On Implementing Real-Time Specification Patterns Using Observers

John D. Backes[1](✉), Michael W. Whalen[2], Andrew Gacek[1], and John Komp[2]

[1] Rockwell Collins, Bloomington, MN 55438, USA
john.backes@gmail.com
[2] University of Minnesota, Minneapolis, MN 55455, USA

Abstract. English language requirements are often used to specify the behavior of complex cyber-physical systems. The process of transforming these requirements to a formal specification language is often challenging, especially if the specification language does not contain constructs analogous to those used in the original requirements. For example, requirements often contain real-time constraints, but many specification languages for model checkers have discrete time semantics. Work in specification patterns helps to bridge these gaps, allowing straightforward expression of common requirements patterns in formal languages. In this work we demonstrate how we support real-time specification patterns in the Assume Guarantee Reasoning Environment (AGREE) using observers. We demonstrate that there are subtle challenges, not mentioned in previous literature, to express real-time patterns accurately using observers. We then demonstrate that these patterns are sufficient to model real-time requirements for a real-world avionics system.

1 Introduction

Natural language requirements specifications are often used to prescribe the behavior of complex cyber-physical systems. Regrettably, such specifications can be incomplete, inconsistent, or ambiguous. For these reasons, researchers have long advocated the use of formal languages, such as temporal logics to describe requirements. Unfortunately, the process of formalizing natural language requirements using formal specification languages is often challenging, especially if the specification language does not contain constructs analogous to those used in the original requirements.

Specification patterns [1,2] are an approach to ease the construction of formal specifications from natural language requirements. These patterns describe how common reasoning patterns in English language requirements can be represented in (sometimes complex) formulas in a variety of formalisms. Following the seminal work of Dwyer [1] for discrete time specification patterns, a

This work was sponsored by DARPA/AFRL Contract FA8750-12-9-0179, AFRL Contract FA8750-16-C-0018, and NASA Contract NNA13AA21C.

© Springer International Publishing Switzerland 2016

S. Rayadurgam and O. Tkachuk (Eds.): NFM 2016, LNCS 9690, pp. 19–33, 2016.
DOI: 10.1007/978-3-319-40648-0_2

variety of real-time specification pattern taxonomies have been developed [2–6]. An example of a timed specification pattern expressible in each is: "Globally, it is always the case that if P holds, then S holds between *low* and *high* time unit(s)."

In most of this work, the specification patterns are mapped to real-time temporal logics, such as TCTL [7], MTL [8], RTGIL [9], and TILCO-X [4]. As an alternative, researchers have investigated using *observers* to capture real-time specification patterns. Observers are code/model fragments written in the modeling or implementation language to be verified, such as timed automata, timed Petri nets, source code, and Simulink, among others. For example, Gruhn [3] and Abid [10] describe real-time specifications as state machines in timed automata and timed Petri nets, respectively. A benefit of this approach is that rather than checking complex timed temporal logic properties (which can be very expensive and may not be supported by a wide variety of analysis tools), it is possible to check simpler properties over the observer.

Despite this benefit, capturing real-time specification patterns with observers can be challenging, especially in the presence of overlapping "trigger events." That is, if P occurs multiple times before *low* time units have elapsed in the example above. For example, most of the observers in Abid [10] explicitly are not defined for 'global' scopes, and Gruhn, while stating that global properties are supported, only checks a pattern for the first occurrence of the triggering event in an infinite trace.

In this work, we examine the use of observers and invariant properties to capture specification patterns that can involve overlapping triggering events. We use the Lustre specification language [11] to describe *synchronous observers* involving a real-valued time input to represent the current system clock[1]. We describe the conditions under which we can use observers to faithfully represent the semantics of patterns, for both positive instances of patterns *and negations of patterns*. We call the former use *properties* and the latter use *constraints*.

The reason that we consider negations of patterns is that our overall goal is to use real-time specification patterns in the service of assume/guarantee compositional reasoning. In recent efforts [12,13], we have used the AGREE tool suite [14] for reasoning about discrete time behavioral properties of complex models described in the Architectural Analysis and Design Language [15][2]. Through adding support for Requirements Specification Language (RSL) patterns [16] and calendar automata [17–19], it becomes possible to lift our analysis to real-time systems. In AGREE, we prove implicative properties: given that subcomponents satisfy their contracts, then a system should satisfy its contract. This means that the RSL patterns for subsystems are used under a negation. We describe the use of these patterns in AGREE and demonstrate their use on a real avionics system. Thus, the contributions of this work are as follows:

[1] Although our formalisms are expressed as Lustre specifications, the concepts and proofs presented in this paper are applicable to many other popular model checking specification languages.

[2] AGREE is available at: http://loonwerks.com.

- We demonstrate a method for translating RSL Patterns into Lustre observers and system invariants.
- We prove that it is possible to efficiently capture patterns involving arbitrary overlapping intervals in Lustre using non-determinism.
- We argue that there is no method to efficiently encode a transition system in Lustre that implements the exact semantics of all of the RSL patterns when considering their negation.
- We demonstrate how to encode these patterns as Lustre constraints for *practical* systems.
- We discuss the use of these patterns to model a real-world avionics system.

