Softw Syst Model (2009) 8:205–220
DOI 10.1007/s10270-007-0076-6

REGULAR PAPER

Use Case Maps as a property specification language

Jameleddine Hassine · Juergen Rilling · Rachida Dssouli

Received: 5 November 2006 / Revised: 23 July 2007 / Accepted: 19 October 2007 / Published online: 7 December 2007

Abstract Although a significant body of research in the area of formal verification and model checking tools of software and hardware systems exists, the acceptance of these tools by industry and end-users is rather limited. Beside the technical problem of state space explosion, one of the main reasons for this limited acceptance is the unfamiliarity of users with the required specification notation. Requirements have to be typically expressed as temporal logic formalisms and notations. Property specification patterns were successfully introduced to bridge this gap between users and model checking tools. They also enable non-experts to write formal specifications that can be used for automatic model checking. In this paper, we propose an abstract high level pattern-based approach to the description of property specifications based on Use Case Maps (UCM). We present a set of commonly used properties with their specifications that are described in terms of occurrence, ordering and temporal scopes of actions. Furthermore, our approach also supports the description of properties with respect to their architectural scope. We provide a mapping of our UCM property specification patterns in terms of CTL, TCTL and Architectural TCTL (ArTCTL), an extension to TCTL, introduced in this research that provides temporal logics with architectural scopes. We illustrate the use of our pattern system for requirement specifications of an IP Header compression feature.

Keywords Formal verification · Temporal logic · Property specification · Use Case Maps · Temporal and architectural scope

Communicated by Dr. Alessandra Cavarra.

J. Hassine (✉) · J. Rilling
Department of Computer Science, Concordia University, Montreal, QC, Canada, H3G 1M8
e-mail: j_hassin@cs.concordia.ca

J. Rilling
e-mail: rilling@cs.concordia.ca

R. Dssouli
Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada, H3G 1M8
e-mail: dssouli@ciise.concordia.ca

1 Introduction

Model checking has been widely used as a method to formally verify finite-state concurrent systems, such as communication protocols. System properties are expressed as temporal logic formulas, and efficient algorithms are used to traverse the resulting model to check whether a system is consistent with the specified properties. Many temporal logics, such as linear-time temporal logic (LTL) [29], computational tree logic (CTL) [13] and ACTL [32] have been suggested as formal languages for property specifications. However, the use of temporal logics is still limited to users with a good mathematical background because temporal logic formulae are difficult to understand and even more difficult to create. To bridge this gap between practitioners and model checking tools, many authors have proposed property specification patterns [1,14,18,25,26] to guide users in expressing system requirements directly in temporal logic.

Previously published pattern systems vary from simple specification patterns dealing with occurrences of events or states (describing *what* must occur) and scopes (describing *when* the pattern must hold) [14], to real-time pattern properties considering information about time [18,26,43]. However, to the best of our knowledge, the existing pattern systems deal mainly with behavioral aspects of systems but fail to capture the architectural scope of a system (describing *where*

the pattern must occur). Applying an architectural scope allows to describe architecture related issues, like "*action P is executed in component C*". Building a property pattern system that considers functional, timing and architectural aspects all together will improve the verification of distributed real-time embedded systems. Such systems often are based on an heterogeneous system architecture; they consist of components that range from fully programmable processor cores to fully dedicated hardware components for time-critical application tasks.

Our research builds upon previous work on property patterns introduced in [14,15,18]. We propose an abstract high level pattern-based approach that supports the description of property specifications using Use Case Maps language (UCM) [42].

In what follows, we present a novel approach that addresses the following concrete issues:

- The UCM language was extended to simplify the writing and understanding of properties, by providing a UCM property pattern system with templates that explicitly capture functional, timing and architectural property patterns. The proposed UCM property patterns system offers users a visual and an easy to learn framework for the specification of complex properties without the use of textual temporal logic formalisms.
- Describing both the requirement specification and the properties with the same formalism (i.e., UCM) will allow for a more detailed analysis while preserving a high level of abstraction.
- A mappings for the UCM property specification patterns in terms of *CTL* and *TCTL* temporal logics is provided.
- We provide an extension to the well-known *TCTL* [2] temporal logic formalism by including additional architectural constraints. The proposed extension is named Architectural TCTL (ArTCTL). However, the definition of a general formal framework for architectural temporal logic is left for future work.

The description of the formal semantics of UCM language is outside the scope of this paper. We refer the reader to:

1. [19], which describes formal semantics of UCM language in terms of Abstract State Machines (ASM).
2. [21], where the authors proposed an extension of UCM language with time and presented the formal semantics to the timed notation in terms of Clocked Transition Systems (CTS).
3. [20], which describes the formal semantics of timed UCMs in terms of timed automata (TA) [3] formalism that can be analyzed and verified with the UPPAAL model checker tool [28].

UCMs [42] can be applied to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a high level of abstraction. They can also provide the stakeholders with guidance and reasoning about the system-wide architecture and behavior. UCM are part of a new proposal to ITU-T for a User Requirements Notation (URN) [22], and have been applied in a number of areas: Design and validation of telecommunication and distributed systems [7,8], detection and avoidance of undesirable feature interactions [12,31], evaluation of architectural alternatives [30] and performance evaluation [36]. UCM is not introduced to replace UML, but rather complement it and help to bridge the gap between requirements (use cases) and design (system components and behaviour). UCM allows developers to model dynamic systems, where scenarios and structures may change at run-time [42]. A formal operational semantics of UCM language in terms of abstract state machines (ASM) was proposed in [19].

Our work has results in two main contributions. First, we have presented a UCM based specification pattern that can simplify creating specification of complex properties without the use of textual temporal logic formalisms and therefore make it available to the novice practitioners. The specification pattern system uses templates to cover most common expected properties found in requirements specifications. Second, we extend the traditional real-time temporal logics to include architectural aspects.

The remainder of the paper is organized as follows. The next section provides an overview of existing pattern systems. In Sect. 3, an overview of Real-time temporal Logics is given. Section 4 introduces the UCM notation and discusses these UCM elements that will be used in the construction of our property patterns. Section 5 presents our UCM property pattern system, which extends real-time patterns to include architectural aspects. In Sect. 6, we give a general overview on how to extend a real-time temporal logic with architectural aspects as well as the syntax and semantics of *ArTCTL*, a proposed extension of *TCTL* formalism. In Sect. 7 we apply our pattern system to the case study introduced in Sect. 4.4. Finally, Sect. 8 discusses the proposed UCM based Specification–Verification framework. Conclusions are presented in Sect. 9.

2 Specification patterns

In this section, we overview the specification patterns by Dwyer et al. [14,15], the timed property patterns by Gruhn et al. [18] and the real-time property pattern by Konrad et al. [26].

2.1 Untimed specification patterns

In [14], Dwyer et al. collected over 500 specifications from several sources and observed that nearly all the properties

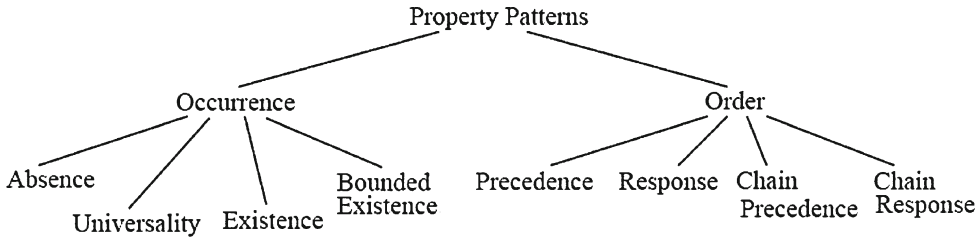


Fig. 1 Pattern hierarchy by Dwyer et al. [14]

could be classified into a hierarchy of basic patterns based on their semantics. This hierarchy, illustrated in Fig. 1, distinguishes properties that deal with the occurrence and ordering of states/events during a system execution. Each of these patterns describes an intent (the structure of the specified behavior), a scope (the extent of program execution over which the pattern must hold), mappings into some specification formalisms for finite-state verification tools (LTL [29], CTL [13], QRE [35]), some known uses, and relationships to other patterns. For instance, the intent of the Precedence pattern is a relationship between a pair of events/states where the occurrence of the first is a necessary precondition for the occurrence of the second (also known as *Enables*).

In what follows we describe briefly the property patterns and their scope as introduced by Dwyer's et al. A more detailed description of these patterns can be found in [14].

- **Absence**. A given event/state P does never occur within a scope.
- **Universality**. A given event/state P occurs throughout a scope.
- **Existence**. A given event/state P must occur within a scope.
- **Bounded Existence**. A given event/state P must occur at least/exactly or at most k times within a scope.
- **Precedence**. An event/state P must always be preceded by an event/state Q within a scope.
- **Response**. An event/state P must always be followed by an event/state Q within a scope.
- **Chain Precedence/Chain Response**. A sequence of events or states $P_1, \ldots, P_n$ must always be preceded/followed by a sequence of events/states $Q_1, \ldots, Q_n$ within a scope.

