Timed Use Case Maps

Jameleddine Hassine[1], Juergen Rilling[1], and Rachida Dssouli[2]

[1] Department of Computer Science, Concordia University, Montreal , Canada
{j_hassin,rilling}@cse.concordia.ca
[2] Concordia Institute for Information Systems Engineering, Montreal , Canada
dssouli@ece.concordia.ca

Abstract. Scenario-driven requirement specifications are widely used to capture and represent functional requirements. Use Case Maps are being standardized as part of the User Requirements Notation (URN), the most recent addition to ITU-T's family of languages. UCM models focus on the description of functional requirements and high-level designs at early stages of the development process. How a system is executed over time and how this may affect its correctness and performance, however, are introduced later in the development process which may require considerable changes in design or even worse at the requirement analysis level. We believe that timing aspects must be integrated into the system model, and this must be done already at an early stage of development. This paper introduces an approach to describe timing constraints in Use Case Maps specifications. We present a formal semantics of Timed UCM in terms of Clocked Transition Systems (CTS). We illustrate our approach using an example of a simplified wireless system.

Key words: Use Case Maps, User Requirements Notation, timing aspects, performance, timed UCM, Clocked Transition Systems.

1 Introduction

In the early stages of common development processes, system functionalities are defined in terms of informal requirements and visual descriptions. Scenario-driven approaches, although often semiformal, are widely accepted because of their intuitive syntax and semantics. These approaches focus mainly on the description of system functionalities and little attention has been given so far to modeling time and performance aspects. These timing and performance issues are often overlooked during the initial system design. They are typically regarded as separate behaviour issues and therefore described in separate models. In recent years there has been a growing interest in integrating these aspects into a unified framework. This integration was mainly driven by the fact that time, performance, and behaviors are tightly related in embedded real-time systems, affecting directly both, functional and non-functional requirements.

Use Case Maps (UCMs) [15], a scenario based language that has gained momentum in recent years within the software requirements and specification community, has been successfully used in describing real-time systems, with a particular focus on telecommunication system and services[3, 4, 7, 19]. Use Case Maps (UCMs), part of a new

proposal to ITU-T for a User Requirements Notation (URN) [14], can be applied to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a higher level of abstraction, and to provide the stakeholders with guidance and reasoning about the system-wide architecture and behavior. UCM is not intended to replace UML, but rather complement it and help to bridge the modeling gap between requirements (use cases) and design (system components and behavior).

UCM abstract syntax and static semantics are informally defined in document Z.152 [15]. In a recent work, we have proposed an operational semantics for the UCM language based on Multi-Agent Abstract State Machines [10]. This ASM model provides a concise semantics of UCM functional constructs and describes precisely the control semantics. Another formalization attempt was presented in [5] where UCM constructs are translated into the formal language LOTOS.

The original UCM notation presented in [15] does not describe semantics involving time, allowing for different interpretations of timing information, such as the time needed for a transition or a responsibility to complete. To date, these issues remain unexplored.

In this work, we extend the Use Case Maps notation with timing information. We define a formal syntax and semantics of timed UCM models based on Clocked Transition Systems [18]. Clocked Transition Systems were introduced as a formal notation to model the behavior of real-time systems. Its definition provides a simple way to annotate state-transitions graphs with timing constraints using finitely many real-valued clock variables. The goal of our semantics is to support the execution and the analysis of timed UCM specifications. This paper is part of the ongoing research towards using UCM to describe, simulate, and verify real-time systems.

In an attempt to make this paper self-contained, we include some of the core background information relevant to this research. In the next section, we provide an overview of the un-timed Use Case Maps notation along with an example that is used throughout the paper. In Section 3, we present the syntax of timed UCM. Section 4 provides the formal semantics of Timed UCM in terms of Clocked Transition Systems (CTS). Section 5 presents the state of the art in describing timing semantics for modeling languages. Finally, Section 6 concludes with a brief discussion and future work.

2 Use Case Maps

The Use Case Maps notation [15] is a high level scenario based modeling technique, used to specify functional requirements and high-level designs for various reactive and distributed systems. A UCM model depicts scenarios as causal flows of responsibilities (e.g. operation, action, task, function, etc.) that can be superimposed on underlying structures of components. Components are generic and can represent software entities (objects, processes, databases, servers, etc.) as well as non-software entities (e.g. actors or hardware). These relationships are said to be causal because they involve concurrency, partial ordering of activities, and they link causes (e.g., preconditions and triggering events) to effects (e.g. post-conditions and resulting events).With the UCM notation, scenarios are expressed above the level of messages exchanged between components, hence, they are not necessarily bound to a specific underlying structure (such

UCMs are called Unbound UCMs). One of the strengths of UCMs is their ability to integrate a number of scenarios together (in a map-like diagram), and to reason about the architecture and its behavior over a set of scenarios.

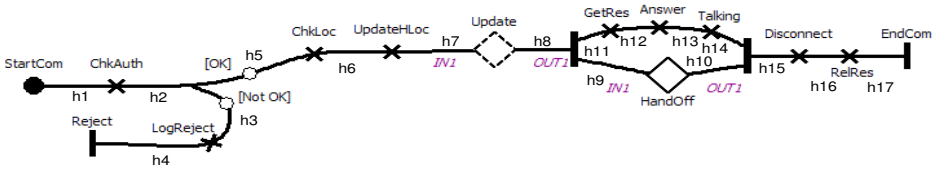


Fig. 1. Root Map for the simplified Wireless System

Figure 1 illustrates some of the basic UCM concepts using a modified version of a simplified wireless system that was initially introduced in [2]. The root map in Figure 1 describes a scenario where a mobile station initiates a call then proceeds with a handoff. Filled circles represent start points, which capture preconditions and triggering events (for instance the start of a communication *StartCom*). End points capturing resulting events and post-conditions are illustrated with bars perpendicular to causal paths (for instance *EndCom*). Paths can fork as alternatives (OR-fork) and may also join (OR-join). Alternative branches can be guarded by conditions, shown between square brackets. A condition needs to be true for the guarded path to be followed. In Figure 2, after tuning to a new channel the signal quality might be better or worse. When it is better, the user profile is updated (*UpdProfile*) and the scenario may continue. Otherwise, the mobile station will tune to the previous channel(*TunePrevChan*). Concurrency and partial ordering of responsibilities and events are supported in UCMs through the use of AND-fork and AND-join. While an OR-join simply indicates overlapping of scenarios that share common paths, an AND-join is a synchronization between two or more paths which must all have been visited for the rest of the scenario to progress.

