A Framework of Hierarchical Requirements Patterns for Specifying Systems of Interconnected Simulink/Stateflow Modules

Changyan Zhou, Ratnesh Kumar, Devesh Bhatt, Kirk Schloegel, Darren Cofer

Abstract

This paper presents an approach for specifying (functional) requirements for a certain class of dynamical systems using a hierarchical system of patterns. The application domain consists of the class of systems that can be modeled using Simulink/Stateflow. The approach is patterns-based and supports modularity and hierarchy to handle complex systems. Further, it supports both the formal analysis and the traceability of the requirements, and also facilitates the design process by being mapped to the underlying design space. The patterns are developed following a component-oriented paradigm, and as a result they also facilitate their reuse. We develop and present a number of requirements patterns for commonly used Simulink/Stateflow modules and show how they can be hierarchically composed. The approach is illustrated through a simple example of a localizer signal processing unit.

1 Introduction

Writing system/software requirements specifications is an important exercise in the software development process due to the evidence that requirements errors, such as incorrect, misunderstood or omitted requirements, are difficult to debug and expensive to correct later in the developmental life-cycle.

The importance of system requirements has led the researchers and software industry to develop approaches and tools for capturing and managing the system requirements. Examples of approaches include problem frames [6], intent specification [12], and multiple views [15]. Requirements documentation and management tools include DOORs, Requisite Pro, and Cradle. The higher-level requirements are generally expressed in natural language, diagrams [1], charts [5, 18], and tables, and lower-level ones in various specification/design/modeling languages such as UML, architecture description language (ADL), specification description language (SDL), and Hardware Description Language (HDL). Natural language is easily understandable but the use of it in describing complex, dynamic systems has the problems of ambiguity, inaccuracy, incompleteness, and inconsistency. Formal languages, on the other hand, can be difficult to learn and expensive to apply correctly since the requirements must be understood by both developers and customers, and many of them are not fluent in any formal language.

State-diagram/table notations have been demonstrated to offer a precise, relatively compact notation for specifying system requirements in a wide range of applications, including avionics systems. In addition, state-diagrams/tables can be assigned a precise mathematical semantics and thus can be analyzed mechanically to expose defects in requirements specifications. State-diagram/table notations based requirements specification languages include Software Cost Reduction (SCR) [11], Requirements State Machine Language (RSML) [14, 16] and its simplified versions RSML-e (RSML without event) [17], and SpecTRM-RL (Specification Toolkit and Requirements Methodology Requirements Language) [13]. Software tools utilizing the state-diagram/table notations include Scade/Lustre, StateChart/StateMate, Esterel.

Using state-diagram/table notations, the system requirements are primarily captured at discrete or logic level. While this can be appropriate for specifying certain types of systems at a high-level of abstraction, this may not be of much help for specifying requirements at a later phase of design, i.e., at a lower-level of abstraction that involves both discrete and continuous dynamics. The requirements patterns we develop support system specification at such lower levels of ab-
2 Our Approach to Requirements Specification

Different problem domains have different characteristics and as a result the design/modeling tools as well as the requirements specification for a system are domain specific. Simulink/Stateflow has become a widely accepted tool for modeling, design, and analysis of many of the input/output systems such as those appearing in aircraft, automobile, medical devices, power systems, nuclear systems, etc. The Simulink/Stateflow tool provides a library of components/modules (such as transfer functions, state-machines, common nonlinearities, superposition blocks, etc.), which can be interconnected in a hierarchical fashion to form an overall system. Thus the tool naturally supports modularity and hierarchy. Formalized requirements can be developed for commonly used and standardized Simulink/Stateflow modules. These formalized requirements are what we call requirements patterns.

A requirements pattern describes the inputs, the outputs, the behavior, and the constraints of basic Simulink/Stateflow module. It can be viewed as a template for specifying the requirement of such a module.

As mentioned in the introduction, this pattern-based approach is motivated by the ease of use, modularity, reuse, and modifiability. (Changes in the requirements can be made easily and in a cost-effective manner by replacing any of the existing patterns or their parameters with new ones.) To support traceability and complexity-management, the approach is modular and hierarchical. Finally the fact the patterns are formalized our approach supports a formal analysis (such as completeness, consistency, and correctness).

