $(pFollowingq \neq \emptyset)$
15. ***then***
16. $q \leftarrow draw(ProbDistTrans(preq, i, oset))$
17. ***else*** $q \leftarrow preq$
18. ***end while***
19. ***end***

Definition: If E is a set of elements and 2^E is the set of all the sub-sets of E, $ProbDistSet_{2^E}$ is the set of all possible distributions of probabilities on all the sub-sets of E. In other words, an element of $ProbDistSet_{2^E}$ is a distribution of probabilities $ProbDist_A$ on a set $A \in 2^E$.

Definition: $ProbDistIn : Q_T \longrightarrow ProbDistSet_{2^{I_T^\mu}}$ where:
$ProbDistIn(q) = ProbDist_{beh_T(q)} = \{< i, pr > | i \in beh_T(q), pr = \sum_{o,r} P_T(q \xrightarrow{i/o} r)\}$ is a distribution of probabilities of valid inputs in the state q.

Definition: $ProbDistTrans : Q_T \times I_T^\mu \times 2^{O_T} \longrightarrow ProbDistSet_{2^{Q_T}}$ where:
$ProbDistTrans(q, i, oset) = ProbDist_{pTrans_T(q,i,oset)} =$
$\{< q', pr > | q' \in pTrans_T(q, i, oset), pr = \frac{\sum_{o \in oset} P_T(q \xrightarrow{i/o} q')}{\sum_{r \in pTrans_T(q,i,oset), o \in oset} P_T(q \xrightarrow{i/o} r)}\}$
is the distribution of probabilities of possible following states for the state q, the input i and the set of outputs $oset$.

Definition: $draw : ProbDistSet_{2^E} \longrightarrow E$ where:
$draw(ProbDist_A)$ returns an element of A with respect to the distribution of probabilities $ProbDist_A$.

The algorithm 1 performs the test data generation. In line 8, an input is chosen according to the probabilities of the transitions leaving the current state (set to the initial state in line 5). This input (if it is not empty) is sent to the interactive application (line 9). Then, the generator waits for the reaction of the application (line10) and the outputs are read (line 11). The set of possible following states is computed (line 13) and a state is chosen according to the distribution of the associated transition probabilities (line 16). Indeed, assume that from the current state s the empty input μ is chosen (the only case where we can have more than one following states) and the transitions $t_1 = (s \xrightarrow{\mu/o_1} s_1)_{pr1}$, $t_2 = (s \xrightarrow{\mu/o_2} s_2)_{pr2}$, ... can be fired : If the application issues o_1 only (or o_2 only) then the transition to s_1 (to s_2) is chosen. But, if the application sends o_1 and o_2 in the same execution cycle, then the new state is chosen according to the probabilities of the transitions $prob(s_1) = \frac{pr_1}{pr_1+pr_2}$, $prob(s_2) = \frac{pr_2}{pr_1+pr_2}$.

Consider the PFSM of Figure 6 and suppose that the current state is q_1 and the chosen input is μ, four test scenarios are possible according to the application behaviour:

- If the application displays a note (*memoDisplayed*) and the user carries a note (*memoCarried*). In this case, two transitions can be fired, to states q_2 and q_3. The probability of the transition to q_2 is $\frac{0.3}{0.3+0.2} = 0.6$ and the probability of the transition to q_3 is $\frac{0.2}{0.3+0.2} = 0.4$.
- If the application displays a note (*memoDisplayed*) and the user does not carry a note, only one transition can be fired, to state q_2.
- Similarly, if the application does not display a note and if the user carries a note (*memoCarried*), only one transition can be fired, to state q_3.
- If the application does not display a note and the user does not carry a not, no transition can be fired.

4.2 Experimental evaluation with Lutess

4.2.1 Lutess overview

Lutess [8] is an environment initially designed for testing synchronous software. Lutess requires a test model, including a specification of the software external environment (i.e. input variables behaviour) as a set of invariant properties and operational profiles [25]. From this non deterministic specification, Lutess builds a generator of test data: at each step, the generator draws a valid vector of inputs conforming the environment specification and sends it to the system under test which reacts with an output vector and feeds back the generator with it. The cycle is repeated while an automatic oracle observes the program inputs and outputs. Valid inputs are selected randomly or in conformance to

the specified operational profiles.