The diamond symbols are called stubs and are used as containers for sub-maps, which are then referred to as plug-in maps. Any map can be a plug-in. The hand-off UCM in Figure 2 is in fact a plug-in for stub Handoff. A hand-off check is triggered (*GoHO*) to determine whether a new channel would result in a better communication quality. Stubs have identifiable input and output segments (IN1, OUT1,...) connected to start points and end points in the plug-in. This binding relation is also made visual in the plug-in, where the connections to the parent stub are shown between curly brackets. Binding relationship ensure that paths flow from parent maps to sub-maps, and back to parent maps. While static stubs contain only one plug-in map (e.g *HandOff*), dynamic stubs (drawn with dashed diamonds e.g. *Update*) contain many plug-ins whose selection can be determined at run time according to a selection policy local to the stub.

Note: h_i in Figure 1 represent the hyper-edges connecting different UCM constructs (see Section 3 for more details).

After having authenticated the call originator and updated its location record in the same database (*UpdateHLoc*), the system needs also to update the visiting databases

if a mobile user enters or leaves a visiting area. This can be expressed by using two alternative plug-ins for stub Update (Figure 3). The first plug-in is selected when the mobile user is in the same area as before, and the visiting profile is updated if this area is not the home area. The second plug-in is selected when the mobile user has entered a different area. Different activities (deletion and creation of visitor profiles are required to handle the various situations where the old and new areas are the home area or visiting areas (i.e. home→visiting, visiting→home, Visiting→other Visiting). After the allocation of the necessary resources (*GetRes*), the call is answered and the two parties can start communicating (*Talking*). Upon disconnection (*Disconnect*), allocated resources are released (*RelRes*) and the communication is terminated.

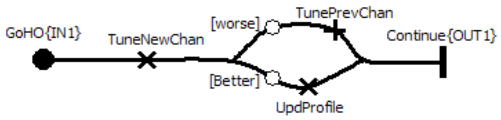
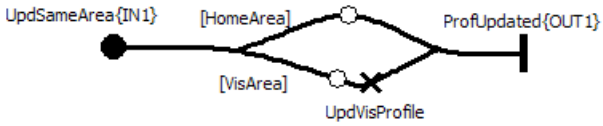
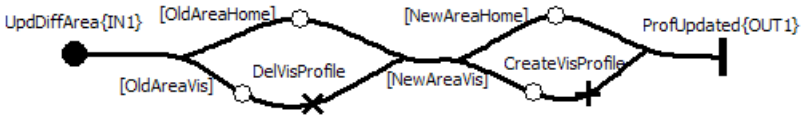


Fig. 2. Plug-in for the HandOff Stub



(a) Plug-in Update same Area



(b) Plug-in Update different Area

Fig. 3. Plug-ins for the Update Dynamic Stub

The set of global variables for the UCM map are: OK (call authenticated or not), Area (homeArea or VisArea), oldArea (oldAreaHome or OldAreaVis), NewArea (NewAreaHome or NewAreaVis) and Quality (better or worse). Different values of these variables are placed on the UCM guards to describe different scenarios'alternatives. A more detailed discussion on the wireless system can be found in [2].

In order to provide a formal semantics to Timed Use Case Maps, we extend the definition of Use Case Maps provided in [10] and [11] to include additional timing information.

3 Syntax of Timed Use Case Maps

We define a UCM as follows:

Definition 1 (Use Case Maps). *We assume that a timed UCM is denoted by a 8-tuple (D, H, λ, C, GVar, Bc, S, Bs) where:*

- *D is the UCM domain, composed of sets of typed elements. D= SP $\cup$ EP $\cup$ R $\cup$ AF $\cup$ AJ $\cup$ OF $\cup$ OJ $\cup$ Tm $\cup$ ST. Where SP is the set of Start Points, EP is the set of End Points, R is the set of Responsibilities, AF is the set of AND-Fork, AJ is the set of AND-Join, OF is the set of OR-Fork, OJ is the set of OR-Join, Tm is the set of Timers, and ST is the set of Stubs.*
- *H is the set of hyper-edges connecting UCM constructs to each other*
- *λ is a transition relation defined as:λ=D$\times$H$\times$D*
- *C is the set of components (C = $\emptyset$ for unbound UCM)*
- *GVar is the set of global variables.*
- *Bc is a component binding relation defined as Bc =D$\times$C. Bc specifies which element of D is associated with which component of C. Bc is empty for unbound UCM.*
- *S is a plug-in binding relation defined as S = ST$\times$RS$\times$GVar.*
- *Bs is a stub binding relation and is defined as Bs =ST$\times\{$IN/OUT$\}\times\{$SP/EP$\}$. Bs specifies how the start and end points of the plug-in map would be connected to the path segments going into or out of the stub.*

Before defining the timed syntax of different UCM constructs, we introduce the following definitions and assumptions:

- **MClock**(Master Clock). The passing of time is modelled by a master clock that increases the global time and adjusts all local clocks accordingly. UCM constructs may be labeled with a time constraint in the following form 'MClock= τ' representing a delay in their execution. In such a case, the construct should be enabled τ time units after starting the UCM execution (i.e., MClock = 0).
- δ**:** Represents the master clock tick, which refers to the smallest time unit used to track system evolution over time. Only a tick advances time and it also defines the granularity of the master clock.
- **Duration.** Denotes the time it takes to carry out an execution of a responsibility. In general, time is only consumed by responsibilities. The absence of a duration value for a responsibility is expressed by the symbol $\perp$. Control constructs, such as OR-fork, are instantaneous (Duration = 0). However, time may elapse in any UCM construct if its execution is delayed. For instance, if the master clock displays 'MClock=2δ', The execution of a UCM construct labeled with a time constraint 'MClock= 4δ' should be delayed by 2δ. Responsibility with undefined duration(i.e., Duration = $\perp$) may cause a system deadlock. To avoid such situation, we assume that a responsibility with undefined duration takes one clock tick to complete.
- Time may elapse in AND-Join constructs, where incoming flows should synchronize (time passes by while waiting for all incoming hyperedges to be enabled).
- We assume that transitions are urgent and instantaneous: Transitions are processed as soon as they are enabled allowing for a maximal progress. Therefore, transitions can be considered as *eager* according to the definition of urgency introduced in [6].

Definition 2. *[Timed UCM Constructs]*

- **Start Points** are of the form *SP(PreCondition-set, TriggerringEvent-set, SPLabel, in, out, T)* where the parameter *PreConditions-set* is a list of conditions that must be satisfied in order for the scenario to be enabled (if no precondition is specified, then by default it is set to true). The parameter *TriggeringEvents-set* is a list that provides the set of events that can initiate the scenario along a path. One event is sufficient for triggering the scenario. The parameter *SPLabel* denotes the label of the start point. A start point should not have an incoming edge except when connected to an end point (called a waiting place). In such a situation, we use the parameter *in* $\in H$ to represent the connection with an end point. The parameter *out* $\in H$ is the (unique) outgoing hyperedge. *T* is optional and it introduces a delay in the start point triggering. '*T* = τ' means that the start point is triggered at 'MClock = τ'.
- **End Points** are of the form *EP(PostCondition-set, ResultingEvent-set, EPLabel, in, out)* where the parameter *PostConditions-set* is a list of conditions that must be satisfied once the scenario is completed. The parameter *ResultingEvent-set* is a list that gives the set of events that result from the completion of the scenario path. The parameter *EPLabel* denotes the label of the end point; the parameter *in* $\in H$ is the (unique) incoming hyperedge. End points have no target hyperedge except when connected to a start point (i.e. a waiting place). In such a case, *out* $\in H$ represents such connection. End points cannot be delayed.
- **Responsibilities** are of the form *Resp(in, Res, out, duration, T)* where *in* $\in H$ is the incoming hyperedge, *Res* is the activity to be executed, and *out* $\in H$ is the outgoing hyperedge. A responsibility is connected to only one source hyperedge and to one target hyperedge. 'duration' is the time taken by the responsibility to complete its execution. Similarly to the start point *T* is used to specify the delay before the start of execution of the responsibility.
- **OR-Forks** are of the form *OR-Fork(in,* $[Cond_i]_{i\leq n}$*,* $[out_i]_{i\leq n}$*, T)* where *in* denotes the incoming hyperedge, $[Cond_i]_{i\leq n}$ is a finite sequence of Boolean expressions, and $[out_i]_{i\leq n}$ is a sequence of outgoing hyperedges. Parameter *T* denotes a possible delay.
- **OR-Joins** are of the form *OR-Join(*$\{in_i\}_{i\leq n}$*, out, T)* where $\{in_i\}_{i\leq n}$ denotes the incoming hyperedges and, *out* is the outgoing hyperedge. Parameter *T* denotes a possible delay.
- **AND-Forks** are of the form *AND-Fork(in,* $\{out_i\}_{i\leq n}$*, T)* where *in* denotes the incoming hyperedge, and $\{out_i\}_{i\leq n}$ is a sequence of outgoing hyperedges. Parameter *T* denotes an optional delay.
- **AND-Joins** are of the form *AND-Join(*$\{in_i\}_{i\leq n}$*, out, T)* where $\{in_i\}_{i\leq n}$ denotes the incoming hyperedges, and *out* is the outgoing hyperedge. Parameter *T* denotes an optional delay.
- **Timers** are of the form *Timer(in, TriggerringEvent-set, out, out_timeout, T)*. The synchronous timer, as defined in [15], is very similar to a basic OR-Fork rule with only two disjoint branches. The parameter *TriggeringEvents-set* is the list that contains the set of events that can trigger the continuation path (i.e. represented by *out*) and the parameter *out_timeout* $\in H$ denotes the timeout path. For timers, *T* defines the timer's expiration time.

– **Stubs** have the form *Stub(*$\{entry_i\}_{i\leq n}$, $\{exit_j\}_{j\leq m}$*, isDynamic,* $[Cond_k]_{k\leq l}$, $[plugin_k]_{k\leq l}$*)* where $\{entry_i\}_{i\leq n}$ and $\{exit_j\}_{j\leq m}$ denote respectively the set of the stub entry and exit points. *isDynamic* indicates whether the stub is dynamic or static. Dynamic stubs may contain multiple plug-ins, $[plugin_k]_{k\leq l}$ whose selection can be determined at run-time according to a selection-policy specified by the sequence of Boolean expressions $[Cond_k]_{k\leq l}$.

We have added the modelling of timing as an orthogonal feature to the untimed UCM syntax presented in [10]. The untimed syntax is restored simply by removing the duration of responsibilities and the delay of execution of different constructs.