Dwyer et al. identified five scopes, or segments of system execution:

- **Global**. The pattern must hold during the complete system execution.
- **Before**. The pattern must hold up to a given event/state Q.
- **After**. The pattern must hold after the occurrence of a given event/state Q.
- **Between**. The pattern must hold from the occurrence of a given event/state Q to the occurrence of a given event/state R.
- **After-Until**. Like *between*, but the designated part of the execution continues even if the second event/state R does not occur.

The pattern catalogue allows for reasoning about occurrence and order of events. However, it does not support quantitative reasoning about time due to the fact that real-time properties cannot be specified using these existing patterns. In Dwyer's pattern system, properties like "P must always be followed by Q within k time units" cannot be expressed. In the following section, we present an overview of the work of Gruhn et al. [18] and Konrad et al. [26] who addressed this shortcoming. We also survey some UML-based approaches for property description.

2.2 Timed specification patterns

Konrad et al. [26] have proposed real-time specification patterns that can be classified into three categories of real-time properties: duration (captures properties that can be used to place bounds on the duration of an occurrence), periodic (describes properties that address periodic occurrences), and real-time order (captures properties that place time bounds on the order of two occurrences). Figure 2 illustrates this pattern classification.

The authors have also provided a pattern description template similar to the one proposed in [14] consisting of a pattern name and classification, a pattern intent, a mapping to timed temporal logics (i.e., MTL [5,27], TCTL [2] and RTGIL [4]), examples of known uses, relationships and a structured English specification. The structured English specification captures the scope (*globally*, *before*, *after*, *between*, or *after-until*) followed by the type (*qualitative* or *real-time*) then the category (*duration*, *periodic*, or *real-time order* for real-time properties, and for quality properties (*occurrence* or *order*) of the property. An example of such an English

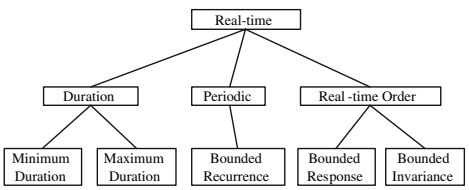


Fig. 2 Real-time specification patterns by Konrad et al.

description is: "Globally, it is always the case that if P holds, then S holds after at most c time unit(s)". Obtaining such a description is the result of the execution of six rules (e.g., property, scope, specification, real-time Type, real-time order category and bounded response pattern).

In [18], Gruhn et al. proposed another catalogue of patterns for real-time requirements. For each pattern, a timed observer automaton is constructed to describe the desired behavior. The observer runs in parallel with the model under verification. The observer reaches an *Error* state if and only if the property can be violated. Therefore, in order to prove that a property holds, it is sufficient to check that the observer cannot reach some location. The catalogue adds the notion of time constraints to the patterns introduced by Dwyer et al. to be able to specify properties like: "Starting from the current point of time, P must occur within k time-units". The corresponding automaton is illustrated in Fig. 3. The catalogue covers many interesting timed patterns and proposes their corresponding timed automata. The use of temporal automata for specifying temporal properties have also been used by several authors [10,34].

A similar observer concept is used in [1], where Alfonso et al. introduced VTS, a visual language to define complex event-based requirements such as freshness, bounded response, event correlation, etc., and a tool that translates these requirements into the input language of the model checker KRONOS. The user has to graphically describe the scenarios violating the requirements, which is in our opinion a major drawback. Indeed, deriving all possible scenarios that violate a given requirement is an error prone activity and the resulting set of scenarios may be incomplete. Tsai et al. [43] describe a testing approach based on scenarios and verification/robustness patterns (SP, VP/RBP). These are temporal patterns (or cause-effect relations) that allow the specification of pre- and post-conditions as well as timing constraints (e.g., optional timeout, time slices, etc.), and are expressed both visually and in LTL temporal logic.

Several approaches for describing properties with UML models have been proposed. These approaches either extend OCL for temporal constraints specification or express behavioral real-time constraints in UML diagrams. Ramakrishnan et al. [37] extend the OCL syntax by additional grammar rules with unary and binary future oriented temporal operators (e.g., always and never) to specify safety and liveness properties. Flake and Muller. [16] have developed a temporal OCL extension that enables modelers to specify state-oriented real-time constraints. This extension covers the consecutiveness of states and state transitions as well as time-bounded constraints.

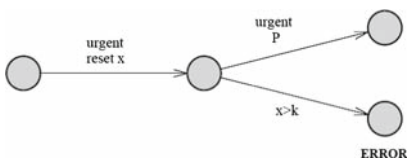


Fig. 3 Timed automaton for time-bounded existence

Schäfer et al. [44] describe systems using UML state machines and use UML collaboration diagrams to specify properties. In order to verify properties using model checker SPIN, state machines model is compiled into a PROMELA model while collaborations are compiled into sets of Büchi automata (i.e., "*never claims*"). Graf et al. [17] proposed a mapping of UML models into a framework of communicating extended timed automata (stereotyped as observers) to serve as property specification language. Although, these UML models have the advantage to be simpler and easier to use for experienced UML users, they suffer from the same drawback as other observer-based approaches. The models require for example the user to describe manually all the scenarios violating the requirements.

3 Real-time temporal logic

Temporal logic has been successfully used for modeling and analyzing the behaviour of reactive and concurrent systems. Standard temporal logics, such as CTL [13], ACTL [32] and LTL [29] which are subset of μ calculus, are inadequate for real-time applications because they only deal with qualitative timing properties. Real-time temporal logics extend standard temporal logics with temporal operators that allow the definition of quantitative temporal relationships—such as distance among events in time units.

In [4,9] many real-time temporal logics have been surveyed and a series of criteria for assessing their capabilities was presented. Among these criteria are the logic expressiveness, the order of the logic, decidability of the logic, the use of temporal operators, the fundamental time entity and the structure of time. In the following we give a brief overview of MTL and TCTL [2]. For a detailed description, we refer the reader to [2,27].

- **MTL**. Metric temporal logic (MTL) [27] is an extension to LTL [29] in which the temporal operators are replaced by time-constrained versions (always ($\Box$), eventually ($\diamond$), next ($\circ$), strong until ($\mathcal{U}$) and weak until($\mathcal{W}$)). For example, the formula $\Box_{[0,k]}\varphi$ expresses that φ holds for the next k time units. MTL is interpreted over a discrete time line and assumes integer time. MTL is undecidable [5].
- **TCTL**. Timed computational tree logic (TCTL) proposed by Rajeev Alur in 1991 [2] is a propositional branching-time logic. TCTL extends CTL [13] by allowing timing constraints on the temporal operators (always (G), eventually (F), strong until (U), and weak

until (W) operators, which are either existentially (E) or universally (A) quantified). For example, the formula $AG(P \Rightarrow AF_k(S))$ expresses the time-bounded response property "*Globally, S responds to P within k time units*". The semantics of TCTL is defined over a dense time line.

In Sect. 6, we will introduce an architectural real-time temporal logic to express architectural constraints for both timed and untimed properties.

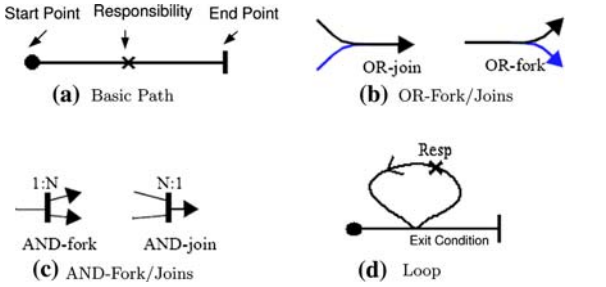


Fig. 4 UCM basic notation

4 UCM language

The UCM notation [42] is a high level scenario based modeling technique that can be used to specify functional requirements and high-level designs for reactive and distributed systems. UCMs are expressed by a simple visual notation that allows for an abstract description of scenarios in terms of causal relationships between responsibilities (e.g., event, operation, action, task, function, etc.) along paths allocated to a set of components. 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). In UCM, scenarios are expressed above the level of messages exchanged between components, hence, they are not necessarily bound to a specific underlying structure (these types of UCMs are called Unbound UCMs). One of the strengths of UCMs are their ability to integrate a number of scenarios together (in a map-like diagram), and the ability to reason about the architecture and its behavior over a set of scenarios. In the following section, we describe and illustrate the UCM notations that are used as part of our specification pattern catalogue. For a detailed description of the all aspects of the UCM notation the reader is refered to [11,42].