2 Definitions

AGREE proves properties of architectural models compositionally by proving a series of lemmas about components at different levels in the model's hierarchy. A description of how these proofs are constructed is provided in [12,14] and a proof sketch of correctness of these rules is described in [14,20]. For the purpose of this work, it is not important that the reader has an understanding of how these proofs are constructed. The AGREE tool translates AADL models annotated with component assumptions, guarantees, and assertions into Lustre programs. Our explanations and formalizations in this paper are described by these target Lustre specifications. Most other SMT-based model checkers use a specification language that has similar expressivity as Lustre; the techniques we present in this paper can be applied generally to other model checking specification languages.

A Lustre program $\mathcal{M} = (V, T, P)$ can be thought of as a finite collection of named variables V, a transition relation T, and a finite collection of properties P. Each named variable is of type *bool*, *integer*, or *real*. The transition relation is a Boolean constraint over these variables and theory constants; the value of these variables represents the program's current *state*, and the transition relation constrains how the state changes. Each property $p \in P$ is also a Boolean constraint over the variables and theory constants. We sometimes refer to a Lustre program as a model, specification, or transition system. The AGREE constraints specified via assumptions, assertions, or guarantees in an AADL model are translated to either constraints in the transition relation or properties of the Lustre program.

The expression for T contains common arithmetic and logical operations ($+$, $-$, $*$, $\div$, $\vee$, $\wedge$, $\Rightarrow$, $\neg$, $=$) as well as the "if-then-else" expression (*ite*) and two temporal operations: $\rightarrow$ and *pre*. The $\rightarrow$ operation evaluates to its left hand side value when the program is in its initial state. Otherwise it evaluates to its right hand side value. For example, the expression: $true \rightarrow false$ is $true$ in the initial state and $false$ otherwise. The *pre* operation takes a single expression as an argument and returns the value of this expression in the previous state of the transition system. For example, the expression: $x = (0 \rightarrow pre(x) + 1)$ constrains the current value of variable x to be 0 in the initial state otherwise it is the value of x in the previous state incremented by 1.

In the model's initial state the value of the *pre* operation on any expression is undefined. Every occurrence of a *pre* operator must be in a subexpression of the right hand side of the → operator. The *pre* operation can be performed on expressions containing other *pre* operators, but there must be → operations between each occurrence of a *pre* operation. For example, the expression: $true \rightarrow pre(pre(x))$ is not well-formed, but the expression: $true \rightarrow pre(x \rightarrow pre(x))$ is well-formed.

A Lustre program models a state transition system. The current values of the program's variables are constrained by values of the program's variables in the previous state. In order to model timed systems, we introduce a real-valued variable t which represents how much time has elapsed during the previous transitions of the system. We adopt a similar model as *timeout automata* as described in [17]. The system that is modeled has a collection of *timeouts* associated with the time of each "interesting event" that will occur in the system. The current value of t is assigned to the least timeout of the system greater than the previous elapsed time. Specifically, t has the following constraint:

$$t = 0 \rightarrow pre(t) + min\_pos(t_1 - pre(t), \ldots, t_n - pre(t)) \tag{1}$$

where $t_1, \ldots, t_n$ are variables representing the timeout values of the system. The function *min_pos* returns the value of its minimum positive argument. We constrain all the timeouts of the system to be positive. A timeout may also be assigned to positive infinity (∞)[3]. There should always be a timeout that is greater than the current time (and less than ∞). If this is true, then the invariant $true \rightarrow t > pre(t)$ holds for the model, i.e., time always progresses.

A sequence of states is called a *trace*. A trace is said to be *admissible* (w.r.t. a Lustre model or transition relation) if each state and its successor satisfy the transition relation. We adopt the common notation (σ, τ) to represent a trace of a timed system where σ is a sequence of states $(\sigma = \sigma_1\sigma_2\sigma_3\ldots)$ and τ is a sequence of time values $(\tau = \tau_1\tau_2\tau_3\ldots)$ such that $\forall i : \tau_i < \tau_{i+1}$. In some literature, state transitions may take place without any time progress (i.e., $\forall i : \tau_i \leq \tau_{i+1}$). We do not allow these transitions as it dramatically increases the complexity of a model's Lustre encoding.

A Lustre program implicitly describes a set of admissible traces. Each state σ_n in the sequence represents the value of the variables V in state n. Each time value τ_n represents the value of the time variable t in state n. We use the notation $\sigma_n \models e$, where e is Lustre expression over the variables V and theory constants, if the expression e is satisfied in the state σ_n. Similarly, we use $\sigma_n \not\models e$ when e is not satisfied in the state σ_n. A property p is true (or invariant) in a model if and only if for every admissible trace $\forall n : \sigma_n \models p$. For the purposes of this work, we only consider models that do not admit so-called "Zeno traces" [21]. A trace (σ, τ) is a Zeno trace if and only if $\exists v \forall i : \tau_i < v$, i.e., time never progresses beyond a fixed point.