Example: The plugin of the HandOff stub of Figure 4 can be described as follows: plug-in-HandOff=(D, H, λ, C, GVar, Bc, S, Bs)

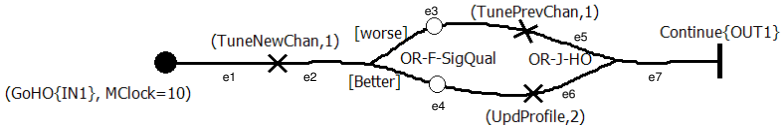


Fig. 4. Timed plug-in for stub HandOff

Where:

– D = {GoHO, TuneNewChan, OR-F-SigQual, TunePrevChan, UpdProfile, OR-J-HO, Continue}
– H = {e1, e2, e3, e4, e5, e6, e7}
– λ = {(GoHO, e1, TuneNewChan),(TuneNewChan, e2, OR-f-SigQual),(OR-F-SigQual, e3, TunePrevChan),(OR-F-SigQual, e4, UpdProfile),(TunePrevChan, e5, OR-J-HO), (UpdProfile, e6, OR-J-HO),(OR-J-HO, e7, Continue)}
– GVar = {Quality}; C = $\emptyset$; Bc = $\emptyset$; S= $\emptyset$; Bs = $\emptyset$

Start point *GoHO* in Figure 4 should be triggered at 'MClock=10'. Responsibilities *TuneNewChan* and *TunePrevChan* take 1 clock tick to complete, while *UpdProfile* takes 2 clock ticks to complete. In this example, responsibilities should start immediately without delaying. They are considered as *eager responsibilities* according to the definition of urgency introduced in [6].

Definition 3 (Access functions).

We define the following access functions:

1. **enables**: D$\rightarrow H^n$. Given a UCM construct *Constr* $\in$ D, *enables* provides the set of hyper-edges that the construct enables after it completes its execution. For instance *enables(Resp(in, Res, out, duration, T))*={*out*}. Outgoing hyper-edges may be associated with guard conditions (i.e., OR-fork and dynamic stubs). Function *enables* evaluates the guards and chooses the outgoing hyperedge associated with the true condition.

2. **triggered**: $\rightarrow D^n$. Gives the set of constructs that should be triggered at the present time. For instance, in Figure 4 at time *MClock=10*, start point *GoHO* may be triggered.
3. **incoming**: D $\rightarrow H^n$. Given a UCM construct, *incoming* provides the set of hyper-edges directly leading to the construct. For instance, Incoming(OR-Join($\{in_i\}_{i\leq n}$, out, T))= $\{in_i\}_{i\leq n}$.
4. **target**: H $\rightarrow$ D. Gives the subsequent construct directly connected to a given hyperedge.
5. **delay**: D $\rightarrow\mathbb{N}$. Gives the delay associated with the UCM construct. For instance *Delay(Resp(in, Res, out, duration, T))= T.*
6. **type**: D$\rightarrow$\{SP, EP, AJ, AF, OJ, OF, Tm, ST\} specifies the type of a UCM construct.

Note: For the sake of clarity, functions *enables*, *target* and *delay* could be applied to a sequence of elements of the specified types and produce a sequence of elements of the resulting types. For instance, target([h1,h2])=[d1,d2] where constructs *d1* and *d2* are respectively the targets of hyperedges *h1* and *h2*.

4 Formal Semantics of Timed Use Case Maps

Defining a solid UCM time semantics is an initial step towards defining a new version of UCM that can be used for simulation and verification of timed models. Our ultimate goal is to use UCM to build system models that combine functional, architectural and temporal aspects of real-time systems, and then apply the resulting models to check the correctness of these systems.

In this section, we define the formal semantics of timed UCM models in terms of Clocked Transition Systems (CTS) [18]. The original CTS definition introduced in [18] assumes many finitely real-valued clocks. However, models of real time have been classified in the literature as either dense time or discrete time depending on whether the time of occurrence of an event is expressed as a real number or approximated by an integer. In our proposed semantics, we consider a discrete time model to be divided into clock ticks indexed by natural numbers. The elapsed time between the events is measured in terms of ticks of a global digital clock which is increased by one with every single tick. This time model corresponds to the fictitious-clock model from [1] or the digital-clock model from [12].

Formally a Clocked Transition System (CTS): Φ= (V, σ_{init}, $\rightarrow$) consists of:

– V = (H-taken, C-active, H-enabled, C-timers, T-trigger, MClock). Where:
  - *H-Taken* represents the set of already traversed hyper-edges.
  - *C-active* represents a sequence of UCM constructs currently executing.
  - *H-enabled* represents a sequence of enabled hyper-edges (i.e. to be traversed during the next transition) associated with the sequence of active constructs in *C-active*.
  - *C-timers* represents a sequence of timers (i.e., clocks) associated with the active constructs *C-active*. *C-timers* monitor the remained executing time of active constructs. *C-timers* is initialized with the duration of execution of every construct in *C-active*.
    Note: Timers in *C-timers* cannot go below zero.

- *T-trigger* represents the sequence of time constraints (or delays) associated with the sequence of active constructs *C-active*. *T-trigger* values correspond to the parameter *T* defined in the constructs' signatures (see Definition 2).
- MClock is the Master Clock.

– σ_{init}: represents the initial state. It is required that for the initial state *MClock = 0*.
– $\rightarrow$: A finite set of transitions. Each transition is a function $\rightarrow \subseteq \Sigma(V) \times \Sigma(V)$ mapping each state $s \in \Sigma$ into a set of successors states $s' \in \Sigma$. Instead of writing $(\sigma,\sigma') \in \rightarrow$, we write $\sigma \rightarrow \sigma'$. Informally, states are assignments of values to variables, called valuations. A valuation maps a variable to a value. A transition from one state to another represents that some variables are assigned a different value, i.e., the valuation changes.

A run of Φ is an infinite sequence of valuations, $\pi = \sigma_0\sigma_1\ldots$ satisfying:

– Initiation : $\sigma_0 \models \sigma_{init}$
– Consecution: For each i=0,1,... the valuation σ_{i+1} is a $\rightarrow$ successor of σ_i, i.e, $\sigma_i \rightarrow \sigma_{i+1}$.