4.1 UCM basic notation

A basic UCM path contains at least the following constructs: start points, responsibilities and end points (Fig. 4a). **Start points.** The execution of a scenario path begins at a start point. A start point is represented as a filled circle representing preconditions and/or triggering events. **Responsibilities.** Responsibilities are abstract activities that can be refined in terms of events, functions, tasks, procedures. Responsibilities are represented as crosses. **End points.** The execution of a path terminates at an end point. End points are represented as bars indicating post conditions and/or resulting effects.

UCMs also provide additional constructs for structuring and integrating scenarios sequences, using alternatives (with OR-forks/joins as illustrated in Fig. 4b) or concurrently (with AND-forks/joins as illustrated in Fig. 4c). **OR-Forks.** Represent a path where scenarios split as two or more alternative paths. Conditions (Boolean expression called guards) can be attached to alternative paths. **OR-Joins**, capture the merge of two or more independent scenario paths. **AND-Forks.** Split a single control into two or more concurrent controls. **AND-Joins.** Capture the synchronization of two or more concurrent scenario paths. **Loop**. Captures explicitly a looping path. An exit-condition attribute specifies the condition under which the loop is exit (Fig. 4d).

Note: When maps become too complex to be represented as one single map, UCM provides a mechanism for defining and structuring sub-maps (called *plugins*) in containers called *stubs*. For a detailed description of this UCM important concept we refer the reader to [11].

4.2 UCM timed notation

The UCM language provides two explicit constructs for expressing time constraints:

- **Timer**: A timer is a waiting place that is triggered by the timely arrival of a specific event. It can also trigger a time-out path when this event does not arrive in time. Figure 5a illustrates the *Timer* construct, where a timer should start after inserting an ATM card into the bank machine. If the user enters his/her PIN within a 10 Time Units (TU) time frame (*EnterPIN*(*10TU*)), the PIN will be checked otherwise the card is returned to the user (i.e., time-out path is triggered).
- **Time Stamp**: Time stamps are start and end points for response time requirements (Fig. 5b). In Sect. 5.2.2 we present in more details two examples involving time stamps.

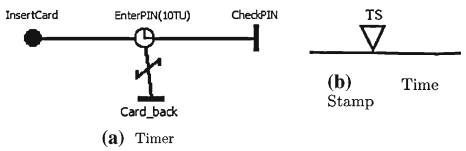


Fig. 5 UCM timed notation

4.3 UCM architectural notation

One of the strengths of UCMs resides in their ability to bind responsibilities to components. The default UCM component notation is abstract enough to represent dependencies (for instance containment), different types (passive, active, etc.), and it even allows to represent run-time instances (without data). Components can be of different types and possess different attributes. Buhr in [11] suggests several types and attributes that are relevant for complex systems (real-time, object-oriented, dynamic, agent-based, etc.).

Figure 6 illustrates some of these component types and attributes proposed by Buhr:

- *Teams* (boxes with sharp corners) are the most generic component that are also most typically used in UCMs. Teams are operational groupings of system-level components.
- *Objects* (boxes with rounded corners) are data or procedure abstractions that are system-level components to support the system comprehension. Processes (Parallelograms) are active components.
- *Slots* (boxes with dashed outlines) may be populated with different instances of components at different times. Slots are containers for *dynamic components* (DC) in execution.
- *Pools* are containers that hold components in readiness to occupy slots (e.g., not executing DC, they act as data).
- *Dynamic components* (see Fig. 6c) can be created, moved, stored, and deleted with dynamic responsibilities such as create, put, get, and move. *Move arrows* (small arrows between paths and pools or slots) are used to indicate the possibility of component movement that may cause slots to become occupied or empty. Movement is a metaphor for changing visibility. Moving a component into a slot allows to make this component visible to those who must interact with it at the slot location level.

The slot notation does not indicate whether slots are empty or not—this requires an analysis of the corresponding paths. Therefore, slots can be seen as places where different components may play the same role at different times. The UCM agent notation is not only used in the context of agent systems, but more generally to represent roles.

UCM are not an Architecture Description Language (ADL), but a high level visual specification language that helps the stakeholders to document and reason about a system-wide architecture and behaviour. ADLs represent a formal way of representing architecture with a primary mission of describing components and their connectivity. ADLs permit analysis of architectures completeness, consistency, ambiguity, performance and support automatic generation of software systems. UCM focus on the behavior of the whole system rather than on their parts.

UCM component relationships depend on scenarios to provide the semantic information about their dependencies. Components are dependent if they share the same scenario execution path. To illustrate the fact that a responsibility is the result of a collaboration among two components, the *shared responsibility* construct is used (see Fig. 7a). The execution of a shared responsibility requires message-like interactions between the involved components. Figure 7b shows one possible refinement of the shared responsibility in terms of a sequence diagram.

4.4 Case study: IP header compression feature

As networks evolve to provide more bandwidth, the applications, services and the consumers of those applications are competing for that bandwidth. In many services and

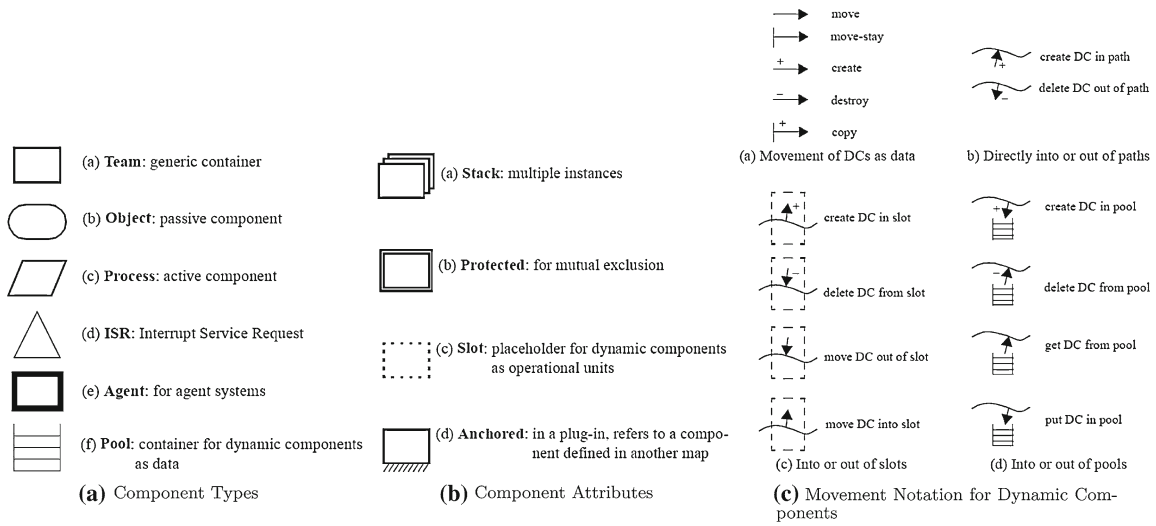
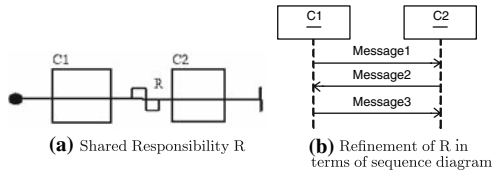


Fig. 6 Component types, attributes and movement notation for dynamic components [6]



(a) Shared Responsibility R **(b)** Refinement of R in terms of sequence diagram

Fig. 7 Shared Responsibility and one possible refinement

applications, such as voice and video over IP, several fields in the header of a given flow remain constant for the length of the flow. IP header compression (IPHC) achieves major gain in terms of packet compression because although some fields in the header change in every packet, the difference from packet to packet is often constant, and therefore the second-order difference is zero. The decompressor can reconstruct the original header without any loss of information. IPHC is a hop-by-hop compression scheme (i.e., works on a point-to-point link). IP header compression can improve throughput and reduce packet loss and delay.

4.4.1 IPHC preconditions

Before any IP packets may be communicated, *PPP* (which allows two machines on a point-to-point communication link to negotiate various parameters for authentication) and *IPCP* (responsible for configuring, enabling, and disabling the IP protocol modules on both ends of the point-to-point link) negotiations must be completed successfully. Figure 8 describes this negotiation phase using the shared responsibility construct.

4.4.2 Compression/decompression types

Mainly three types of compression were presented in RFC 2507 [38], RFC2508 [39] and RFC1144 [40]: RTP (RTP compression: Real Time Protocol), cUDP (UDP compression) and cTCP (TCP compression). For non-RTP traffic another type of compression called non-TCP can be used as well.