The domain of applicability of a system of requirements patterns is determined by the richness of the atomic patterns and their composition mechanisms. The set of basic Simulink/Stateflow modules we consider has been developed by the researchers at Honeywell over years for avionics and other embedded applications. These modules have been populated in a library called Honeywell Autocode Manager (HAM), which is a group of customized Simulink modules representing the primitive control system elements found in Honeywell avionic systems. Our current work is targeted at developing requirements patterns for the modules in the existing HAM library. We also develop composite patterns that are used to combine the basic ones.

A complex system is normally built using a hierarchy of several interconnected components/modules, which in turn may be built using subcomponents/submodules. For such systems, it is natural to develop requirements for the overall system as well as for the various subsystems and submodules appearing in the system. In our approach the requirement patterns of one level may be aggregated to form composite requirements patterns. The requirements for a system at a given level of design/abstraction can be specified using the basic and composite requirements patterns of that level. We identify requirement patterns for some of the basic Simulink/Stateflow modules that would ap-
pear at the lowest level of hierarchy. We also suggest some composite patterns that serve to specify requirements for the interconnection of modules using cascade, superposition, and feedback. Finally we specify how the basic and composite patterns can be used in certain conditional assertions to specify the requirements of a system formed using a hierarchical interconnection of the basic Simulink/Stateflow modules.

The desirable characteristics for requirements specifications include completeness, consistency, correctness, modifiability, testability, traceability, unambiguous-ness, and verifiability. Our approach exhibits most these traits. Associated with each requirements pattern is an input/output (black-box) behavior description, and also its temporal logic representation. Thus the approach is amenable to formal analysis. As an example the temporal logic property specifications can be easily extracted from the requirements patterns, and the requirements can be formally analyzed for consistency, completeness, and correctness. The use of requirements patterns also supports their reuse (the composite patterns already do this), and their modifiability. Also since the approach is hierarchical it supports traceability and complexity management. Furthermore, our requirements patterns are described in a natural language and can be understood by all users.

3 An Example

The example of a localizer signal processing unit presented here has been taken from the context of a flight control system developed by Honeywell Aerospace. The unit is used to filter the incoming signal when it is “valid for use”, and otherwise it produces a constant output. We next give the currently used requirements for the localizer signal processing unit, whereas Figure 1 shows the Simulink design model of the unit.

R1: The incoming localizer signal shall be declared valid for use by the system if the raw signal valid is true continuously for at least one second.

R2: The incoming localizer signal shall be declared invalid for use by the system if the raw signal valid is false continuously for at least 0.5 seconds.

R3: The incoming localizer signal shall be attenuated by -3dB at a frequency of 5 Hz and by -20dB at 50 Hz.

R4: When the raw localizer signal is invalid, the input to the attenuation filter shall be held to the last valid value of the raw input.

R5: When the incoming localizer signal is declared invalid for use by the system, the output of the value to the system shall be set to 600 microAmps.

From the system requirements and its Simulink model, we observe that the requirements R1 and R2 refer to the same Simulink module “Debounce”, R4 refers to two different Simulink modules, and R3 is mapped to the Simulink module “lag”. These references raise the following issues:

1. The case, in which two requirements reference the same module, a reasonable question is whether there exists any relation between R1 and R2 so they be written as a single requirement.

2. R4 maps to two different modules, whereas it is sensible to consider composing the associated Simulink sub-modules into a single, stand alone module, and defining a single requirement for that module.

3. It is questionable how the requirement R3 is mapped to the “lag” module with the specific transfer function $\frac{1}{0.0318s + 1}$, for only two points in the frequency domain are specified by R3. It seems that the designer is using a mental model containing additional detail for the lag module, which further implies that the requirement R3 is ambiguous and incomplete.

4. R4 seems to be not about what the system should do, rather how.

5. The requirements are written in a heuristic manner.

The approach we propose is able to resolve the issues raised above.

4 Basic Requirements Patterns

This section presents the patterns we have developed for specifying the requirements for the Simulink modules existing in the HAM library. Each pattern is associated with a requirement description and a temporal logic representation. The requirement description part describes the interface (inputs/outputs and their types), the parameters, the functionality, and the constraints. The functionality is written in a natural language so it can be easily understood by all users. The temporal logic representation part first describes the behavior in a language that is amenable for temporal logic specification. Next a temporal logic encoding
in first-order metric temporal logic is provided. (The syntax and semantics of the propositional metric temporal logic is included in the appendix for a quick reference.) Having been encoded within a requirements pattern, the temporal logic properties are ready-to-use for any formal analysis. The requirement description is standardized, and for a specific system only the corresponding inputs, outputs, and parameters need to be captured.