A computation of Φ is a run satisfying:

– Time divergence: The sequence σ_0(MClock) σ_1(MClock) ... grows beyond any bound. That is, as *i* increases, the value of MClock at σ_i increases beyond any bound.

We assume that the run-to completion principle applies to the execution of a construct. The execution of a UCM construct cannot be interrupted until it is completed.

We distinguish two types of transition relations $\rightarrow$:

1. **Configuration Transitions**: When a Configuration Transitions is taken, the system configuration defined by the three sequences: *H-taken*, *C-active* and *H-enabled* is updated to indicate which transition has just been taken.
   (H-taken, C-active, H-enabled, C-timers, T-trigger, MClock)$\rightarrow$(H-taken$'$, C-active$'$, H-enabled$'$, C-timers$'$, T-trigger$'$, MClock$'$).
   Where (H-taken$'\neq$H-taken) $\wedge$ (C-active$'\neq$C-active) $\wedge$ (H-enabled$'\neq$H-enabled) $\wedge$ (C-timers$'$=C-timers - δ) $\wedge$ (T-trigger$'\neq$T-trigger) $\wedge$ (MClock$'$=MClock + δ).
   A configuration transition is executed upon the expiration of one or many of elements of *C-timers* (i.e., $\exists$ t $\in$ C-timers such that, t=0) or when the delay associated with a construct elapses (i.e., MClock$\geq$T)).

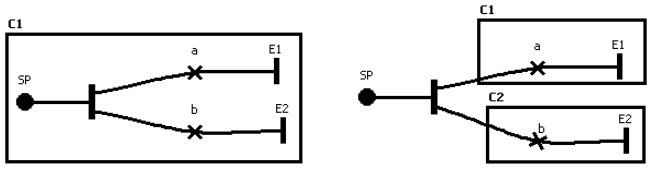
2. **Time Transitions**: When a time transition is taken, then the only variables that change are the global time MClock (which is incremented by a clock tick (δ)), and the timers$\in$C-timers which are decremented by a clock tick (δ). However, the system configuration remains unchanged.
   (H-taken, C-active, H-enabled, C-timers, T-trigger, MClock)$\rightarrow$(H-taken$'$, C-active$'$, H-enabled$'$, C-timers$'$, T-trigger$'$, MClock$'$)
   Where (H-taken$'$=H-taken) $\wedge$ (C-active$'$=C-active) $\wedge$ (H-enabled$'$=H-enabled) $\wedge$ (T-trigger$'$=T-trigger) $\wedge$ (C-timers$'$=C-timer-δ) $\wedge$ (MClock$'$ = MClock + δ)
   Time transitions are executed when none of the timers is about to expire (i.e., $\forall$t $\in$C-timers such that, t$>$0) and none of the constructs is about to start execution (i.e., MClock $<$ T).

In order to establish binding relationship between *C-active*, *H-enabled* and *C-timers*, we define the following correspondance functions: **Atimers**:*C-timers*→*C-active*; and **Htimers**: *C-timers*→*H-enabled*. For instance, let *C-active*=[a1, a2, a3];*H-enabled*=[h1, h2, h3] and *C-timers*=[t1, t2, t3], **Atimers**(t1)=a1 and **Htimers**(t2)=h2.

4.1 Concurrency Model and Time Evolution

The UCM construct AND-Fork allows many paths to execute concurrently. Considering the assumption of run to completion introduced earlier, different scenarios may behave either in:

- **Interleaving Semantics.** At any given time t, only one responsibility may be executing.
  Or
- **True concurrency Semantics.** At any given time t, more than one responsibility may be executing.



(a) Interleaving Semantics (b) True-Concurrency

Fig. 5. Concurrency Semantics

We assume that in presence of UCM components, concurrent paths bounded to the same component are sharing also the same component resources(for instance same CPU). Therefore, these concurrent paths must behave in interleaving semantics. Figure 5(a) illustrates a UCM with two parallel paths bounded to one component. At any time, no more than one responsibility should be active. However, the choice of which responsibility goes first is non deterministic in this case. Adding timing constraints may eliminate non determinism (i.e., if responsibilities *a* and *b* have different values in *T-trigger*).

Parallel paths bounded to different components may behave either according to interleaving semantics or to true concurrency semantics. Figure 5(b) illustrates two parallel paths allocated to two different components. Responsibilities *a* and *b* can be executed in true-concurrency model, since they are enabled at the same time and they don't share the same resources. However, the decision to go with either semantics depends on the real mapping of components to different nodes on a network or to different CPUs, where true concurrency can be achieved.

Note: We assume interleaving concurrency model for unbound UCMs.

In what follows we provide the detailed semantic rules for both concurrency models. The top part of a rule is either a boolean condition that must be true or a computation of a set/subset/variable. For the sake of simplicity, we consider only unfolded UCMs, where all stubs in the root map were already replaced with their corresponding plug-in maps.

4.2 Step Semantics for Interleaving Model

The choice of an interleaving semantics reduces the size of the CTS Variables. Indeed, allowing only one construct to be executed in a given configuration, reduces the set of used variables. Therefore, sequences *C-active* and *C-timers* are reduced to one element since only one variable per sequence is necessary to track the configuration evolution.

Initial State: σ_{init} is defined with the following valuation: (H-taken=$\emptyset$, C-active =$\emptyset$, H-enabled:=enables(triggered), C-timers=0, T-trigger=$\emptyset$, MClock=0). Start points were not included in *C-active* to avoid carrying more than one active construct.

Configuration Transition. Rules 1 and 2 illustrate the configuration transition. As stated earlier in Section 3, time may elapse in AND-join constructs (waiting for all incoming hyperedges to be synchronized). To reflect this fact, we distinguish two cases: a case where no extra delay is involved (Rule 1) and a case where there is an implicit extra delay involved (Rule 2).