Compression/decompression takes place either in the fast Path (ASIC forwarding) or the slow path (software forwarding) depending on the type of traffic. A possible design of IPHC may consider compressing/decompressing cRTP and cUDP traffic on a fast path while compressing/decompressing

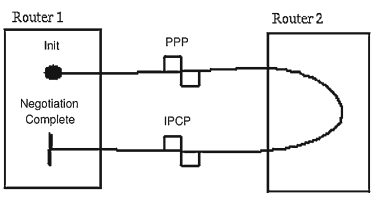


Fig. 8 PPP and IPCP negotiation

cTCP traffic on slow path since protocol packets over the TCP transport would constitute a significantly lower percentage of all traffic in typical application profiles that use IPHC.

4.4.3 IP header compression requirements

Compression scenario. Figure 9 illustrates a high level view of the compression scenario. UCM start point *Rec-Uncompr-Packet* denotes the reception of a non-compressed packet. Then the router checks whether the egress interface (towards the destination) is IPHC enabled. If the egress interface is not IPHC-enabled, the packet is then forwarded uncompressed towards its destination (i.e., Dynamic responsibility *Send-Uncompressed*). Otherwise, the compressor checks the packet type to distinguish compressible packets. The design presented in Fig. 9 does not consider plain IP packets and IP packets with options for compression. Compressible packets (RTP, UDP and TCP) are looked up in a repository of packet headers (i.e., UCM responsibility *HeaderLookup*). If a matching header (that corresponds to the context of a given flow) is found, the incoming packet is compressed. If no match is found, the packet header is copied into that repository and a new context is defined. Then depending on the protocol type the corresponding compression type is selected and applied to the packet (i.e., cRTP, cUDP or cTCP). Compression latency is expected to be within 100 μs.

Decompression scenario. Figure 10 illustrates a high level view of the decompression scenario. The decompressor should handle two types of packets: (a) Full Header and (b) Compressed packet. The start of a compressed flow is indicated by the arrival of a Full header. The decompressor will store the contents of the header from the Full header packet (i.e., responsibility *StoreContextID*). Subsequent compressed packets will be decompressed by using the stored context from the Full header packet (i.e., *RetrieveContextID*) and the information present in the compressed packet. If there is no matching with the stored context ID or the packet is out of sequence, then the packet is discarded and a context state packet (CS packet) is sent to the compressor to notify that something wrong happened (i.e., *GenerateCS-Packet*).

5 UCM property pattern system

In this section, we present a graphical specification pattern catalogue based on the UCM notation. Our proposed pattern system covers all qualitative specification patterns introduced by Dwyer et al. [14] as well as real-time specification patterns presented in [1,18,26]. The research is motivated by the goal to capture both qualitative properties and quantitative timing requirements. Furthermore, as the structural aspects of a system can be captured without the user having to be familiar with temporal logic for the representation of

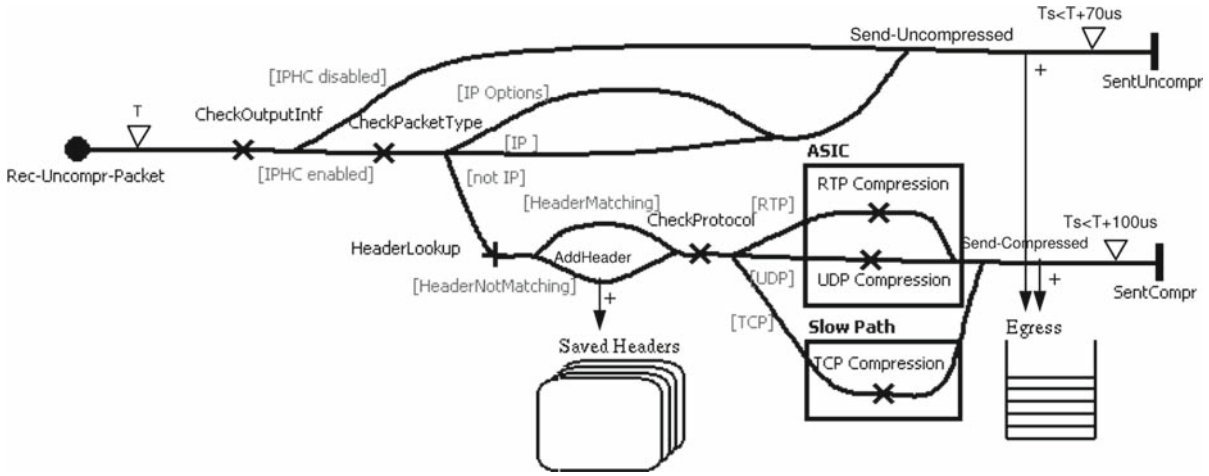


Fig. 9 IPHC: compression scenario

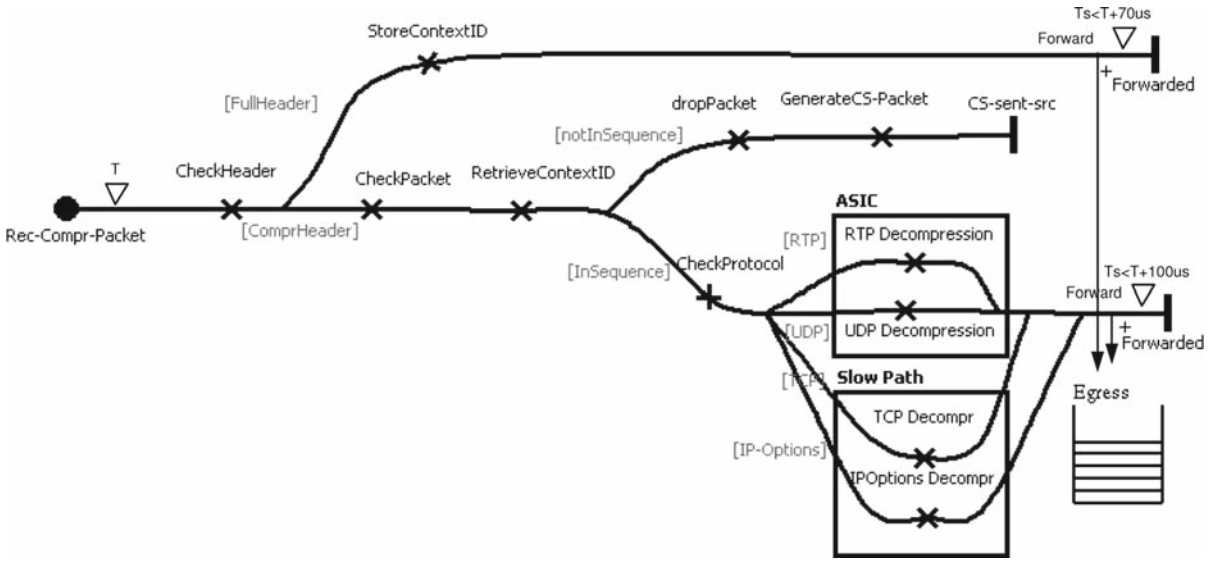


Fig. 10 IPHC: decompression scenario

the properties and the description of scenarios that violate the requirements [1,18].

Although, UCM is primarily a functional description language (i.e., behavior oriented), it can be used to reason about atomic propositions as well. In addition to the UCM representation, we provide a mapping of our pattern catalogue to temporal logics CTL and TCTL. When reasoning about responsibilities/actions, our UCM-based pattern system can be easily mapped to ACTL [32], which extends CTL with actions. Like CTL, ACTL is a propositional branching-time temporal logic. While CTL is interpreted over Kripke structures, ACTL is interpreted over labeled transition systems (LTSs). A more detailed description of the relationship between CTL and ACTL can be found in [33]. Real-time properties may also be mapped to a real-time version of ACTL called ATCTL [23].

In the context of our research, one important aspect for us was not to have to extend the existing UCM language by introducing additional new notations. Instead, we extend the use of existing UCM notations. For example we extended the use of UCM labels that are typically applied to identify different UCM constructs (e.g., construct's name), by existential and universal quantifiers. These quantifiers can be then applied to specify the scope of our specification patterns.

5.1 Patterns

In this section, we describe the qualitative properties of the patterns introduced by Dwyer et al. [14] using an UCM based representation and their mapping to CTL logic. Formulas in CTL are composed of atomic propositions, boolean connectors, and temporal operators. Temporal operators consist of

forward-time operators (**G** globally, **F** in the future, **X** next time, and **U** until) preceded by a path quantifier (**A** all computation paths, and **E** some computation paths).

For clarity purpose we will use a "*global scope*" to represent these properties. Temporal scopes will be discussed in Sect. 5.2.1.

5.1.1 Absence

In order to describe that a given responsibility/event P never occurs within a defined scope, we extend both the UCM responsibility labels with the negation operator *not(P)*, which represent any sequence of responsibilities not containing P. The label "not($P_1, \ldots, P_n$)" denotes any sequence of responsibilities that does not contain the set of responsibilities $P_1, \ldots, P_n$.