Some of the modules included in the HAM library are IsValid (also called debounce), Filter, Delay, Sample-&-Hold, Pulse Generator, Latch, Flip-flop, and Hysteresis. In the following, we present a sample of a few of the patterns associated with the individual modules. More patterns can be found in the report [19].

(i). IsValid Pattern

- Requirements Description:
  - Inputs: control input In(type)
  - Outputs: control output Out(type)
  - Parameters: debounce-on time tON(type), debounce-off time tOFF(type)
  - Functionality: When the control input is ON for at least debounce-on time, a certain signal shall be valid for use, when the control input is OFF for at least debounce-off time, the said signal shall be invalid for use.
  - Constraints: Initial value of the output must be set to either 0 or 1, and tON and tOFF must be the multiples of the sampling rate.

- Temporal Logic Representation:
  - Behavior: Initially, the output is set to either 0 or 1. It is always the case that if the input is 1 for at least tON time unit, then the output is set to 1, and if the input is 0 for at least tOFF time unit, then the output is set to 0.
  - Instantiation:
    \[
    (\text{Out.InitialValue} = 0 \lor 1) \land \square_{[0,\infty)}([\square_{[0,\infty)}{\text{In.Value}} = 1] \Rightarrow \diamond_{\leq 1}{\text{Out.Value}} = 1) \land ([\square_{[0,\infty)}{\text{In.Value}} = 0] \Rightarrow \diamond_{\leq 1}{\text{Out.Value}} = 0)
    \]

(ii). Filter Pattern

- Requirements Description:
  - Inputs: input signal In(type)
  - Outputs: output signal Out(type)
  - Parameters: passing band \([f_1, f_2]\) with attenuation \(K_1\) dB, roll-off band \([f_1 - \Delta, f_1]\) or \([f_2, f_2 + \Delta]\) with attenuation \(K_1\) dB/octave, attenuation \(K_2\) dB for non-passing band
  - Functionality: Within the pass-band \([f_1, f_2]\), the output shall be the input with attenuation of less than \(K_1\) dB, within the roll-off band \([f_1 - \Delta, f_1]\) and \([f_2, f_2 + \Delta]\), the output shall be attenuated at a rate of \(K_1\) dB/octave, and otherwise the output shall be attenuated more than \(K_2\) dB.
  - Constraints:

- Temporal Logic Representation:
  - Behavior: It is always the case that if the frequency is within the range of \([f_1, f_2]\), then the output signal shall be passed with attenuation of less than \(K_1\) dB; if the frequency is within the roll-off regions \([f_1 - \Delta, f_1]\) and \([f_2, f_2 + \Delta]\), then the output signal shall be
attenuated at a rate of $K_1$dB per octave; otherwise, the output signal shall be attenuated more than $K_2$db.

- Initialization:
  \[ \forall f, [f_1 \leq f \leq f_2] \implies Out.Magnitude(f)/InMagnitude(f) \leq K \land (f - \Delta \leq f \leq f_1) \lor (f_2 \leq f \leq f_2 + \Delta) \implies \frac{df}{dt}(Out.Magnitude(f)/InMagnitude(f)) \leq K_1 \land (f \leq f_1 - \Delta) \lor (f \geq f_2 + \Delta) \implies Out.Magnitude(f)/InMagnitude(f) \geq K_2 \]

(iii). **Delay Pattern**

- Requirements Description:
  - Inputs: input signal In(type)
  - Outputs: output signal Out(type)
  - Parameters: delay time T(type)
  - Description: Input shall be delayed for T time unit.
  - Constraints: T must be a multiple of the sampling rate.

- Temporal Logic Representation:
  - Behavior Description: Initially, the output signal is set to either ON or In. It is always the case that if the input signal is ON (resp., OFF) for at least T seconds, then the output signal is set to ON (resp., OFF), otherwise, the output is set to OFF (resp., ON).
  - Instantiation:
    \[ [(Out.InitialValue = 1) \lor (\lozenge_{[0, \infty)}(\square_{[0, T]} In.Value = 1) \implies \lozenge_{[0, \infty)}(\square_{[0, T]} In.Value = 0) \implies (Out.InitialValue = 0)] \]

(iv). **Sample-&-Hold Pattern**

- Requirements Description:
  - Inputs: input signal In(type), control signal CtrIn(type)
  - Outputs: output signal Out(type)
  - Parameters:
  - Functionality: The input signal shall be sampled if and only if it is valid for use.
  - Constraints: The output type is the same as the input type. The control input is compared to threshold value of 0.5 to determine if it is logic 0 or 1.