The specification language of Lutess is an extension of Lustre, a synchronous declarative data-flow language [11]. Within Lustre, any variable or expression represents an infinite sequence of values and takes its n-th value at the n-th cycle of the program execution. Lustre offers usual arithmetic, boolean and conditional operators and two specific operators: The (`pre`) operator which refers to the "*previous*" value of an expression, and the "*followed-by*" (`->`) operator which is used to set the initial value of a flow. Let `E` and `F` be two expressions of the same type denoting the sequences of values $(e_0, e_1, ..., e_n, ...)$ and $(f_0, f_1, ..., f_n, ...)$; f_i is the value of `F` at instant i. Then `pre(E)` denotes the sequence $(nil, e_0, e_1, ..., e_n, ...)$ where nil is an undefined value; while `E->F` denotes the sequence $(e_0, f_1, ..., f_n, ...)$. A Lustre program is structured into nodes. A node is a set of equations which define the node's outputs as a function of its inputs. Once a node is defined, it can be used inside other nodes like any other operator. Lustre is an executable specification language, which also provides the main characteristics of a linear temporal logic of the past [12]. Therefore, temporal logic formulas can be easily implemented as Lustre programs. The user can define her/his own logical or temporal operators to express invariants or properties.

Within Lutess, the test model is specified in a special Lustre node called *testnode*. A testnode has as inputs (resp. outputs) the outputs (resp. inputs) of the software under test. The general form of a testnode is given in Figure 8. There are four operators specifically introduced for testing purposes:

- The `environment` operator makes it possible to specify invariant properties of the program environment.
- The `prob` operator is used to create operational profiles where the selection of the program inputs is performed with respect to probabilities specified by the tester. `prob(C,E,P)` means:
  · `C` is a condition relating to the past values of the input/ output parameters,
  · `E` is an expression returning a boolean value,
  · `P` is a real constant in the interval [0.0..1.0],
  · if the condition `C` holds, then the probability for `E` to hold is equal to `P`.
- The `safeprop` operator is used for safety property guided testing (which leads the test generation towards situations that could violate the program properties). In addition, the `hypothesis` operator is used for specifying hypotheses on the program under test in order to ease the computation of test data for safety property guided testing.

```
testnode Env(<SUT outputs>) returns (<SUT inputs>);
var <local variables>;
let
   environment(Ec1,Ec2, ....,Ecn);
   prob(C1,E1, P1);
   ...
   prob(Cm,Em, Pm);
   safeprop(Sp1, Sp2, ...., Spk);
   hypothesis(H1,H2, ....,Hl);
   <definition of local variables>;
tel;
```

Figure 8. Testnode syntax

4.2.2 Automatic generation of the Lutess test model

In this section it is shown how the Lutess test model (testnode) associated with a task tree can be automatically built. For sake of clarity, this construction is presented in two steps. In the first step it is shown how the FSM associated with the task tree is represented without any probability considerations. Probability assignments are introduced in the second step.

Building the basic test model

Consider the FSM $(Q_T, q_0, I_T, O_T, Trans_T)$ ($Q_T = \{q_0, q_1, ..., q_n\}$ is the state set, q_0 is the initial state, $I_T = \{i_0, i_1, ..., i_n\}$ is the input set, $O_T = \{o_0, o_1, ..., o_n\}$ is the output set, $Trans_T$ is the transition set) corresponding to a task tree T. It is assumed that the root task of this tree is an iterative task, then the final state is not reachable (this means that the interactive system is executed until the user performs an exit action).

The inputs I of the interactive application are the outputs of the testnode and, conversely, its outputs O will be the inputs of the testnode. These inputs/outputs are defined as boolean parameters: A true value for an input/output means that this input/output occurs.