[3] In practice, we allow a timeout to be a negative number to represent infinity. This maintains the correct semantics for the constraint for t in Formula 1.

3 Implementing RSL Patterns

3.1 Formalizing RSL Patterns Semantics

For this work, we chose to target the natural language patterns proposed in the CESAR project because they are representative of many types of natural language requirements [16]. These patterns are divided into a number of categories. The categories of interest for this work are the *functional patterns* and the *timing patterns*. Some examples of the functional patterns are:

1. **Whenever** event **occurs** event **occurs during** interval
2. **Whenever** event **occurs** condition **holds during** interval
3. **When** condition **holds during** interval event **occurs during** interval
4. **Always** condition

Some examples of timing patterns are:

1. Event **occurs each** period [**with jitter** jitter]
2. Event **occurs sporadic with IAT** interarrivaltime [**and jitter** jitter]

Generally speaking, the timing patterns are used to constrain how often a system is required to respond to events. For instance, a component that listens to messages on a shared bus might assume that new messages arrive at most every 50 ms. The second timing pattern listed above would be ideal to express this assumption. In AGREE, this requirement may appear as a system assumption using the pattern shown in Fig. 1.

new_message **occurs sporadic with IAT** 50.0

Fig. 1. An instance of a timing pattern to represent how frequently a message arrives on a shared bus.

The functional patterns can be used to describe how the system's state changes in response to external stimuli. Continuing with the previous example, suppose that the bus connected component performs some computation whenever a new message arrives. The functional patterns can be used to describe when a thread is scheduled to process this message and how long the thread takes to complete its computation. The intervals in these patterns have a specified lower and upper bound, and they may be open or closed. The time specified by the lower and upper bound corresponds to the time that progresses since the triggering event occurs. Both the lower and upper bounds must be positive real numbers, and the upper bound must be greater than or equal to the lower bound. An AGREE user may specify the instances of patterns shown in Fig. 2 as properties she would like to prove about this system. For the purposes of demonstration we assume that the thread should take 10 ms to 20 ms to execute.

Always *new_message* = *thread_start*
Whenever *thread_start* **occurs** *thread_stop* **occurs during** [10.0, 20.0]

Fig. 2. Two instances of a functional patterns used to describe when a thread begins executing, and how long it takes to execute.

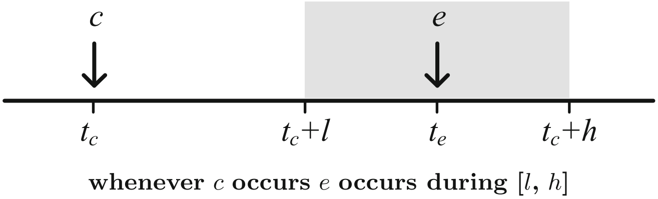

Fig. 3. A graphical representation for the RSL pattern

Figure 3 shows a graphical representation of the first functional pattern listed at the beginning of this section. The variable t_c represents the time that event c occurs. Similarly, the variable t_e represents the time that event e occurs. The formal semantics for many of the RSL patterns are described in [5]. The semantics for the pattern described in Fig. 3 are represented by the set of admissible traces $\mathcal{L}_{patt}$ described below.

$$\mathcal{L}_{patt} = \{(\sigma, \tau) \mid \forall i \exists j : \sigma_i \models c \Rightarrow (j > i) \wedge (\tau_i + l \leq \tau_j \leq \tau_i + h) \wedge (\sigma_j \models e)\}$$

The remainder of this section discusses how the pattern in Fig. 3 can be translated into either a Lustre property or a constraint on the admissible traces of a transition system described by Lustre. Although we discuss only this pattern, the techniques that we present can be applied generally to all except one of the functional and timing RSL patterns[4].

3.2 Implementing RSL Patterns as Lustre Properties

One can determine if a transition system described in Lustre admits only traces in $\mathcal{L}_{patt}$ by adding additional constraints over fresh variables (variables that are not already present in the program) to the model. This commonly used technique is referred to as adding an *observer* to the model. These constraints are over fresh variables: $run, timer, rec_c$ and *pass*; they are shown in Fig. 4. The constraints only restrict the values of the fresh variables, therefore they do not restrict the traces admissible by the transition relation.

The intuition behind these constraints is that one can record how much time progresses since an occurrence of c. This time is recorded in the *timer* variable. The value of the timer variable only increases if the previous value of the *run*

[4] The single pattern that cannot be implemented requires an independent event to occur for each of an unbounded number of causes. There are 12 functional and timing RSL patterns in total.

$$\begin{array}{ll}
1. & run = (rec_c \rightarrow ite(pre(run) \wedge e \wedge l \leq timer \leq h, \\
 & \qquad\qquad\qquad\qquad false, \\
 & \qquad\qquad\qquad\qquad ite(rec_c, true, pre(run)))) \\
2. & timer = (0 \rightarrow ite(pre(run), pre(timer) + (t - pre(t)), 0)) \\
3. & rec_c \Rightarrow c \\
4. & pass = (timer \leq h)
\end{array}$$

Fig. 4. The constraints added to a transition relation to verify if only the traces of $\mathcal{L}_{patt}$ are admissible. The transition relation only admits traces of $\mathcal{L}_{patt}$ if and only if the variable *pass* is invariant.

variable is true. The *run* variable is true if an occurrence of c is *recorded* and no occurrence of e happens until after the timer counts to at least l. The variable rec_c non-deterministically records an occurrence of c. If the transition system admits a trace outside of $\mathcal{L}_{patt}$, then the rec_c variable can *choose* to record only an event that violates the conditions of $\mathcal{L}_{patt}$. In this case the *pass* variable will become false in some state.