- Temporal Logic Representation:
  - Behavior: It is always the case that if the control input CtrIn is logic 1, then the output samples equals with the input signal, otherwise (i.e., CtrIn is logic 0), the output remains constant at its last sample value.
  - Instantiation:
    \[ \forall t, [(CtrIn = 1) \implies Out.Value(t) = In.Value(t)] \land [(CtrIn = 0) \implies Out.Value(t) = Out.Value(t - 1)] \]

(v). **Hysteresis Pattern**

- Requirements Description:
  - Inputs: input signal In(type)
  - Outputs: output signal Out(type)
  - Parameters: switch-on point UpLimit, switch-off point LoLimit, function f : Out \mapsto In
  - Functionality: When the input signal’s value is either above switch-on point or below switch-off point, the output shall be a function f of the input; otherwise, the output shall be 0.
  - Constraints:

- Temporal Logic Representation:
  - Behavior: It is always the case that if the input signal’s value is either above a switch-on point UpLimit or below a switch-off point LoLimit, then the output is a function f of the input, otherwise (i.e., the input’s value is between the switch-on and switch-off point), then the output is set to 0.
  - Instantiation:
    \[ \forall t, [(In.Value(t) > UpLimit) \lor (In.Value(t) < LoLimit) \implies Out.Value(t) = f(In.Value(t))] \land (LoLimit \leq In.Value(t) \leq UpLimit \implies Out.Value(t) = 0) \]

5 **Composite Requirements Patterns**

A complex system is normally composed of several interconnected components in manners such as cascade, superposition, and feedback. Correspondingly, requirements patterns can be composed to derive requirements patterns for an interconnected system of basic modules. Such patterns are called composite patterns. Here we present only three types of composite
patterns denote as, \textit{A then B} (cascade interconnection of modules A and B), \textit{superposition of A and B} (superposed interconnection of modules A and B), and \textit{A with B in feedback} (feedback interconnection of modules A and B). Other composition mechanisms, specially those of Stateflow modules, are also possible. A listing of those can be found in the report [19].

(a) \textbf{A then B Pattern}

- Requirements Description:
  - Inputs: \textit{input signal} In(type)
  - Outputs: \textit{output signal} Out(type)
  - Parameters:
  - Functionality: The composition \textit{shall be that of cascade-composition} A and B.
  - Constraints: input of A then B = input of A, output of A then B = output of B, and input of B = output of A.

- Temporal Logic Representation:
  - Behavior: It is always the case that the output Out is equal to the output of B, the input of B is equal to the output of A, and the input of A is equal to the input In.
  - Instantiation:
    \[
    \forall t, [In.Value(t) = A.In.Value(t) \land A.Out.Value(t) = B.In.Value(t) \land Out.Value = B.Out.Value(t)]
    \]

(b) \textbf{Superposition of A and B Pattern}

- Requirements Description:
  - Inputs: \textit{input signal} In(type)
  - Outputs: \textit{output signal} Out(type)
  - Parameters:
  - Functionality: The composition \textit{shall be that of superposition} A and B.
  - Constraints: input of superposition of A and B = input of A, input of A + B = input of B, and output of superposition of A and B = (output of A + output of B).

- Temporal Logic Representation:
  - Behavior: It is always the case that the output Out is equal to the summation of the outputs of A and B, the inputs of A and B is equal to the input In.
  - Instantiation:
    \[
    \forall t, [(In.Value(t) - B.Out.Value(t)) = A.In.Value(t) \land A.Out.Value(t) = B.In.Value(t)]
    \]

6 \textbf{Specifying System Requirements}

In our approach the requirements for a system can be specified using conditional assertions (or rules) of the form:

\begin{center}
\begin{tabular}{ll}
When inputs/past-outputs of the system satisfy a certain condition, the present/future outputs of the system shall be computed in accordance with certain basic or composite requirements patterns.
\end{tabular}
\end{center}

\textbf{Example 1} In the following we apply our requirements patterns to specify the requirements for the localizer signal processing unit described above.