`testnode CTT_T(`o_0`, `o_1`,..., `o_n`: bool)returns (`i_0`, `i_1`,..., `i_n`: bool);`

Boolean local variables (`$q_0$`, `$q_1$`, ..., `$q_n$`) represent the effective state of the testnode, in addition to other variables (`$pq_0$`, `$pq_1$`, ..., `$pq_n$`) that abstract the possible next states at a given instant. A local variable `mu` is used to define the empty input μ (false valuation for all the input variables).

```
var
-- effective state
q0, q1, ..., qn: bool;
-- possible next states
```

```
pq0, pq1, ..., pqn: bool;
-- variable defining the empty input μ
mu: bool;


let
-- modeling the empty input μ
mu = not (i0 or i1 or ... or in);
```

For every state q_k, a boolean variable pq_k is defined, true when a transition to the state q_k is possible from the current state.

```
-- modeling that qk can be the next state:
-- qh --ih/oh--> qk, qj --ij/oj--> qk, ...,
-- qk --il/ol--> ql, qk --im/om--> qm, ...
pqk = false->
pre (qh and ih and oh)
or pre (qj and ij and oj)
or ...
or (pre (qk
and not (il and ol)
and not (im and om)
and not ... ));
```

The initial state q_0 is a source state:

```
-- q0 --ih/oh--> qh, q0 --ij/oj--> qj, ...
pq0 = true->
pre (q0
and not (ih and oh)
and not (ij and oj)
and not ... );
```

The current state is randomly set by assigning values to the variables (q_0, q_1, ..., q_n). At a given instant, q_0 can be active (q_0 = true) only if there is a possible transition to this state (pq_0 = true):

```
environment (implies (q0, pq0)
and ...
and implies (qn, pqn));
```

There is exactly one active state at the same time:

```
environment (#(q0, ..., qn) [1]
```

[1] The (#) operator means that at most one of the parameters is true at a given step

and (q_0 or ... or q_n));

As every transition of the FSM issued from the task tree is labeled by one input, at most one input is active, at every step:

environment (# (i_0, i_1, ...,i_n));

The inputs are produced in conformance to the current state.

-- $q_k \xrightarrow{i_l/o_l} q_l$,..., $q_k \xrightarrow{i_m/o_m} q_m$, ...
environment(if q_k then (i_l or ... or i_m) else if q_h then ...);

For the example of Memo, considering the FSM of the Figure 6, the resulting testnode is illustrated in Figure 9.

Adding operational profiles

Consider the PFSM $(Q_T, q_0, I_T, O_T, Trans_T, P_T)$ corresponding to a probabilistic task tree T. Assuming the current state to be q_k, the valid inputs are those labeling the transitions leaving this state. By using the function $probDistIn$ defined in the section 4.1, the probability of each input to be generated can be computed. If in the state q_k, the valid inputs are: i_l, i_m, ... and $probDistIn(q_k) = \{< i_l, pr_l >, < i_m, pr_m >, ...\}$ then the following statements are added in the testnode:

$$\begin{aligned} &prob\ (pq_k,\ i_l,\ \ pr_l);\\ &prob\ (pq_k,\ i_m,\ pr_m); \end{aligned} \tag{1}$$

Indeed, a state can have more than one possible successor states (more than one pq_i is set to true) when the empty input (μ) is chosen. For instance, in the Memo example of paragraph 4.1, there are two possible transitions from q_1 with the empty input μ: one to q_2 and another to q_3 when $memoDisplayed$ and $memoCarried$ occur in the same time. In this case, the operational profile information is taken into account to choose the next state.