Theorem 1. *Let $\mathcal{L}_M$ represent the admissible traces of a transition system containing the constraints of Fig. 4. The transition system admits only traces in $\mathcal{L}_{patt}$ if and only if the property pass is invariant. Formally: $(\mathcal{L}_M \subseteq \mathcal{L}_{patt}) \Leftrightarrow (\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass)$*

Proof. First we show that if *pass* is invariant for a trace of the transition relation, then that trace is in $\mathcal{L}_{patt}$.

Lemma 1. $(\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass) \Rightarrow (\mathcal{L}_M \subseteq \mathcal{L}_{patt})$.

Proof. Towards contradiction, assume $\mathcal{L}_M \nsubseteq \mathcal{L}_{patt}$. Let (σ, τ) be a trace in $\mathcal{L}_M$ but not in $\mathcal{L}_{patt}$. Since $(\sigma, \tau) \notin \mathcal{L}_{patt}$, by definition there exists i such that $\sigma_i \models c$ and

$$\forall j : (j > i) \wedge \tau_i + l \leq \tau_j \leq \tau_i + h \Rightarrow \sigma_j \not\models e. \tag{2}$$

Without loss of generality, we can assume that this is the only time when c is recorded. That is, $\sigma_i \models rec_c$ and $\forall k : k \neq i \Rightarrow \sigma_k \not\models rec_c$. From constraint 1 in Fig. 4 we have

$$\forall j : ((j < i) \Rightarrow \sigma_j \not\models run) \wedge ((\tau_i \leq \tau_j < \tau_i + l) \Rightarrow \sigma_j \models run)$$

This can actually be strengthened more. From Formula 2 the event e does not occur between $\tau_i + l$ and $\tau_i + h$. So the variable *run* will become invariant after τ_i.

$$\forall j : ((j < i) \Rightarrow \sigma_j \not\models run) \wedge (\tau_i \leq \tau_j) \Rightarrow \sigma_j \models run)$$

From this and constraint 2 in Fig. 4, we have

$$\forall j : (j \leq i) \Rightarrow \sigma_j \models timer = 0$$

and

$$\forall j : (\tau_i < \tau_j) \Rightarrow (\sigma_j \models timer = (pre(timer) + (\tau_j - \tau_{j-1})))$$

From this and the invariant $\forall i : \tau_{i+1} > \tau_i$, we have

$$\forall j : (\tau_i < \tau_j) \Rightarrow (\sigma_j \models timer > pre(timer))$$

Therefore since the value of timer is zero before τ_i and always increasing after τ_i, and since we only consider non-Zeno traces ($\forall v \exists i : v < \tau_i$), eventually $timer > h$ and so *pass* becomes false. This contradicts the assumption $(\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass)$. Therefore $\mathcal{L}_M \subseteq \mathcal{L}_{patt}$. □

Next we show if a trace of $\mathcal{L}_M$ is in $\mathcal{L}_{patt}$, then *pass* is invariant for this trace.

Lemma 2. $(\mathcal{L}_M \subseteq \mathcal{L}_{patt}) \Rightarrow (\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass)$

Proof. Towards contradiction, assume that there exists a trace of $\mathcal{L}_M$ for which *pass* is not invariant. This means that for some state $\sigma_j \models timer > h$. For this to be true, the timer must be running continuously since it started with some recorded occurrence of c. That is there exists i such that $\sigma_i \models timer = 0$, $\sigma_i \models rec_c$, $\sigma_i \models c$, $\forall k : i \leq k < j \Rightarrow \sigma_k \models run$, and $\tau_j - \tau_i > h$. Thus $\forall k : i \leq k \leq j \Rightarrow \sigma_k \models timer = \tau_k - \tau_i$. By the definition of $\mathcal{L}_{patt}$ we have a k such that $\tau_i + l \leq \tau_k \leq \tau_i + h$ and $\sigma_k \models e$. This means $l \leq \tau_k - \tau_i \leq h$ and so $\sigma_k \models l \leq timer \leq h$. Therefore $\sigma_k \not\models run$. We also have $\tau_k \leq \tau_i + h < \tau_j$ so that $k < j$. Thus from $\forall k : i \leq k < j \Rightarrow \sigma_k \models run$ we have $\sigma_k \models run$ which is a contradiction. Therefore, *pass* is invariant. □

From Lemmas 1 and 2 we have $(\mathcal{L}_M \subseteq \mathcal{L}_{patt}) \Leftrightarrow (\forall \sigma, \tau, i : (\sigma, \tau) \in \mathcal{L}_M \Rightarrow \sigma_i \models pass)$. □