Rule 1. ***Configuration Transition: case type(C-active)$\neq$AJ***

$$\begin{gathered}
\textit{(C-timers=0)} \wedge \textit{type(C-active)}\neq \textit{AJ} \\
\textit{h:=}\{\textit{Select any e} \in \textit{H-enabled, such that, delay(target(e)}\leq \textit{MClock }\} \\
\textit{if h=}\emptyset \textit{ then } \{\ \textit{C-active}'\textit{:=}\emptyset\ ;\ \textit{H-taken}'\textit{:=H-taken; H-enabled}'\textit{:=H-enabled }\} \\
\textit{else } \{\textit{C-active}'\textit{:=target(h)} \\
\textit{H-enabled}'\textit{:=H-enabled} \cup \textit{target(C-active}')\textit{-}\{h\} \\
\textit{H-taken}'\textit{:=H-taken} \cup \{h\}\ \} \\
\textit{MClock}'\textit{:=MClock} + \delta \\
\textit{C-timers}'\textit{:=duration(C-active}') \\
\textit{T-trigger}'\textit{:= delay(C-active}')
\end{gathered}$$

$$\textit{(H-taken, C-active, H-enabled, C-timers, T-trigger,MClock)}\rightarrow\textit{(H-taken}', \textit{C-active}', \textit{H-enabled}', \textit{C-timers}', \textit{T-trigger}', \textit{MClock}')$$

Rule 2. ***Configuration Transition case type(C-active)=AJ***

$$\begin{gathered}
\textit{(C-timers=0)} \wedge \textit{type(C-active)=AJ} \\
\textit{Incoming(C-active)} \subseteq \textit{H-taken, such that h:=}\{\textit{any e}\in\textit{H-enabled, such that,} \\
\textit{delay(target(e)}\leq \textit{MClock }\}\} \\
\textit{Incoming(C-active)} \not\subseteq \textit{H-taken, such that, h:=}\{\textit{any e} \in \textit{H-enabled, such that} \\
\textit{e}\notin\textit{enables(C-active)}\} \\
\textit{if h=}\emptyset \textit{ then } \{\ \textit{C-active}'\textit{:=}\emptyset\textit{; H-taken}'\textit{:=H-taken; H-enabled}'\textit{:=H-enabled }\} \\
\textit{else } \{\textit{C-active}'\textit{:= target(h)} \\
\textit{H-enabled}'\textit{:= H-enabled} \cup \textit{target(C-active}')\textit{-}\{h\} \\
\textit{H-taken}'\textit{:= H-taken} \cup h\ \}
\end{gathered}$$

$$MClock' := MClock + \delta$$
$$C\text{-}timers' := duration(C\text{-}active')$$
$$T\text{-}trigger' := delay(C\text{-}active')$$

$$(H\text{-}taken, C\text{-}active, H\text{-}enabled, C\text{-}timers, T\text{-}trigger, MClock) \rightarrow (H\text{-}taken', C\text{-}active', H\text{-}enabled', C\text{-}timers', T\text{-}trigger', MClock')$$

When multiple hyperedges are enabled at a transition, one hyperedge is chosen in a non-deterministic way. Consequently, multiple runs (or timed traces) can be generated from the same UCM scenario.

Time Transition. Rule 3 shows the time transition.

Rule 3. ***Time Transition***

$$(C\text{-}timers \neq 0)$$
$$MClock' := MClock + \delta \ ; \ C\text{-}timers' := C\text{-}timers - \delta$$

$$(H\text{-}taken, C\text{-}active, H\text{-}enabled, C\text{-}timers, T\text{-}trigger, MClock) \rightarrow (H\text{-}taken, C\text{-}active, H\text{-}enabled, C\text{-}timers', T\text{-}trigger, MClock')$$

4.3 Step Semantics for True Concurrency Model

Contrary to the interleaving semantics, *C-active*,*C-timers* and *T-trigger* may have more than one element in presence of concurrent paths. Indeed, *C-active* contains UCM constructs that are being executed concurrently. *C-timers* and *T-trigger* contain their respective sequence of timers and sequence of time delays.

Initial State: σ_{init} is defined with the following valuation: (H-taken=$\emptyset$, C-active =$\emptyset$, H-enabled:=enables(triggered), C-timers=$\emptyset$, T-trigger=$\emptyset$, MClock=0)

Configuration Transition. Rules 4 and 5 show the configuration transition. As stated in the previous section we devise a special rule for AND-join.

Rule 4. ***Configuration Transition: case*** $\forall$***constr*** $\in$ ***C-active, type(constr)*** $\neq$ ***AJ***

$$let\ expire \subseteq C\text{-}timers\ such\ that\ expire \neq \emptyset\ and\ \forall t \in expire,\ t=0$$
$$\exists constr \in C\text{-}active\ such\ that\ delay(constr) \leq MClock$$
$$C\text{-}active' := C\text{-}active - Atimers(expire) \cup target(Htimers(expire))$$
$$H\text{-}enabled' := H\text{-}enabled - Htimers(expire) \cup enables(target(Htimers(expire)))$$
$$H\text{-}taken' := H\text{-}taken \cup Htimers(expire)$$
$$MClock' := MClock + \delta$$
$$\forall t \in C\text{-}timers\ such\ that\ t>0,\ C\text{-}timers' := (C\text{-}timers - \delta) - expire \cup duration(target(Htimers(expire)))$$
$$T\text{-}trigger' := delay(C\text{-}active')$$

$$(H\text{-}taken, C\text{-}active, H\text{-}enabled, C\text{-}timers, T\text{-}trigger, MClock) \rightarrow (H\text{-}taken', C\text{-}active', H\text{-}enabled', C\text{-}timers', T\text{-}trigger', MClock')$$

Rule 5. ***Configuration Transition case*** $\exists$***constr*** $\in$ ***C-active, type(constr)=AJ***

let expire ⊆ C-timers such that expire≠∅ and ∀t∈expire, t=0
let AJ-active⊆C-active / ∀ aj∈AJ-active, type(aj)=AJ and Incoming(aj) ⊆ H-taken
C-active′:= C-active - Atimers(expire) ∪ target(Htimers(expire))
∪target(enables(AJ-active))
H-enabled′:= H-enabled - Htimers(expire) ∪ enables(target(Htimers(expire)))∪
enables(target(enables(AJ-active)))
H-taken′:= H-taken ∪ Htimers(expire)∪ enables(AJ-active)
MClock′:= MClock + δ
∀t∈C-timers such that t>0, C-timers′:= (C-timers- δ)- expire ∪
duration(target(Htimers(expire)))∪ duration(target(enables(AJ-active)))
T-trigger′:= delay(C-active′)

(H-taken, C-active, H-enabled, C-timers, T-trigger, MClock)→(H-taken′, C-active′,
H-enabled′, C-timers′, T-trigger′, MClock′)

Time Transition. Rule 6 shows the time transition.