For every state q_h with transitions labelled with the empty input (μ), the successor states are computed as follows:

$$suc(q_h, i) = \{q_k | \exists o,\ q_h \xrightarrow{i/o} q_k\} \tag{2}$$

of computation.

```
testnode CTT_Memo
(memoDisplayed, memoCarried, memoTaken, memoSet, memoRemoved :bool)
returns (move, get,set, remove :bool);
var
   pq0, pq1, pq2, pq3 : bool;
   q0, q1, q2, q3 : bool;
   mu: bool;
let
   mu = not (move or get or set or remove);
   pq0 =true ->pre (q0 and not (move)
        and not (memoDisplayed and mu)
        and not (memoCarried and mu));
   pq1 = false ->pre (q0 and move)
     or pre (q1 and move)
     or pre (q2 and get and memoTaken)
     or pre (q2 and remove and memoRemoved)
     or pre (q3 and set and memoSet)
     or pre (q3 and remove and memoRemoved)
     or pre (q1 and not move
           and not (mu and memoDisplayed)
           and not (mu and memoCarried));
   pq2 = false ->
          pre (q1 and mu and memoDisplayed)
       or pre (q0 and mu and memoDisplayed)
       or pre (q2
          and not (get and memoTaken)
          and not(remove and memoRemoved));
   pq3 = false ->
           pre (q1 and mu and memoCarried)
        or pre (q0 and mu and memoCarried)
        or pre (q3
          and not (set and memoSet)
          and not(remove and memoRemoved));
   environment ( implies(q0, pq0) and
      implies(q1, pq1) and implies(q2, pq2)
      and implies(q3, pq3));
   environment (#(q0, q1, q2, q3)
             and (q0 or q1 or q2 or q3));
   environment (#(move, get,set, remove));
   environment (if q0 then (move or mu)
         else if q1 then (move or mu)
         else if q2 then (get or remove)
         else (set or remove)); -- state q3
tel
```

Figure 9. The testnode for Memo

For the example of Memo, the successors are:

$suc(q_0, move) = \{q_1\}$, $suc(q_0, \mu) = \{q_2, q_3\}$
$suc(q_1, move) = \{q_1\}$, $suc(q_1, \mu) = \{q_2, q_3\}$
$suc(q_2, get) = \{q_1\}$, $suc(q_2, remove) = \{q_1\}$
$suc(q_3, set) = \{q_1\}$, $suc(q_3, remove) = \{q_1\}$

This function makes possible to compute the states which can be simultaneously active: $\{q_2, q_3\}$). States q_1 and q_0 can only be active alone.

The probabilities of the inputs leaving q_0 and q_1 are specified in Lutess as explained above (c.f. formula 1):

```
prob (pq0, move , 0.5);
prob (pq0, mu, 0.5);
prob (pq1, move , 0.5);
prob (pq1, mu, 0.5);
```

In the case of more than one successor states, the Lutess generator has to choose one state according to the operational profile specification. To do so, the function *suc* is used (c.f. equation 2). Assuming the cardinality of $suc(q_h, i_h)$ is more than one, several cases are possible during the execution according to the reaction of the interactive application under test:

- one transition from q_h to a state $q_k \in suc(q_h, i_h)$.
- more than one transitions from q_h to some states in $suc(q_h, i_h)$.

So, the set of all the subsets of $suc(q_h, i_h)$ is considered and for every not empty subset ($\forall Q \in (2^{suc(q_h, i_h)} \setminus \{\emptyset\})$) the following condition can hold during the execution:

C_Q `= false -> pre` q_h `and (`$\forall q_k \in Q$ `:` pq_k`) and (`$\forall q_j \in Q'$ `: not` pq_j`)`

where $Q' = suc(q_h, i_h) \setminus Q$ is the complement of Q.

For Memo, $suc(q_1, \mu) = \{q_2, q_3\}$ and $2^{\{q_2, q_3\}} = \{\{q_2\}, \{q_3\}, \{q_2, q_3\}, \emptyset\}$, so the following conditions can hold during the execution:

- $C_{\{q_2\}}$ `= false -> pre q1 and pq2 and not pq3`
- $C_{\{q_3\}}$ `= false -> pre q1 and pq3 and not pq2`
- $C_{\{q_2, q_3\}}$ `= false -> pre q1 and pq2 and pq3`

For a given condition C_Q, since one state has to be chosen in Q, a probability is specified as follows:

For all $q_k \in Q$ the probability to reach q_k is:

$pr_{q_k} = \frac{Pr(q_h,i_h,q_k)}{Pr(q_h,i_h,Q)}$ where:

- $Pr(q_h, i_h, q_k) = \sum_o P_T(q_h \xrightarrow{i_h/o} q_k)$ is the probability of the transitions from q_h to q_k.
- $Pr(q_h, i_h, Q) = \sum_{q \in Q} Pr(q_h, i_h, q)$ is the sum of the probabilities of the transitions from q_h to all states in Q.