3.3 Implementing RSL Patterns as Lustre Constraints

As we demonstrated with Fig. 4, one can specify a Lustre property that verifies whether or not some transition system only admits traces of $\mathcal{L}_{patt}$. However, it is surprisingly non-trivial to actually implement a transition system that admits *exactly* the traces of $\mathcal{L}_{patt}$. Naively, one could attempt to add the constraints of Fig. 4 to a transition system and then assert that *pass* is invariant. However, this transition system will admit all traces where every occurrence of c is never recorded ($\forall \sigma_i : \sigma_i \not\models rec_c$). Clearly some of these traces would not be in $\mathcal{L}_{patt}$.

We conjecture that given the Lustre expression language described in Sect. 2 it is not possible to model a transition system that admits only and all of the traces of $\mathcal{L}_{patt}$. The intuition behind this claim is that Lustre specifications contain a fixed number of state variables, and variables have non-recursive types. Thus a Lustre specification only has a finite amount of memory (though it can, for example, have arbitrary sized integers). If a Lustre specification has n variables we can always consider a trace in $\mathcal{L}_{patt}$ where event c occurs more than n times in a tiny interval. In order for the pattern to hold true, the Lustre specification must constrain itself so that at least one occurrence of e occurs precisely between $t_c + l$ and $t_c + h$ after each event c. This requires "more memory" than the Lustre specification has available.

Rather than model the exact semantics of this pattern, we choose to take a more pragmatic approach. We model a strengthened version of Fig. 3 which does not allow overlapping instances of the pattern. That is, after an event c there can be no more occurrences of c until the corresponding occurrence of e. We do this by proving that c cannot occur frequently enough to cause an overlapping occurrence of the pattern. Then if we constrain the system based on a simple non-overlapping check of the pattern, the resulting system is the same as if we had constrained it using the full pattern. This simple non-overlapping check and the property limiting the frequency of c are both easily expressed in Lustre since they only look back at the most recent occurrence of c. Moreover, they can both be used freely in positive and negative contexts. Formally, the property we prove is $\mathcal{L}_{prop}$ and the constraints we make are $\mathcal{L}_{cons}$:

$$\mathcal{L}_{prop} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models c \Rightarrow \forall j : (j > i) \wedge (\tau_j \leq \tau_i + h) \wedge \sigma_j \models c \Rightarrow \\ \exists k \in (i, j] : \tau_i + l \leq \tau_k \wedge \sigma_k \models e\}$$

$$\mathcal{L}_{cons} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models c \Rightarrow \exists j : (j > i) \wedge \\ [(\tau_i + l \leq \tau_j \leq \tau_i + h \wedge \sigma_j \models e) \vee (\tau_j \leq \tau_i + h \wedge \sigma_j \models c)]\}$$

The correctness of $\mathcal{L}_{prop}$ and $\mathcal{L}_{cons}$ are captured by the following theorem.

Theorem 2. *Let M be a transition system and $\mathcal{L}_M$ its corresponding set of admissible traces. Suppose $\mathcal{L}_M \subseteq \mathcal{L}_{prop}$. Then $\mathcal{L}_{cons}$ and $\mathcal{L}_{patt}$ are equivalent restrictions on $\mathcal{L}_M$, that is $\mathcal{L}_M \cap \mathcal{L}_{cons} = \mathcal{L}_M \cap \mathcal{L}_{patt}$.*

Proof. We prove the theorem by showing that the subset relationship between $\mathcal{L}_M \cap \mathcal{L}_{cons}$ and $\mathcal{L}_M \cap \mathcal{L}_{patt}$ holds in both directions.

Lemma 3. $\mathcal{L}_M \cap \mathcal{L}_{patt} \subseteq \mathcal{L}_M \cap \mathcal{L}_{cons}$

Proof. From the definitions of $\mathcal{L}_{patt}$ and $\mathcal{L}_{cons}$ it follows directly that $\mathcal{L}_{patt} \subseteq \mathcal{L}_{cons}$. Therefore $\mathcal{L}_M \cap \mathcal{L}_{patt} \subseteq \mathcal{L}_M \cap \mathcal{L}_{cons}$. □

Lemma 4. *Suppose $\mathcal{L}_M \subseteq \mathcal{L}_{prop}$, then $\mathcal{L}_M \cap \mathcal{L}_{cons} \subseteq \mathcal{L}_M \cap \mathcal{L}_{patt}$*

Proof. Suppose towards contradiction that $\mathcal{L}_M \cap \mathcal{L}_{cons} \not\subseteq \mathcal{L}_M \cap \mathcal{L}_{patt}$. Consider a trace $(\sigma, \tau) \in \mathcal{L}_M \cap \mathcal{L}_{cons}$ with $(\sigma, \tau) \notin \mathcal{L}_M \cap \mathcal{L}_{patt}$. Then we have $(\sigma, \tau) \in \mathcal{L}_{cons}$, $(\sigma, \tau) \in \mathcal{L}_{prop}$, and $(\sigma, \tau) \notin \mathcal{L}_{patt}$. From the definition of $\mathcal{L}_{patt}$ we have an i such that $\sigma_i \models c$ and

$$\forall j : (j > i) \wedge (\tau_i + l \leq \tau_j \leq \tau_i + h) \Rightarrow \sigma_j \not\models e. \tag{3}$$