Figure 11a illustrates this absence property.

$$Mapping\ to\ CTL : AG(\neg P) \tag{1}$$

5.1.2 Universality

Universality is a dual of the absence property stating that a given responsibility/event P occurs (Fig. 11b). Adding, an existential quantifier (i.e., there exists) to the start point label shows that responsibility/event P should occur at least once during a possible execution (e.g., at least one path).

$$Mapping\ to\ CTL : AG(P) \tag{2}$$

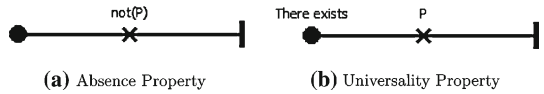
5.1.3 Existence

The start point is labeled with the universal quantifier to state that for all possible execution paths P must occur (Fig. 12a).

$$Mapping\ to\ CTL : AF(P) \tag{3}$$

5.1.4 Bounded existence

Responsibility P must occur at least/exactly/most k times. This is achieved by adding cardinalities to the responsibility label. "$P(n \ldots m)$" denotes that P is repeated at least n times and at most m times (Fig. 12b).



(a) Absence Property **(b)** Universality Property

Fig. 11 Absence and Universality

(a) Existence Property **(b)** Bounded existence

Fig. 12 Existence and bounded existence

One instance of the bounded existence pattern, where P occurs at most 2 times, is represented by the following CTL formula:

$$Mapping\ to\ CTL : \neg EF(\neg P \wedge EX(P \wedge EF(\neg P \wedge EX(P \wedge EF(\neg P \wedge EX(P)))))) \tag{4}$$

5.1.5 Response

A directed arrow between responsibilities P and Q shows that when P occurs then an occurrence of Q should follow. The star in front of the responsibility label means: "*if P occurs*" (Fig. 13a).

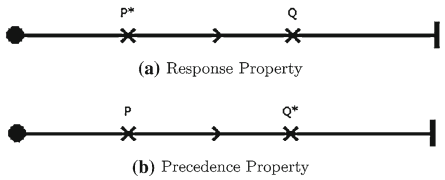
$$Mapping\ to\ CTL : AG(P \Rightarrow AF(Q)) \tag{5}$$

Causality is always defined by construction in UCMs. However, the arrow is added to distinguish the general response property from a restricted response property (i.e., Q should immediately follow P). In the later, the directed arrow is omitted.

5.1.6 Precedence

The precedence property represents a restriction of the response property in the sense that Q can only follow P (Fig. 13b). The star in front of the responsibility label means "*if Q occurs*".

$$Mapping\ to\ CTL : \neg E[\neg P\ U(Q \wedge \neg P)] \tag{6}$$



(a) Response Property

(b) Precedence Property

Fig. 13 Response and precedence

5.1.7 Chain precedence/chain response

A sequence of responsibilities $P_1, \ldots, P_n$ must always be preceded/followed by a sequence of responsibilities $Q_1, \ldots, Q_n$. In addition to its name, each responsibility is labeled with the name of the chain it belongs to. Figure 14a, b illustrates the chain Precedence/Response. $P_1, P_2, \ldots, P_n$ belong to chain *S1* while $Q_1, Q_2, \ldots, Q_n$ belong to chain *S2*.

Note: The chain precedence/response makes only sense for cases with non-overlapping chains.

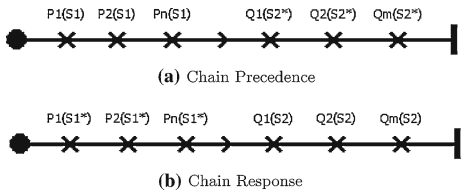


Fig. 14 Chain precedence and chain response

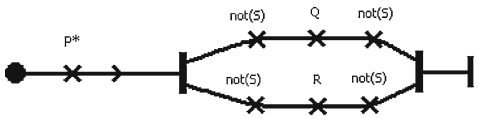


Fig. 15 Separated responses

The CTL mapping of the precedence chain "P_1 precedes Q_1 and Q_2" is as follows:

$$\neg E[\neg P_1 \, U(Q_1 \wedge \neg P_1 \wedge EX(EF(Q_2)))] \tag{7}$$

The CTL mapping of the response chain "Q_1, Q_2 responds to P_1" is as follows:

$$AG(P_1 \Rightarrow AF(Q_1 \wedge AX(AF(Q_2)))) \tag{8}$$

We now have introduced the elements necessary to describe a more complex requirement:
Separated Response. Describes that a responsibility P is followed by two responses Q and R, which are not separated by S. An *AND-Fork* is used to specify that Q and R may occur in any order (Fig. 15).

$$Mapping\ to\ CTL: AG(P \Rightarrow AF(Q \wedge \neg S \wedge AX(A[\neg S \, U \, R]))) \tag{9}$$

5.2 Specification pattern scopes

5.2.1 Temporal scopes

The optional temporal scopes define when the above patterns must hold. The scope is determined by specifying a start and an end state/event for the pattern. Dwyer et al. [14] defined five different types of scope. For each temporal scope, we present only the mapping of the precedence property in terms of CTL. For a complete CTL mapping of the qualitative properties with respect to all the described scopes, we refer the reader to [41].

– **Global**: Start and end point labels are left blank to state that the pattern must hold during the complete system execution (Fig. 16a). The CTL Mapping for "S precedes P" is given by

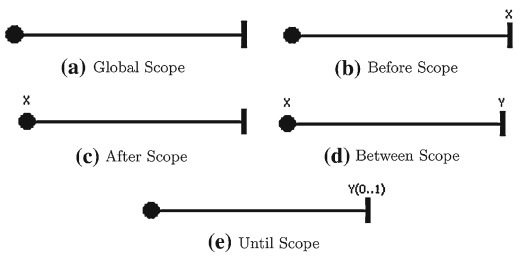


Fig. 16 Temporal scopes

$$A[\neg P \, W \, S] \tag{10}$$

– **Before**: The end point is labeled with the event X to state that the pattern must hold up to a responsibility/event X (Fig. 16b). The CTL mapping for "S preceded P Before R" is

$$A[(\neg P \vee AG(\neg R)) W (S \vee R)] \tag{11}$$

– **After**: The start point is labeled with the event X to describe that the pattern must hold after the occurrence of a responsibility/event X (Fig. 16c). The CTL mapping for "S preceded P After Q" is:

$$A[\neg Q W (Q \wedge A[\neg P \, W \, S])] \tag{12}$$

– **Between**: The start point is labeled with X and the end point is labeled with Y to describe that the pattern must hold from the occurrence of X to the occurrence of Y (Fig. 16d). The CTL mapping for "S precedes P Between Q and R":

$$AG(Q \wedge \neg R \Rightarrow A[(\neg P \vee AG(\neg R)) W (S \vee R)]) \tag{13}$$

– **Until**: The same as "*between*", but the pattern must hold even if Y never occurs. The end point is labeled with Y having a cardinality of either 0 (in case Y never occurs) or 1 (in case Y occurs) (Fig. 16e). The CTL mapping for "S preceded P After Q until R":

$$AG(Q \wedge \neg R \Rightarrow A[\neg P \, W (S \vee R)]) \tag{14}$$

Note: A scope label may coexist with a pattern related label on a start point. For instance, the start point of a UCM describing "an existence property that should hold after the occurrence of an event X", is labeled with "There exists|X".

5.2.2 Examples of timing requirements

– **Bounded Response:** Figure 17a describes a bounded response, where P is followed by Q after 10 time units

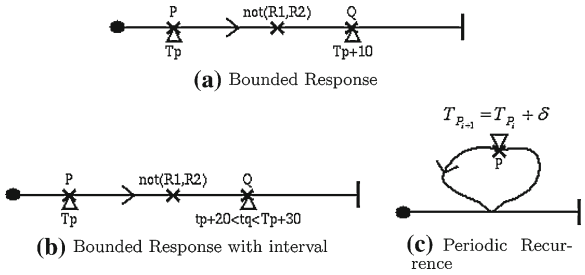


Fig. 17 Examples of Timing Requirements

and neither *R1* nor *R2* should occur between *P* and *Q*. Its corresponding TCTL mapping is

$$AG(P \rightarrow (AF_{\leq 10} Q) \wedge \neg R1 \wedge \neg R2) \tag{15}$$

Figure 17b describes the same property but with a relaxed interval for responsibility *Q*. *Q* is supposed to occur 20 TU after the occurrence of *P* but not more later than 30 TU. Its corresponding TCTL mapping is:

$$AG(P \rightarrow (AF_{[20,30]} Q) \wedge \neg R1 \wedge \neg R2) \tag{16}$$

– **Periodic Recurrence:** Figure 17c illustrates a periodic occurence of responsibility *P* where *P* occurs every δTU.

$$Mapping\ to\ TCTL : AG(AF_{\leq \delta} P) \tag{17}$$

5.2.3 Architectural scopes

Architectural descriptions are playing an increasingly important role in the ability of software engineers to describe and comprehend software systems. Architecture is generally considered to consist of components and the connectors (interactions) between them. Architectural reasoning needs to cope with evolving system requirements, where systems evolve to migrate to new technologies or/and to include new features. These changes may modify the assumptions on which system functionalities are based. Therefore, test engineers may want to:

– Ensure that the desired topology is preserved for a specific feature (e.g., feature functionalities should be bound to a specific topology).
– Ensure that components that are intended to interact can indeed do so (e.g., there exists a scenario that is divided into many components).