\textbf{Localizer Signal Processing Unit Requirements:}

\textbf{Inputs:} rawLocalizer(float), rawLocalizerValid(boolean)

\textbf{Outputs:} Localizer(float)
High-level requirements:
When the input signal rawLocalizer is valid for use, output signal Localizer shall be sampled then filtered input. Otherwise, output shall be 600mA.

Low-level requirements:

Req1 (IsValid Pattern): When rawLocalizerValid is ON for at least 1s, the input signal rawLocalizer shall be valid for use, when rawLocalizer is OFF for at least 0.5s, the input signal rawLocalizer shall be invalid for use.

Req2 (Filter Pattern): Within the pass-band [0, 5], the output Localizer shall be the sampled input with attenuation of less than -3dB, within roll-off band [5, 50], the output Localizer shall be attenuated at a rate of -17/45dB/octave, and otherwise the output Localizer shall be attenuated more than -20dB.

Req3 (Sample-&-Hold Pattern): The rawLocalizer shall be sampled if and only if it is valid for use.

It follows from the above composite requirements that the localizer signal processing unit could consist of three basic HAM modules, namely, IsValid, Filter, and Sample-&-Hold. Also it is evident that the issues associated with the previously used requirements (refer Section 3) are largely resolved. Issue 1 is resolved by having a single requirement (Req1) that captures both the conditions under which the input signal is valid for use. Issue 2 is resolved by rephrasing R4 using a single Sample-&-Hold requirements pattern (Req3). This is made possible by taking a patterns-based approach to requirements. Issue 3 is resolved by using a Filter requirements pattern (Req2), which, by virtue of being pre-defined, insists that a complete set of filter parameters be provided. Issue 4 is resolved since our requirements only state what the system should do, and not how. Finally, issue 5 is resolved since we follow a systematic approach in which all requirements are formalized through the use of patterns.

7. Conclusion and Future Work

We proposed a hierarchical patterns-based approach for specifying requirements for the class of systems that can be represented using hierarchical interconnection of Simulink/Stateflow modules. The patterns-based approach is motivated from their ease of use and understanding in specifying requirements. To support modularity, modifiability, and reuse, the approach is component-based in the sense that patterns are developed for certain basic components (or modules). To support complexity management, the approach is hierarchical that uses certain “composition patterns” to generate higher-level composite patterns from the basic ones. This further supports traceability. To support verifiability and formal analysis (such as completeness, consistency, and correctness), patterns embed encodings of the input-output behaviors in the first-order metric temporal logic. Finally to support specification of systems possessing both discrete and continuous dynamics, patterns for certain commonly used Simulink/Stateflow modules developed by Honeywell researchers is provided. The patterns are standardized and in order to specify the requirements of a specific system, only the corresponding inputs, outputs, and parameters need to be specified.

The future work will include (1) Identifying requirements patterns for other useful modules (both the basic and the composite). We will focus on commonly used modules in the flight controls domain. (2) Developing a simple and high-level user interface for specifying the requirements using the proposed system of patterns. Users need as much tool support for developing requirements as they currently have for developing designs. (3) Developing approaches and tools for automatically performing formal analysis (such as completeness, consistency, correctness) of the requirements. Given that higher-level system requirements can be composed from lower-level requirements, and that these lower-level requirements have formal semantics defined for them, it should be feasible to automatically compose high-level formal requirements. The next challenge will be to automatically analyze these. (4) Developing approaches and tools for requirements-based design and test-generation. Given requirement patterns, how can we automatically generate tests from these. (5) We will also explore the relationship between the requirements at various levels of design abstraction, and how the steps of requirements specification, design, and refinement can be iterated to obtain a final set of requirements and design. This will, among other things, will require the integration of the non-functional requirements and the environmental constraints on the system design. (6) The following issues are of interest when specifying the system or software requirements: interfaces, functional capabilities, performance levels, data structures/elements, safety, reliability, security/privacy, quality, and constraints & limitations. (The first four refer to the functional requirements whereas the last five refer to the non-functional requirements.) Our approaches addresses the function-
ality issues (interface, functional capabilities, and data structure), and it will be interesting to also address the non-functional issues.