Then from the definition of $\mathcal{L}_{cons}$ with $\sigma_i \models c$ we have a j such that $j > i$ and either $(\tau_i + l \leq \tau_j \leq \tau_i + h \wedge \sigma_j \models e)$ or $(\tau_j \leq \tau_i + h \wedge \sigma_j \models c)$. The former option directly contradicts Formula 3, so we must have $\tau_j \leq \tau_i + h$ and $\sigma_j \models c$. From the definition of $\mathcal{L}_{prop}$ with $\sigma_i \models c$ and our j, we have a k in $(i, j]$ such that $\tau_i + l \leq \tau_k$ and $\sigma_k \models e$. From $k \leq j$ we have $\tau_k \leq \tau_j$ and thus $\tau_i + l \leq \tau_k \leq \tau_i + h$. Instantiating Formula 3 with k yields $\sigma_k \not\models e$, a contradiction. Therefore $\mathcal{L}_M \cap \mathcal{L}_{cons} \subseteq \mathcal{L}_M \cap \mathcal{L}_{patt}$. □

From Lemmas 3 and 4 have $\mathcal{L}_M \cap \mathcal{L}_{cons} = \mathcal{L}_M \cap \mathcal{L}_{patt}$. □

Example 1. Suppose we want to model a system of components communicating on a shared bus. The transition relation for this system must contain constraints that dictate when threads can start and stop and how frequently new messages may arrive. First we constrain the event *new_message* from occurring too frequently according to the pattern instance in Fig. 1. Let $\mathcal{L}_{nm}$ represent the set of admissible traces for this pattern. This set is defined explicitly in Formula 1.

$$\begin{aligned}\mathcal{L}_{nm} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models \mathit{new\_message} \Rightarrow\\ \neg[\exists j : (j > i) \wedge (\tau_j < \tau_i + 50) \wedge (\sigma_j \models \mathit{new\_message})]\}\end{aligned}$$

Suppose we wish to constrain the system to the pattern instances in Fig. 2. The first pattern instance is represented by the set $\mathcal{L}_{start}$ and the second by $\mathcal{L}_{stop}$:

$$\mathcal{L}_{start} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models \mathit{new\_message} \Rightarrow \sigma_i \models \mathit{thread\_start}\}$$

$$\begin{aligned}\mathcal{L}_{stop} = \{(\sigma, \tau) \mid \forall i \exists j : \sigma_i \models \mathit{thread\_start} \Rightarrow\\ (j > i) \wedge (\tau_i + l \leq \tau_j \leq \tau_i + h) \wedge (\sigma_j \models \mathit{thread\_stop})\}\end{aligned}$$

Let $\mathcal{L}_M$ denote the admissible traces of the transition system that is being modeled. The goal is to specify the transition system in Lustre such that $\mathcal{L}_M = \mathcal{L}_{nm} \cap \mathcal{L}_{start} \cap \mathcal{L}_{stop}$. Writing a Lustre constraint to represent the set of traces $\mathcal{L}_{start}$ is trivial. The traces that are contained in $\mathcal{L}_{start}$ are those whose states all satisfy the expression *new_message* = *thread_stop*. However, as we noted earlier, it is not possible to develop a set of Lustre constraints that admit only (and all of) the traces of $\mathcal{L}_{stop}$.

Note that the second pattern in Fig. 2 is an instance of the pattern described in Fig. 3. Therefore we can split the set $\mathcal{L}_{stop}$ into two sets, $\mathcal{L}_{stopc}$ and $\mathcal{L}_{stopp}$:

$$\begin{aligned}\mathcal{L}_{stopc} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models \mathit{thread\_start} \Rightarrow \exists j : (j > i) \wedge\\ [(\tau_i + l \leq \tau_j \leq \tau_i + h \wedge \sigma_j \models \mathit{thread\_stop}) \vee\\ (\tau_j \leq \tau_i + h \wedge \sigma_j \models \mathit{thread\_start})]\}\end{aligned}$$

$$\begin{aligned}\mathcal{L}_{stopp} = \{(\sigma, \tau) \mid \forall i : \sigma_i \models \mathit{thread\_start} \Rightarrow \forall j : (j > i) \wedge\\ (\tau_j \leq \tau_i + h) \wedge \sigma_j \models \mathit{thread\_start} \Rightarrow\\ \exists k \in (i, j] : \tau_i + l \leq \tau_k \wedge \sigma_k \models \mathit{thread\_stop}\}\end{aligned}$$

In this example, the sets of admissible traces representing the patterns happen to have the following relationship:

$$\mathcal{L}_{nm} \cap \mathcal{L}_{start} \subseteq \mathcal{L}_{stopp} \tag{4}$$