In an effort to address these architectural issues, we introduce architectural scopes with the goal to increase the understandability and reasoning about architectural designs. At the same time we allow for improved analysis and testing while preserving a high level of abstraction. UCM have the benefit of integrating both behavioural and architectural aspects in one representation.

A user may for example want to express a response property, where *Q* should follow the occurrence of *P* and this should happen between any occurrence of *X* and *Y*. In addition, the property should hold only and only if the responsibility/event *P* is executed by *Process1* while responsibility/event *Q* is executed within *Object1*. This generic response property can be described as shown in Fig. 18.

We can define five distinctive architectural scopes:

– **Component Specific:** The pattern must take place within a pre-defined component. The architectural property is violated when the responsibility/event occurs within a different component. Figure 19 illustrates a generic property where responsibility *R* should occur as a part of process "Process 1".
– **Multiple Same Type Components:** The *Component Specific* scope is relaxed to give the user the possibility to specify more than one component of the same type for a certain event/responsibility. Figure 20 illustrates a generic property where responsibility *R* should occur as a part of either *Agent1* or *Agent2*.
– **Multiple Different Type Components:** The *Component Specific* scope is relaxed to give the user the possibility to specify more than one component of different types for a certain event/responsibility. Figure 21 illustrates such a generic property where responsibility *R* should occur as a part of either agent *Agent1* or process *Process1*.
– **Any Component:** The property may occur within any component of a predefined type. This is described by

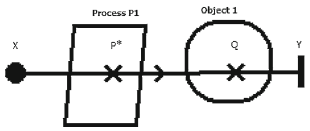


Fig. 18 Property involving three scopes: Occurrence, temporal and architecture

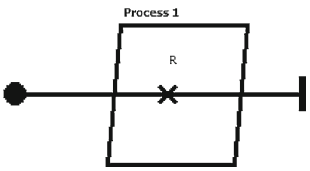


Fig. 19 Component Specific

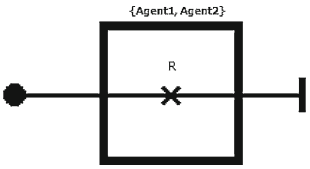


Fig. 20 Multiple Same Type Components

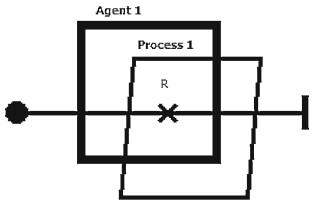


Fig. 21 Multiple Different Type Components

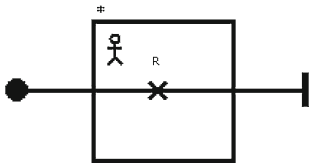


Fig. 22 Any Component

using "*" as the name of the component. Figure 22 illustrates a generic property, where responsibility *R* should occur as a part of an actor. The actor name in this case is not specified.

– **Unbound:** For unbound event/responsibility (i.e., not attached to any component), the component name or type are not relevant. The event/responsibility can take place within any component of any type. The focus is on the behaviour and timing aspects rather than the architectural aspect. Figure 23 illustrates a generic property, where responsibility *R* is not attached to any other component.

Note: A component may be part of another component (For instance a process can fork to have childs). This architectural containment dependency may be represented as part of the property definition. Figure 24 illustrates a property with a responsibility *R* performed by *object1* which is part of process *Process1*.

In the following Section, we give a general overview on how to extend real-time temporal logics with architectural aspects. We extend TCTL, one variant of real-time temporal logic, to include architectural constraints. We describe the formal syntax and semantics of what we name *ArTCTL*.

The definition of a complete syntax and semantics for architectural real-time temporal logics is out of the scope of this paper.

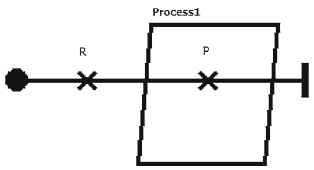


Fig. 23 Unbound

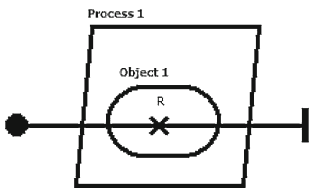


Fig. 24 Architectural Containment Dependency

6 Architectural real-time temporal logic

Labelled transition systems (LTSs) are used to reason about non-real-time systems. For real-time systems, timed transition systems (TTS) are used, which can be seen as an extension of labelled transition systems. The passing of time is modelled by labelling transitions with non-negative real numbers. Timed automata (TA) [3] have been defined to describe timed languages. The semantics of a TA are usually described in terms of a timed transition system (TTS).

We slightly modify the classical definitions related to TTS and TA by adding the architectural scope.

Definition 1 (*Architectural TTS*) An Architectural timed transition system $\mathcal{T}$ is a tuple $\langle S, \iota, \Sigma, \Phi, \rightarrow \rangle$ where S is a (possibly infinite) set of states, $\iota \in S$ is the initial state, Σ is a finite set of labels, Φ is a finite set of components and $\rightarrow \subseteq S \times \Sigma \cup \Phi \cup \mathbb{R}^{\geq 0} \times S$ is the transition relation where $\mathbb{R}^{\geq 0}$ is the set of positive real numbers. If $(q, \sigma, q') \in \rightarrow$, we write $q \xrightarrow{\sigma} q'$.

A trajectory of an Architectural TTS $\mathcal{T} = \langle S, \iota, \Sigma, \Phi, \rightarrow \rangle$ is a sequence $\pi = (s_0, t_0), \ldots, (s_k, t_k)$ such that for $0 \leq i \leq k$, $(s_i, t_i) \in S \times \mathbb{R}$ and for $\leq i < k$, $s_i \xrightarrow{\sigma} s_{i+1}$ and either $\sigma \in \Sigma \cup \Phi$ and $t_{i+1} = t_i$, or $\sigma \in \mathbb{R}^{>0}$ and $t_{i+1} = t_i + \sigma$. A state s of $\mathcal{T}$ is *reachable* if there exists a trajectory $\pi = (s_0, t_0), \ldots, (s_k, t_k)$ such that $s_0 = \iota$ and $s_n = s$.

Definition 2 (*Architectural TA*) An Architectural timed automaton is a tuple $\mathcal{A} = \langle$Loc, C, q_0, Lab, Comp, Edg$\rangle$ where:

– Loc is a finite set of locations representing the discrete states of the automaton.
– $C = \{c_1, \ldots, c_n\}$ is a finite set of real-valued variables.
– $q_0 = (l_0, v0)$ where $l_0 \in$ Loc is the initial location and v_0 is the initial clock valuation.
– Lab is a finite alphabet of labels.
– Comp is a the set of architectural constraints. Comp $\subseteq$ CompId $\times$ CompType. Where CompId represents the explicit component Id or "*" (to denote the *any component* scope) and CompType = { Process, Agent, Actor, Slot, team, etc. }.
– Edg $\subseteq$ Loc $\times$ Loc $\times$ G $\times$ Lab $\times$ Comp $\times 2^C$ is a set of edges. An edge (l, l', g, σ, c, R) represents a jump from location l to location l′ with guard g, event σ, component cp and a subset $R \subseteq C$ of variables to be reset.

Architectural constraints can be associated with any qualitative or quantitative temporal logic since no new operators are introduced. In what follows, we present the syntax and the semantics of what we name *ArTCTL*.

6.1 Architectural TCTL

We extend TCTL logic with an architectural dimension by associating an architectural scope to atomic propositions.

Definition 3 (*Syntax of ArTCTL formulas*) Let $\mathcal{A}$ be a timed automaton, $\mathcal{AP}$ a set of atomic propositions, *Comp* a set of architectural constraints (as defined in Definition 2) and D a non-empty set of clocks that is disjoint from the clocks of $\mathcal{A}$, i.e., $C \cap D = \emptyset$. $\sim$ denotes one of the binary relations $<, \leq, =, \geq, >$.