Rule 6. ***Time Transition***

∀t∈C-timers / t≠0
MClock′:=MClock+δ ; C-timers′:=C-timers-δ

(H-taken, C-active, H-enabled, C-timers, T-trigger, MClock)→(H-taken, C-active,
H-enabled, C-timers′, T-trigger, MClock′)

Note that the runs in the true concurrency semantics model have less states compared to the same runs in the interleaving semantics.

4.4 Applying Timed Semantics to the Simplified Wireless System

Due to space constraints, we limit ourselves to a partial run of the UCM introduced in Figure 1. We have chosen a scenario, where the call originator is not authenticated, resulting in a call rejection (i.e., OK=false). We assume also *duration(ChkAuth)=1* while *duration(LogReject)=2* and δ=1. Figure 6 illustrates the corresponding run.

$H-taken$	$C-active$	$H-enables$	$C-timers$	$T-trigger$	$MClock$
{}	[]	[h1]	[]	[]	0
$\{h1\}$	[ChkAuth]	[h2]	[1]	[⊥]	1
$\{h1\}$	[ChkAuth]	[h2]	[0]	[⊥]	2
$\{h1, h2\}$	[OR-F-Auth]	[h3]	[0]	[⊥]	3
$\{h1, h2, h3\}$	[LogReject]	[h4]	[2]	[⊥]	4
$\{h1, h2, h3\}$	[LogReject]	[h4]	[1]	[⊥]	5
$\{h1, h2, h3\}$	[LogReject]	[h4]	[0]	[⊥]	6
$\{h1, h2, h3, h4\}$	[Reject]	[]	[0]	[⊥]	7

Fig. 6. Partial Execution

To illustrate the concurrency model semantics, we present another partial run of the UCM of Figure 1 starting from the AND-Fork construct. The scenario starts at MClock=9 to allow the start point *GoHO* to be enabled at MClock=10. Variable *Quality* is initialized to *better* and responsibilities *talking* and *UpdProfile* have respectively 5 and 2 as durations. The duration of the remaining responsibilities is fixed to 1. Only new elements of *H-taken* are shown in Figure 7.

$H-taken$	$C-active$	$H-enables$	$C-timers$	$T-trigger$	$MClock$
$\{h1..h8\}$	[AF-Root]	[h9,h11]	[]	[⊥]	9
$\{h9, h11\}$	[GoHO, GetRes]	[e1,h12]	[0,1]	[10,⊥]	10
$\{e1\}$	[tuneNewChan,GetRes]	[e2,h12]	[1,0]	[⊥,⊥]	11
$\{h12\}$	[tuneNewChan,Answer]	[e2,h13]	[0,1]	[⊥,⊥]	12
$\{e2\}$	[OR-F-SigQual,Answer]	[e4,h13]	[0,0]	[⊥,⊥]	13
$\{h13, e4\}$	[UpdProfile,talking]	[e6,h14]	[2,5]	[⊥,⊥]	14
{}	[UpdProfile,talking]	[e6,h14]	[1,4]	[⊥,⊥]	15
{}	[UpdProfile,talking]	[e6,h14]	[0,3]	[⊥,⊥]	16
$\{e6\}$	[OR-J-HO,talking]	[e7,h14]	[0,2]	[⊥,⊥]	17
$\{e7\}$	[Continue,talking]	[h10,h14]	[0,1]	[⊥,⊥]	18
$\{h10\}$	[AJ-Root,talking]	[h15,h14]	[0,0]	[⊥,⊥]	19
$\{h14\}$	[AJ-Root]	[h15]	[0]	[⊥]	20
$\{h15\}$	[Disconnect]	[h16]	[1]	[⊥]	21
{}	[Disconnect]	[h16]	[0]	[⊥]	22
$\{h16\}$	[RelRes]	[h17]	[1]	[⊥]	23
{}	[RelRes]	[h17]	[0]	[⊥]	24
$\{h17\}$	[EndCom]	[]	[0]	[⊥]	25

Fig. 7. Partial Execution: True Concurrency

5 Related Work

In this section, we discuss work related to the notion of time and its support in other modeling languages. The research in this area has taken several directions. One direction consists on focusing on the enhancement of current modeling languages by adding new constructs. UML Real-Time profile [20] uses this approach and adds features for describing a variety of aspects used to model real-time systems, such as timing, resources, performance, schedulability, etc. The current standard UML 2.0 [22] pays more attention to time related aspects than the previous UML version [21]. Indeed, timers and time related types, are present in UML 2.0. In the context of SDL [13], an ITU standard formal description language described in Z.100 document, each action takes an indeterminate time to execute, and that a process stays an unfixed amount of time in a certain state before taking the next fireable transition. This choice may be practical for code generation, in the sense that actual implementations of the system conform to it. However, for simulation purposes, it might be unreasonable since we need to consider

all possible combinations of execution times, timer expirations and timers consumptions. Existing simulation tools consider that actions take 0 time to execute allowing for a high degree of determinism. Our timed UCM semantics, primarily used for simulation purposes, provide a fixed duration to actions (may be relaxed in the future by providing only an upper bound). A valid model of the interpretation of an SDL system is a complete interleaving of different processes at the level of all actions that cannot be transformed into a list of actions (possibly containing implicit states). While using this notion of atomic actions, our proposed semantics consider both concurrency modes: interleaving and true concurrency. The selection of either mode is based on architectural choices.