This is because for every trace in $\mathcal{L}_{nm}$ the event *new_message* only occurs at most every 50 ms. Likewise, for each state of every trace of $\mathcal{L}_{start}$ the variable *thread_start* is true if and only if *new_message* is true. Finally, the set $\mathcal{L}_{stopp}$ contains every trace where *thread_start* occurs at most every 20 ms. From Formula 4 and Theorem 2 we have $\mathcal{L}_{nm} \cap \mathcal{L}_{start} \cap \mathcal{L}_{stopc} = \mathcal{L}_{nm} \cap \mathcal{L}_{start} \cap \mathcal{L}_{stop}$. Thus the system $\mathcal{L}_{nm} \cap \mathcal{L}_{start} \cap \mathcal{L}_{stopc}$, which we can model in Lustre, is equivalent to a system constrained by the pattern instances in Figs. 1 and 2.

Example 1 is meant to demonstrate that, in practical systems, there is usually some constraint on how frequently events outside the system may occur. Systems described by the functional RSL patterns generally have some limitations on how many events they can respond to within a finite amount of time. The Lustre implementations of $\mathcal{L}_{cons}$ and $\mathcal{L}_{prop}$ are simpler than Fig. 4, and their proof of correctness is also simpler then Theorem 1, though we omit both due to space limitations.

4 Application

We implemented a number of RSL patterns into the AGREE tool. These patterns were used to reason about the behavior of a real-world avionics system. Specifically, the patterns were used to model the logic and scheduling constraints of threads running on a real-time operating system on an embedded computer on an air vehicle. Each thread in the system has a single entry point that is dispatched by some sort of event. The event may be the arrival of data from a bus or a signal from another thread. When a thread receives an event, the current state of the thread's inputs are latched. Each thread runs at the same priority as every other thread (no thread may preempt any other thread). A thread begins executing after it receives an event and no other thread is executing.

The patterns in Figs. 1 and 2 are actually fairly representative of the constraints used in this model. Figure 5 shows some of the RSL patterns that were used to describe these scheduling constraints. We added an additional tag "exclusively" before the second event in the patterns to indicate that the second event occurs only in the specified interval after the first pattern (and never any other time). We found that this was a useful shorthand because one often wants to specify a signal that only occurs under a specified condition and not at any other time.

assert *"thread A runtime"* **: whenever** *thread_A_start_running* **occurs**
thread_A_finish **exclusively occurs during** [10.0**,** 50.0]**;**

assert *"thread B runtime"* **: whenever** *thread_B_start_running* **occurs**
thread_B_finish **exclusively occurs during** [10.0**,** 50.0]**;**

assert *"thread C runtime"* **: whenever** *thread_C_start_running* **occurs**
thread_C_finish **exclusively occurs during** [10.0**,** 50.0]**;**

Fig. 5. Assertions about the how the operating system schedules threads

The results that each thread produces after it finishes executing are described by an assume-guarantee contract. Generally speaking, the assumptions restrict the values of inputs that the thread expects to see. Likewise, the thread's guarantees constrain the values of the thread's outputs based on it's current state and input values. The AADL component that contains the threads has assumptions about how frequently it receives inputs and has guarantees about how quickly it produces outputs. These assumptions are translated to constraints in the Lustre transition system, and the guarantees are translated to properties. Figure 6 illustrates one of these assumptions and guarantees.

The "eq" statements in Fig. 6 are used to constrain a variable to an expression. They are usually used as a convenient short hand to make AGREE contracts easier to read. In this case, the first "eq" statement is used to set the variable *change_status_request* to true if and only if a new message has arrived and the content of the message is requesting that the vehicle change its status. Likewise, the second statement is used to record the last requested change value into the *change_request* variable. The contract assumes that this new message arrives periodically (with some jitter). The contract guarantees that if a new message arrives requesting that the vehicle change its status, then the vehicle's status will be set to the requested value within 500 ms. In this application we assumed that all time units are expressed in microseconds. This means that the timing constraints expressed in Fig. 5 are also expressed in microseconds. Other constraints are used to assert that the *vehicle_status* variable corresponds to one of the state variables in the component's threads.

```
eq change_status_event : bool =
    new_message and message_content.change_vehicle_status;

eq change_request : bool =
    ite(change_status_event,
        message_content.status,
        false → pre(change_request));

assume "periodic messages" : new_message occurs
    each 10000.0 with jitter 50.0;

guarantee "new message can change vehicle status" :
    whenever change_status_event occurs
        vehicle_status = change_request during [0.0, 500.0];
```

Fig. 6. Assumptions and guarantees about the component containing the threads.

The guarantee of this component is invariant if and only if the threads in the component's implementation are scheduled in such a way that whenever a new message arrives its content is parsed and sent to the correct threads to be processed in a timely manner. The logic expressed in the contract of each thread determines how the content of this message is transmitted to other threads in the system.