An ArTCTL formula ϕ has the following syntax rules.

$$\phi ::= p_{\text{cp}} \mid \neg\phi \mid \phi \vee \psi \mid z \times \phi \mid E[\phi U_{\sim c} \psi] \mid A[\phi U_{\sim c} \psi] \quad (18)$$

where $p \in \mathcal{AP}$, cp $\in$ *Comp*, and $z \in D$. z is called the freeze identifier and bounds the clock z in ϕ. For instance, using the freeze identifier the formula $A[\phi_{C1} U_{\leq 5} \psi_{C2}]$ can be defined by: z in $A[(\phi_{C1} \wedge z \leq 5) U \psi_{C2}]$

Definition 4 (*Semantics of ArTCTL*) The satisfaction relation $(A, s) \models \phi$ (i.e., ϕ is satisfied at state s in TA A) is defined inductively as follows:

- $A, s \models p_{\text{cp}}$ iff p is true in state s and satisfies constraint cp (i.e., let cp = (cpType, cpId), p is true within component cpId of type cpType)
- $A, s \models \neg\phi$ iff $A, s \not\models \phi$
- $A, s \models \phi \vee \psi$ iff either $A, s \models \phi$ or $A, s \models \psi$
- $A, s \models z.\phi$ iff $A, s\{z\} \models \phi$
- $A, s \models E[\phi U_{\sim c} \psi]$ iff there exists a run $(s_1, t_1)(s_2, t_2) \ldots$ such that $s_1 = s$ in A and there exist an $i \geq 1$ and a $\delta \in [0, t_{i+1} - t_i]$ such that
  - $A, s_i + \delta \models \psi$
  - for all j, δ', if either $(1 \leq j < i) \wedge (\delta' \in [0, t_{j+1} - t_j])$ or $(j = i) \wedge (\delta' \in [0, \delta))$, then $A, s_j + \delta' \models \phi$
- $A, s \models A[\phi U_{\sim c} \psi]$ iff for all runs $(s_1, t_1)(s_2, t_2) \ldots$ such that $s_1 = s$ in A and there exist an $i \geq 1$ and a $\delta \in [0, t_{i+1} - t_i]$ such that
  - $A, s_i + \delta \models \psi$
  - for all j, δ', if either $(1 \leq j < i) \wedge (\delta' \in [0, t_{j+1} - t_j])$ or $(j = i) \wedge (\delta' \in [0, \delta))$, then $A, s_j + \delta' \models \phi$

For instance, the absence property "*P does never occur within the component CpId of type CpType*" can be expressed in ArTCTL with: $AG(\neg P_{(CpId, CpType)})$.

7 Applying property patterns to IPHC case study

In this section, we apply our pattern system to the case study presented earlier in Sect. 4.4.

Requirement 1: An RTP packet is compressed in the fast path (ASIC) and the latency is less than 150 µs

This requirement is described in Fig. 25. Its corresponding ArTCTL formula is:

$$\neg E[\neg RTPpacket\ U\ (RTPpacket_{(Team,ASIC)} \wedge EG_{\leq 150} \neg RTPcompr)] \quad (19)$$

The design shown in Fig. 9 satisfies this property since RTP flows are compressed in the fast path (ASIC) and the latency is wihin the acceptable range (100 < 150 µs).

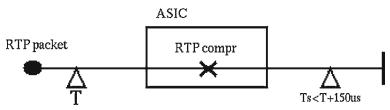


Fig. 25 Bounded existense property satisfying IPHC design

Requirement 2: A TCP packet is compressed in the fast path (ASIC) and the latency is less than 50 µs

This requirement is described in Fig. 26. Its corresponding ArTCTL formula is:

$$\neg E[\neg TCPpacket\ U\ (TCPpacket_{(Team,ASIC)} \wedge EG_{\leq 50} \neg TCPcompr)] \quad (20)$$

In this case the design in Fig. 9 violates this property since TCP flow compression takes place in slow path (software) and the latency is higher (100 > 50 µs) than the one specified in the property.

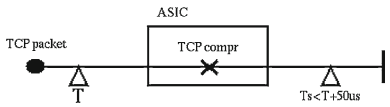


Fig. 26 Bounded existense property violating IPHC design

Requirement 3: In the compression scenario, the header lookup is followed by a protocol check

This requirement is described in Fig. 27. Its corresponding CTL formula is:

$$AG(HeaderLookup \Rightarrow AF(CheckProtocol)) \quad (21)$$

The design in Fig. 9 satisfies this property since responsibility *HeaderLookup* is followed by responsibility *CheckProtocol* for all paths that contain *HeaderLookup*.

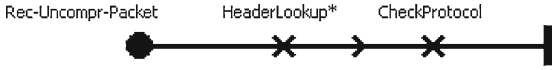


Fig. 27 Response property satisfies IPHC design

Requirement 4: IP options are not compressed

This requirement is described in Fig. 28. Its corresponding architectural CTL formula is:

$$AG(\neg IPOptionsCompression_{(Team,*)}) \quad (22)$$

The design in Fig. 9 satisfies this property since IP options packets are not compressed.

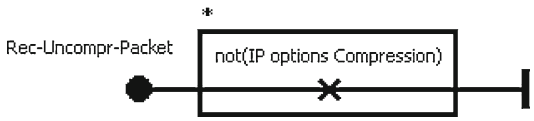


Fig. 28 Absence property satisfies IPHC design

Requirement 5: In the decompression scenario, packet drop is always preceeded by context ID storage

This requirement is described in Fig. 29. Its corresponding CTL formula is:

$$\neg E[\neg StoreContextID\ U(dropPacket \wedge \neg StoreContextID)] \quad (23)$$

This property is not satisfied since responsibilities *StoreContextID* and *dropPacket* belong to two distinct paths in Fig. 10.

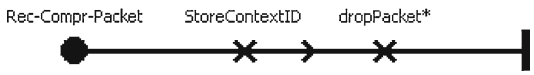


Fig. 29 Precedence property violating IPHC design

8 UCM-based specification–verification framework

Our ultimate goal is to use UCM to build system models that combine functional, architectural and temporal aspects of real-time systems, and then check their correctness. However, to the best of our knowledge none of the existing model checkers tolls consider architectural aspects. Designing architecture based model checking algorithms is left for future work.

Figure 30 illustrates the proposed UCM based Specification Verification Framework. System specifcations are expressed using the freely available UCM editing tool *jUCMNav* [24](an eclipse based open-source tool for editing and analysing URN models). These UCM models are then translated into a network of timed automata [20] that can be analyzed by the model checker UPPAAL [28]. We are currently in the process of integrating UCM property templates into *jUCMNav*. Users will be able to select properties from the UCM pattern catalogue, that can be then translated into TCTL formulas according to the UPPAAL format.

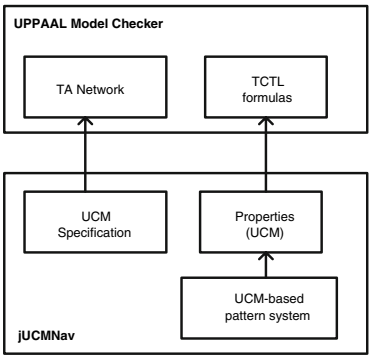


Fig. 30 UCM-based specification–verification framework

9 Conclusion

Specification building is one of the most difficult activities of model-based verification. Our work has yielded two main contributions. First, we have presented a UCM based specification pattern that can simplify this activity and make it available to the novice practitioner. The specification pattern system uses templates to cover most common expected properties found in requirements specifications. We provide a mapping of our UCM-based system to popular temporal logics CTL and TCTL. These templates combine qualitative, real-time and architectural properties into a single graphical representation. To the best of our knowledge, no existing pattern system has considered these three scopes together. However, we do not claim that our real-time specification pattern system is complete. Second, we extend the traditional real-time temporal logics to include architectural aspects. We give an overview of the semantics of the systems targeted by what we call "*Architectural real-time temporal logic*". We provide formal syntax and semantics of *ArTCTL*, an extension of TCTL with architectural aspects. We believe that having the requirement specification and properties described using the same formalism will enable greater degrees of analysis while preserving a high level of abstraction.

As part of our future work, we will evaluate the completeness and the effectiveness of our pattern system by surveying real-world specifications. We will also define a complete formal semantics for architectural real-time temporal logic and investigate the integration of architectural aspects into existing model checking algorithms.