For every $q_k \in Q$, given the function $probDistIn(q_k) = \{< i_l, pr_l >, < i_m, pr_m >, ...\}$ (which specifies the probabilities of valid inputs in q_k), probabilities for Lutess can be specified as follows:

$$\begin{aligned} &prob\ (C_Q,\ q_k\ and\ i_l,\ \ pr_{q_k} * pr_l); \\ &prob\ (C_Q,\ q_k\ and\ i_m,\ pr_{q_k} * pr_m); \\ &... \end{aligned} \tag{3}$$

In the Memo example, when the condition $C_{\{q_2,q_3\}}$ holds, the probabilities to choose the states q_2, q_3 are respectively:

- $pr_{q_2} = \frac{pr(q_1,\mu,q_2)}{pr(q_1,\mu,q_2)+pr(q_1,\mu,q_3)} = \frac{0.3}{0.3+0.2} = 0.6$
- $pr_{q_3} = \frac{pr(q_1,\mu,q_3)}{pr(q_1,\mu,q_2)+pr(q_1,\mu,q_3)} = \frac{0.2}{0.3+0.2} = 0.4$

The corresponding Lutess specification is the following:

```
prob (false -> pre q1 and pq2 and pq3, q2 and get, 0.6 * 0.8);
prob (false -> pre q1 and pq2 and pq3, q2 and remove, 0.6 * 0.2 );

prob (false -> pre q1 and pq2 and pq3, q3 and set, 0.4 * 0.3 );
prob (false -> pre q1 and pq2 and pq3, q3 and remove, 0.4 * 0.7);
```

The whole probability specification for Memo is given in Figure 10.

Table 2 shows an extract of the execution trace resulting from a test operation. It can be observed that when a note is displayed the user prefers doing "get" than "remove", and when a note is carried the user prefers doing "remove" than "set". When there is a note displayed and a note carried the user prefers handling the displayed note by taking this note. The user has no preference between moving and handling notes. This behaviour conforms to the operational profile of Memo user of Figure 1.

```
prob (pq0, move , 0.5);
prob (pq0, mu, 0.5);
prob (pq1, move , 0.5);
prob (pq1, mu, 0.5);
-- Choosing current state and input according to operational profile
information when in-determinism can occur:
```
– suc(q1,mu), $C_{\{q2\}}$
```
prob (false -> pre q1 and pq2 and not pq3, get, 0.8);
prob (false -> pre q1 and pq2 and not pq3 , remove, 0.2);
```
– suc(q1,mu), $C_{\{q3\}}$
```
prob (false -> pre q1 and pq3 and not pq2, set, 0.3);
prob (false -> pre q1 and pq3 and not pq2, remove, 0.7);
```
– suc(q1,mu), $C_{\{q2,q3\}}$
```
prob (false -> pre q1 and pq2 and pq3, q2 and get, 0.48);
prob (false -> pre q1 and pq2 and pq3, q2 and remove, 0.12);
prob (false -> pre q1 and pq2 and pq3, q3 and set, 0.12);
prob (false -> pre q1 and pq2 and pq3, q3 and remove,0.28);
```
– suc(q0,mu), $C_{\{q2\}}$
```
prob (false -> pre q0 and pq2 and not pq3, get, 0.8);
prob (false -> pre q0 and pq2 and not pq3 , remove, 0.2);
```
– suc(q0,mu), $C_{\{q3\}}$
```
prob (false -> pre q0 and pq3 and not pq2, set, 0.3);
prob (false -> pre q0 and pq3 and not pq2, remove, 0.7);
```
– suc(q0,mu), $C_{\{q2,q3\}}$
```
prob (false -> pre q0 and pq2 and pq3, q2 and get, 0.48);
prob (false -> pre q0 and pq2 and pq3, q2 and remove, 0.12);
prob (false -> pre q0 and pq2 and pq3, q3 and set, 0.12);
prob (false -> pre q0 and pq2 and pq3, q3 and remove,0.28);
```