Another research direction is the combination of an existing modeling notation with another formal description technique to provide better handling of timing aspects. The semantics presented in this paper is comparable to the one presented in [8] where Eshuis presented a formal semantics of UML activity diagrams in terms of clocked transition systems (CTS). However, no distinction between concurrency modes is discusses. The authors in [17] translated UML models with timed properties (e.g. guarded timeouts, transitions dependent on other transition times, etc.) into first-order temporal logic with time support. Knapp et al. [16] used timed state machines for describing a model, and collaboration diagrams with time constraints to describe system properties. In [9] the authors used OCL 2.0 [23] to describe real-time constraints specifications.

6 Conclusion

In this paper, we have presented an extension to the Use Case Maps language that introduces timing information for modeling real-time systems. We have provided a concise formal operational semantics for timed UCM based on Clocked Transition Systems. However, our approach does not consider checking the consistency of the time constraints in the model. In fact, when using true concurrency semantics, one has to ensure that concurrent responsibilities are not updating the same global variables. This can be achieved through a data flow analysis.

As part of our ongoing work, we are investigating the possible extension of our timed UCM syntax and semantics by adding new timed UCM constructs such as *asynchronous timers*, as well as offering the possibility to describe new time constraints.

As part of our future work, we will investigate the possible use of our timed semantics to check the correctness and the consistency of timed UCM specifications.

References

1. Alur, R., Dill, D.L., A Theory of Timed Automata, Theoretical Computer Science, 126, pp.183-235, 1994.
2. Amyot D., Introduction to the user requirements notation: learning by example, Computer Networks: The International Journal of Computer and Telecommunications Networking, Vol. 42, No. 3, pp. 285-301, 2003.
3. Amyot D. and Andrade R., Description of wireless intelligent network services with Use Case Maps, SBRC'99, 17th Simposio Brasileiro de Redes de Computadores, Salvador, Brazil, May 1999, pp. 418-433.

4. Amyot D., Buhr R.J.A., Gray T. and Logrippo L., Use Case Maps for the Capture and Validation of Distributed Systems Requirements. RE'99, Fourth IEEE International Symposium on Requirements Engineering, Limerick, Ireland, June 1999,44-53. http://www.UseCaseMaps.org/pub/re99.pdf
5. Amyot D., Formalization of Time threads Using LOTOS. Master Thesis, Department of Computer Science, University of Ottawa, Canada, 1994.
6. Bornot S., Sifakis J., and Tripakis S., Modeling urgency in timed systems. In International Symposium: Compositionality - The Significant Difference, LNCS 1536, 1998.
7. Buhr R. J. A., Elammari M., Gray T. and Mankovski S., Applying Use Case Maps to multiagent systems: A feature interaction example. In 31st Annual Hawaii International Conference on System Sciences, 1998.
8. Eshuis R., Semantics and Verification of UML Activity Diagrams for Workflow Modelling, Ph.D. Thesis, University of Twente, 2002.
9. Flake S. and Mueller W., A UML Profile for Real-Time Constraints with the OCL. In S. Cook J. M. Jezequel, H. Hussmann, editor, UML2002, Dresden, Germany, number 2460 in LNCS. Springer Verlag, 2002.
10. Hassine, J., Rilling, J., and Dssouli, R. (2005) An Abstract Operational Semantics for Use Case Maps. In: Farn Wang (Ed.): Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October, 2005. LNCS 3731 Springer 2005, 366-380
11. Hassine, J., Dssouli, R., and Rilling, J. (2004) Applying Reduction Techniques to Software Functional Requirement Specifications. In Amyot, D. and Williams, A.W. (Eds) System Analysis and Modeling - Fourth International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 2-4, 2004, LNCS, Volume 3319, Springer, 2005, 138-153
12. Henzinger, T. A., Manna, Z., Pnueli, A., What good are digital clocks?, Proceedings of the ICALP92, LNCS 623, pp.545-558, Spriger-Verlag,1992.
13. ITU-T Recommendation Z.100: Specification and Description Language (SDL). (2002)
14. ITU-T - International Telecommunications Union (2003). Recommendation Z.150 (02/03), User Requirements Notation (URN) - Language Requirements and Framework. Geneva, Switzerland.
15. ITU-T, URN Focus Group (2003), Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM). Geneva. Switzerland, Sept. 2003
16. Knapp A., Merz S., and Rauh C., Model Checking - Timed UML State Machines and Collaborations. In Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Oldenburg, Germany, September 9- 12, 2002, volume 2469 of Lecture Notes in Computer Science, pages 395-416. Springer, 2002.
17. Lavazza L., Quaroni G., and Venturelli M., Combining uml and formal notions for modelling real-time systems. In Joint 8th European Software Engineering Conference, 9th ACM SIGSOFT. ACM SIGSOFT, 2001.
18. Manna Z. and Pnueli A., Clocked transition systems. In A. Pnueli and H. Lin, editors, Logic and Software Engineering, pages 3-42. World Scientific, 1996.
19. Nakamura N., Kikuno T., Hassine J., and Logrippo L., Feature Interaction Filtering with Use Case Maps at Requirements Stage. In: Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00), Glasgow, Scotland, UK, May 2000.
20. OMG. Response to the OMG RFP for Schedulability, Performance and Time, v. 2.0. OMG document ad/2002-03-04, March 2002.
21. OMG Unified Modeling Language Specification, Version 1.5, June 2002.
22. Object Management Group: UML 2.0 Superstructure Specification. (2004)
23. OMG Unified Modeling Language Specification - Object Constraint Language Version 2.0, 2003.