References

1. Alfonso, A., Braberman, V.A., Kicillof, N., Olivero, A.: Visual timed event scenarios. In: 26th International Conference on Software Engineering (ICSE 2004), pp. 168–177 (2004)
2. Alur, R.: Techniques for automatic verification of real-time systems. PhD thesis, Stanford University (1991)
3. Alur, R., Dill, D.L.: A theory of Timed Automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
4. Alur, R., Henzinger, T.A.: Logics and models of real time: a survey. In: de Bakker, J., Huizing, K., de Roever, W.-P., Rozenberg, G. (eds.) Real Time: Theory in Practice, Lecture Notes in Computer Science, vol. 600, pp. 74–106. Springer, Heidelberg (1992)

5. Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. Inform. Comput. **104**(1), 35–77 (1993)
6. Amyot, D.: Use Case Maps Quick Tutorial. http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/WebHome (1999)
7. Amyot, D., Buhr, R.J.A., Gray, T., Logrippo, L.: Use Case Maps for the capture and validation of distributed systems requirements. In: RE'99, Fourth IEEE International Symposium on Requirements Engineering, pp. 44–53. Limerick, Ireland, June 1999. http://www.UseCaseMaps.org/pub/re99.pdf
8. Amyot, D., Andrade, R.: Description of wireless intelligent network services with Use Case Maps. In: SBRC'99, 17th Simposio Brasileiro de Redes de Computadores, pp. 418–433. Salvador, Brazil, May 1999
9. Bellini, P., Mattolini, R., Nesi, P.: Temporal logics for real-time system specification. ACM Comput. Surv. (CSUR) **32**(1), 12–42 (2000)
10. Braberman, V.A., Felder, M.: Verification of real-time designs: Combining scheduling theory with automatic formal verification. In: ESEC/SIGSOFT FSE, pp. 494–510, 1999
11. Buhr, R.J.A.: Use Case Maps as architectural entities for complex systems. IEEE Trans. Softw. Eng. **24**(12), 1131–1155 (1998)
12. Buhr, R.J.A., Elammari, M., Gray, T., Mankovski, S.: Applying Use Case Maps to multiagent systems: a feature interaction example. In: 31st Annual Hawaii International Conference on System Sciences, 1998
13. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **2**, 244–263 (1986)
14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, pp. 411–420. IEEE Computer Society Press (1999)
15. Dwyer, M., Avrunin, G., Corbett, J.: Property specification patterns for finite-state verification. In: Ardis, M. (ed.) Proceeedings of the Second Workshop on Formal Methods in Software Practices, pp. 7–15 (1998)
16. Flake, S., Muller, W.: Specification of real-time properties for UML models. In: Proceedings of the Hawai'i International Conference on System Sciences (HICSS-35), IEEE, Hawaii, USA (2002)
17. Graf, S., Ober, I.: Model checking of UML models via a mapping to communicating extended timed automata. In: Graf, S., Mounier, L. (eds.) Proceedings of SPIN'04 Workshop, Barcelona, Spain, LNCS 2989 (2004)
18. Gruhn, V., Laue, R.: Specification patterns for time-related properties. Temporal representation and reasoning. In: TIME 2005. 12th International Symposium on Volume, Issue, 23–25 June 2005, pp. 189–191
19. Hassine, J., Rilling, J., Dssouli, R.: An abstract operational semantics for Use Case Maps. In: Wang, F. (ed.) Formal techniques for networked and distributed systems–FORTE 2005. 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October, 2005, pp. 366–380. LNCS 3731 Springer, Heidelberg (2005)
20. Hassine, J., Rilling, J., Dssouli, R.: Formal verification of Use Case Maps with real time extensions. In: 13th SDL forum (SDL'07), Paris, France, September 2007 (to appear). LNCS, Springer, Heidelberg
21. Hassine, J., Rilling, J., Dssouli, R.: Timed Use Case Maps. In: Fifth Workshop on System Analysis and Modelling (SAM'06) Kaiserslautern, Germany, May (2006)
22. ITU-T, URN Focus Group: Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM). Geneva (2002)
23. Jansen, D.N., Wieringa, R.J.: Extending CTL with actions and real time. J. Logic Comput. **12** (4), 607–621. ISSN 0955-792X
24. jUCMNAV project: http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/WebHome. Last accessed, July 2007 (2006)
25. Konrad, S., Cheng, B.H.C.: Facilitating the construction of specification pattern based properties. In: Proceedings of the IEEE International Requirements Engineering Conference (RE05), Paris, France (2005)
26. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the International Conference on Software Engineering (ICSE05), St Louis, MO, USA, May (2005)
27. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990)
28. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997)
29. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer, New York, (1992)
30. Miga, A., Amyot, D., Bordeleau, F., Cameron, C., Woodside, M.: Deriving message sequence charts from Use Case Maps scenario specifications. In: Tenth SDL Forum (SDL'01), pp. 268–287. Copenhagen, 2001. LNCS 2078
31. Nakamura, N., Kikuno, T., Hassine, J., 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 (2000)
32. Nicola, R.D., Vaandrager, F.W.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) Proceedings Ecole de Printemps on Semantics of Concurrency, Lecture Notes in Computer Science, vol. 469, pp. 407–419 (1990)
33. Nicola, R.D., Fantechi, A., Gnesi, S., Ristori, G.: An action based framework for verifying logical and behavioural properties of concurrent systems. Computer Networks and ISDN Systems 25 (1993), pp. 761–778. In: Proceedings of 3rd Workshop on Computer Aided Verification (1991)
34. Ober, I., Kerbrat, A.: Verification of quantitative temporal properties of SDL specifications. In: SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML (2001), pp. 182–202
35. Olender, K., Osterweil L., Cecil: A sequencing constraint language for automatic static analysis generation. IEEE Trans. Softw. Eng. **16**(3), 268–280 (1990)
36. Petriu, D.C., Woodside, M.: Software performance models from system scenarios in Use Case Maps. In: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools, pp. 141–158, April 14–17, 2002
37. Ramakrishnan, S., McGregor, J.: Extending OCL to support temporal operators. In: 21st International Conference on Software Engineering (ICSE99), Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA, USA, May 1999
38. RFC 2507: IP header compression, February 1999. http://www.faqs.org/rfcs/rfc2507.html (1999)
39. RFC 2508: Compressing IP/UDP/RTP headers for low-speed serial links, February 1999. http://www.faqs.org/rfcs/rfc2508.html (1999)
40. RFC 1144: Compressing TCP/IP headers for low-speed serial links, Feb 1990. http://www.faqs.org/rfcs/rfc1144.html (1990)
41. Spec Patterns: SAnToS Laboratory. http://patterns.projects.cis.ksu.edu/documentation/patterns/ctl.shtml
42. Use Case Maps Web Page and UCM Users Group: http://www.UseCaseMaps.org (1999)
43. Tsai, W.T., Paul, R., Yu, L., Wei, X.: Rapid pattern-oriented scenario-based testing for embedded systems. In: Yang, H. (ed.) Software Evolution with UML and XML, pp. 222–262. IDEA Group Publishing, 2005
44. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. In: CAV 2001 Workshop on Software Model Checking Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Paris, France, vol. 55(3), of ENTCS, 2001

Author's biography

Jameleddine Hassine is a Ph.D candidate in Computer Science at Concordia University, Canada. Prior to this, he received his M.Sc. in Computer Science from the University of Ottawa, Canada in 2001. During his academic journey, Mr. Hassine held many research contracts and he has been a teaching assistant of many courses. His research interests are in Formal verification and validation of distributed systems, requirements engineering (languages and methods), software maintenance and Telecommunication Service Engineering. He is a Senior Verification Consultant at Cisco Systems, Canada since June 2005.

Juergen Rilling is an Associate Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. His major research focus is on traceability and software maintenance, with more than 50 refereed articles in software engineering, software maintenance, program comprehension, software traceability and requirement engineering. Dr. Rilling received his M.Sc. in computer science from the University of Reutlingen, Germany in 1991 and a second M.Sc. in computer science from the University of East Anglia, UK in 1993. He received his Ph.D. from the Illinois Institute of Technology, Chicago, USA in 1998. Dr. Rilling serves on the program committees of numerous conferences and workshops in software maintenance and program comprehension and as session chair at many conferences.

Dr. Rachida Dssouli is a full professor in the Department of Electrical and Computer Engineering since 2001. She received her Master (1978), DEA (1979), Doctorat de 3eme Cycle in Networking (1981) from l'UniversitT Paul Sabatier (Toulouse, France) and her Ph.D. in Computer Science from UniversitT de MontrTal. She published more than 150 papers in journals, and referred conferences in her area of research. She has published in IEEE Transaction on Software Engineering, IEEE Networks, International Journal of Web Information Systems, Information and Software Technology, Journal of Computer Networks, Computer Networks and ISDN Systems. Her research interests are in Communication Software Engineering, Testing based on Formal Methods, Requirements Engineering, Systems Engineering and Telecommunication Service Engineering. Dr. Dssouli was nominated Director of Concordia Institute for Information and systems engineering June 2002, her mandate was the creation of an interdisciplinary research and learning institute dedicated for graduate studies.