Figure 10. the probability specification of Memo for Lutess

5 Conclusion and future work

Model-based testing of interactive applications has been studied for several years. Naturally, adapting models used in reactive system verification seems a suitable approach, as it has been shown, for instance, in [17, 16, 4, 27, 6]. But the corresponding notations are not common in the human-computer interaction domain. In this article, it is proposed to use an extended version of a well-known notation in interactive application design, task trees (and more precisely, CTT). Task trees are enhanced with operational profiles to make possible the definition of various interaction scenarios. Then, adequate formal semantics are defined for the CTT operators making possible to translate a task tree into a probabilistic input-output FSM modeling the user behaviours. Such a model can be used for automatic test data generation either by means of an ad hoc generator, either using already existing tools, such as the Lutess testing environment, as illustrated in section 4.2.

move	get	set	remove	mDis	mCar	mTak	mSet	mRem
0	0	0	0	**1**	0	0	0	0
0	**1**	0	0	0	1	1	0	0
0	0	0	0	0	**1**	0	0	0
0	0	0	**1**	0	0	0	0	1
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0
0	0	0	0	**1**	0	0	0	0
0	**1**	0	0	0	1	1	0	0
0	0	0	0	0	**1**	0	0	0
0	0	0	**1**	0	0	0	0	1
1	0	0	0	1	0	0	0	0
0	0	0	0	**1**	0	0	0	0
0	**1**	0	0	0	1	1	0	0
1	0	0	0	1	1	0	0	0
0	0	0	0	**1**	**1**	0	0	0
0	**1**	0	0	0	1	1	0	0
1	0	0	0	1	1	0	0	0
1	0	0	0	1	1	0	0	0
0	0	0	0	**1**	**1**	0	0	0
0	**1**	0	0	0	1	1	0	0

Table 2
An extract of the execution trace of Memo where
`mDis: memoDisplayed, mCar: memoCarried, mTak: memoTaken, mSet: memoSet, mRem: memoRemoved`

There are several perspectives for future work. In terms of test modeling, user-defined operational profiles could be improved to optimize the probability to find errors, as it is suggested in [17]. Moreover, properties that the interactive application should verify could be specified and, then, used for property-guided testing (for instance using the Lutess features).

References

[1] Y. Aït Ameur, M. Baron, and P. Girard. Formal validation of HCI user tasks. In *the International Conference on Software Engineering Research and Practice, SERP '03, Volume 2*, pages 732–738, Las Vegas, Nevada, USA, 23 - 26 June 2003.

[2] César Andrés, Luis Llana, and Ismael Rodriguez. Formally comparing user and implementer model-based testing methods. In *4th Workshop on Advances in Model Based Testing (A-MOST)*, Lillehammer, Norway, 2008.

[3] Jullien Bouchet and Laurence Nigay. ICARE: a component-based approach for the design and development of multimodal interfaces. In *Extended abstracts of the 2004 Conference on Human Factors in Computing Systems, CHI 2004*, pages 1325–1328, Vienna, Austria, 24 - 29 April 2004.

[4] Jullien Bouchet, Laya Madani, Laurence Nigay, Catherine Oriat, and

Ioannis Parissis. Formal testing of multimodal interactive systems. In *Engineering Interactive Systems, EIS'2007*, Salamanca, Espagne, March 2007.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and J. A. Plaice. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages (POPL)*, pages 178–188, 1987.

[6] Bruno d'Ausbourg. Using model checking for the automatic validation of user interface systems. In *Design, Specification and Verification of Interactive Systems'98, Proceedings of the Fifth International Eurographics Workshop*, pages 242–260, Abingdon, United Kingdom, 3-5 June 1998.