Acknowledgment: The work was done while the first author was a graduate research intern at Honeywell Technology Center, Minneapolis, MN in summer 2006, and the internship support of Honeywell is acknowledged. The second author acknowledges the support from NSF under the grants NSF-ECS-0218207, NSF-ECS-0244732, NSF-EPNES-0323379, NSF-ECS-0424048, and NSF-ECS-0601570.

References


A Metric Temporal Logic

Propositional metric temporal logic (MTL) is an extension of the propositional linear temporal logic (LTL) that can specify certain real-time properties. Its temporal operators include time-bounded versions of until, next, since, future and past. Given a finite set $P$ of propositions, the set of MTL formulas is defined using the following grammar.

$$
\phi := true | p \land \phi_1 \land \phi_2 | \phi_1 \land \phi_2 | \phi_1 \land \phi_2 U \psi | \phi_1 \land \phi_2 S \psi,
$$

where $p \in P$, and $I$ is one of the following:

1. An interval of the non-negative real line whose left and right end-points are natural numbers or $\infty$. For a number $n$, the expression $\pm I \pm n$ denotes the interval $\{\pm y \pm n | y \in I\} \cap [0, \infty]$.

2. A relative congruence expression $\approx_d c$ for integers $d \geq 2$ and $c \geq 0$. $y \in \approx_d c$ denotes $y = c \mod d$, and $\pm I \pm n$ the set $\{y | y = \pm c \pm n \mod d\}$. 

3. An absolute congruence expression $=_{d} c$ for integers $d \geq 2$ and $c \geq 0$. The expression $y \in=_{d} c$ denotes $y = c \mod d$ and $\pm I \pm n$ the set $\{y | y = c \mod d\}$.

We assume that the integer constants that occur in a formula are encoded in binary format.

We interpret MTL formulas over finite timed state sequences. A timed state sequence $\rho = (\pi, t)$ is a pair consisting of a finite sequence $\pi$ of states $\pi_i \subseteq P$, and a finite sequence of real numbers $\tau$ with $|\pi| = |\tau|$ and $\pi_i \leq \pi_{i+1}$ for each $i$. Define $|\rho| = |\pi|$. Intuitively, a sequence $\rho$ represents a timed execution of a system and is understood as follows: at time $\tau_i$ the system was observed to be in state $\pi_i$. Let $\pi[i, j]$ denote $\pi_i, \pi_{i+1}, \ldots, \pi_j$, and similarly for $\tau[i, j]$, and let $\rho[i, j] = (\pi[i, j], \tau[i, j])$. Given a timed state sequence $\rho$ and a position $1 \leq i \leq \rho$, we define what it means for $(\rho, i)$ to satisfy a formula $\phi$, written $(\rho, i) \models \phi$, as follows:

$(\rho, i) \models true$ is always true
$(\rho, i) \models false$ is always false
$(\rho, i) \models p$ iff $p \in \pi_i$
$(\rho, i) \models \phi_1 \land \phi_2$ iff $(\rho, i) \models \phi_1$ and $(\rho, i) \models \phi_2$
$(\rho, i) \models \circ_I \phi$ iff $i < |\rho|$, $(\rho, i + 1) \models \phi$ and $\tau_{i+1} \in \tau_i + I$
$(\rho, i) \models \circ_I \phi$ iff $i > 1$, $(\rho, i - 1) \models \phi$ and $\tau_{i-1} \in \tau_i - I$
$(\rho, i) \models \phi U_I \psi$ iff $(\rho, j) \models \psi$ for some $j \geq i$
with $\tau_j \in \tau_i + I$ and $(\rho, k) \models \phi$ for all $i \leq k < j$
$(\rho, i) \models \phi S_I \psi$, iff $(\rho, j) \models \psi$ for some $j \leq i$
with $\tau_j \in \tau_i - I$ and $(\rho, k) \models \phi$ for all $j < k \leq i$

The defined operators $\diamond_I \phi$ (time-bound eventually) and $\square_I \phi$ (time-bound always) stand for true $U_I \phi$ and $\neg \diamond_I \neg \phi$, respectively. It follows that the formula $\diamond_I \phi$ (or $\square_I \phi$) holds at time $t \in R$ of a timed observation sequence iff $\phi$ holds at all times (at some time, respectively) within the interval $t + I$.

Note that the standard LTL falls as a degenerate sublogic of MTL where only the interval $[0, \infty)$ is allowed.