[7] A. Dittmar. More precise descriptions of temporal relations within task models. In *Interactive Systems: Design, Specification, and Verification, 7th International Workshop DSV-IS, Proceedings*, pages 151–168, Limerick, Ireland, 5-6 June 2000.

[8] L. du Bousquet, F. Ouabdesselam, I. Parissis, J. L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool Support for System Specification, Development and Verification*, pages 48–61. Advances in Computer Science. Springer Verlag, June 1998.

[9] D. J. Duke and M. D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.

[10] G. P. Faconti, N. Zani, and F. Paternò. The input model of standard graphics systems revisited by formal specification. *Computer Graphics Forum*, 11(3):237–251, 1992.

[11] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language lustre. *IEEE Transactions on Software Engineering (TSE)*, 18(9), 1992.

[13] Francis Jambon, Patrick Girard, and Yohann Boisdron. Dialogue validation from task analysis. In *Design, Specification and Verification of Interactive Systems'99, Proceedings of the Eurographics Workshop in Braga, Portugal*, pages 205–224. Springer, 2-4 June 1999.

[14] Joost Pieter Katoen, Rom Langerak, and Diego Latella. Modeling systems by probabilistic process algebra: an event structures approach. In *Formal Description Techniques, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques - FORTE '93*, pages 253–268, Boston, MA, USA, 26-29 October 1993.

[15] Andreas Lecerof and Fabio Paternò. Automatic support for usability evaluation. *IEEE Transactions on Software Engineering (TSE)*, 24(10): 863–888, 1998.

[16] Luis Fernando Llana-Díaz, Manuel Núñez, and Ismael Rodríguez. Customized testing for probabilistic systems. In *18th International Confer-*

ence on Testing Communicating Systems, TestCom'06, volume 3964 of *LNCS*, pages 87–102. Springer, 2006.

[17] Luis Fernando Llana-Díaz, Manuel Núñez, and Ismael Rodríguez. Derivation of a suitable finite test suite for customized probabilistic systems. In *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference*, volume 4229 of *Lecture Notes in Computer Science*, pages 467–483, Paris, France, 26-29 September 2006. Springer.

[18] Laya Madani and Ioannis Parissis. Automated test of interactive applications using task trees. In *4th Workshop on Advances in Model-based Testing (A-MOST)*, Lillehammer, Norway, 2008.

[19] Laya Madani, Catherine Oriat, Ioannis Parissis, Jullien Bouchet, and Laurence Nigay. Synchronous testing of multimodal systems: An operational profile-based approach. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005)*, pages 325–334, Chicago, IL, USA, 8-11 November 2005.

[20] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering (TSE)*, 27(2), 2001.

[21] G. Mori, F. Paternò, and C. Santoro. CTTE: Support for developing and analyzing task models for interactive system design. *IEEE Transactions on Software Engineering (TSE)*, 28(8):797–813, 2002.

[22] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, March 1993.

[23] D. Navarre, Ph. A. Palanque, R. Bastide, A. Schyn, M. Winckler, L. Porcher Nedel, and C. M. Dal Sasso Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In *Human-Computer Interaction - INTERACT 2005, IFIP TC13 International Conference*, pages 170–183, Rome, Italy, 12-16 September 2005.

[24] Manuel Núñez and David de Frutos-Escrig. Testing semantics for probabilistic LOTOS. In *Formal Description Techniques VIII, Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques*, pages 367–382, Montreal, Canada, October 1995.

[25] Farid Ouabdesselam and Ioannis Parissis. Constructing operational profiles for synchronous critical software. In *6th International Symposium on Software Reliability Engineering*, pages 286–293, Toulouse, France, October 1995.

[26] Ph. A. Palanque and R. Bastide. Verification of an interactive software by analysis of its formal specification. In *Human-Computer Interaction, INTERACT '95, IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 191–196, Lillehammer, Norway, 27-29 June 1995.

[27] Richard K. Shehady and Daniel P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *27th International Symposium on Fault-Tolerant Computing, FTCS '97*, June 1997.