4.1 Results

We had three properties of interest for the vehicle. These properties were related to timing, schedulability, and behavior of the system's threads. We ran the translated Lustre file, which contained about 1000 lines, from the AADL/AGREE model on the latest version of JKind on a Linux machine with an Intel(R) Xeon(R) E5-1650 CPU running at 3.50 GHz. JKind uses k-induction, property directed reachability, and invariant generation engines to prove properties of Lustre models. In the case of this experiment, it took about 8 h to prove all three properties. One of the properties was proved via k-induction, the other two were proved by the property directed reachability engine.

JKind allows users to export the lemmas used to prove a property. These lemmas can be exported and used again in order to speed up solving for similar models and properties. We found that when these lemmas were used again to prove the properties a second time all of the properties were proved in less than 10 s. This seems to indicate that the properties are not particularly *deep*. That is to say, to prove the properties via k-induction, the inductive step does not need to unroll over many steps. We are currently exploring techniques for lemma discovery for properties specified with RSL patterns.

5 Related Work

Our work focuses on the real-time patterns in the Requirements Specification Language (RSL) [5] that was created as part of the CESAR project [16]. This language was an extension and modularization of the Contract Specification Language (CSL) [22]. The goal of both of these projects was to provide contract-based reasoning for complex embedded systems. We chose this as our initial pattern language because of the similarity in the contract reasoning approach used by our AGREE tool suite [14].

There is considerable work on real-time specification patterns for different temporal logics. Konrad and Cheng [2] provide the first systematic study of real-time specification patterns, adapting and extending the patterns of Dwyer [1] for three different temporal logics: TCTL [7], MTL [8], and RTGIL [9]. Independently, Gruhn [3] constructed a real-time pattern language derived from Dwyer, presenting the patterns as observers in timed automata. In Konrad and Cheng, multiple (and overlapping) occurrences of patterns are defined in a trace, whereas in Gruhn, only the first occurrence of the pattern considered. This choice sidesteps the question of adequacy for overlapping triggering events (as discussed in Sect. 3), but limits the expressiveness of the specification. We use a weaker specification language than Konrad [2] which allows better scaling to our analysis, but we also consider multiple occurrences of patterns, unlike Gruhn [3]. Bellini [4] creates a classification scheme for both Gruhn's and Konrad's patterns and provides a rich temporal language called TILCO-X that allows more straightforward expression of many of the real-time patterns. Like [2], this work considers multiple overlapping occurrences of trigger events.

The closest work to ours is probably that of Abid et al. [10], who encode a subset of the CSL patterns as observers in a timed extension of Petri nets called TTS, and supplement the observers with properties that involve both safety and liveness in LTL. For most of the RSL patterns considered, the patterns are only required to hold for the first triggering event, rather than globally across the input trace. In addition, the use of full LTL makes the analysis more difficult with inductive model checkers. Other recent work [6] considers very expressive real-time contracts with quantification for systems of systems. This quantification makes the language expressive, but difficult to analyze.

Other researchers including Pike [23] and Sorea [24] have explored the idea of restricting traces to disallow overlapping events in order to reason about real-time systems using safety properties. The authors of [25] independently developed a similar technique of using a *trigger* variable to specify real-time properties that quantify over events.

6 Conclusion

We have presented a method for translating RSL patterns into Lustre observers. While we only specifically discussed a single pattern in detail, the techniques we presented can be applied analogously to other functional or timing patterns. Similarly, the techniques we presented can be applied to other synchronous data flow languages. The RSL patterns have been incorporated into the AGREE plugin for the OSATE AADL integrated development environment. We used these patterns to show that we could successfully model, and prove properties about, scheduling constraints for a real-world avionics application. Future work will focus on lemma generation to improve scalability for reasoning about real-time properties.

References

1. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, pp. 411–420. IEEE (1999)
2. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, pp. 372–381. ACM (2005)
3. Gruhn, V., Laue, R.: Patterns for timed property specifications. Electron. Notes Theoret. Comput. Sci. **153**, 117–133 (2006)
4. Bellini, P., Nesi, P., Rogai, D.: Expressing and organizing real-time specification patterns via temporal logics. J. Syst. Softw. **82**, 183–196 (2009)
5. Reinkemeier, P., Stierand, I., Rehkop, P., Henkler, S.: A pattern-based requirement specification language: mapping automotive specific timing requirements. In: Fachtagung des GI-Fachbereichs Softwaretechnik, pp. 99–108 (2011)
6. Etzien, C., Gezgin, T., Froschle, S., Henkler, S., Rettberg, A.: Contracts for evolving systems. In: ISORC, pp. 1–8 (2013)
7. Alur, R.: Techniques for automatic verification of real-time systems. Ph.D. thesis, stanford university (1991)

8. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-time Syst. **2**, 255–299 (1990)
9. Moser, L.E., Ramakrishna, Y., Kutty, G., Melliar-Smith, P.M., Dillon, L.K.: A graphical environment for the design of concurrent real-time systems. ACM Trans. Softw. Eng. Methodol. (TOSEM) **6**, 31–79 (1997)
10. Abid, N., Dal Zilio, S., Le Botlan, D.: Real-time specification patterns and tools. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 1–15. Springer, Heidelberg (2012)
11. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. Proc. IEEE **79**, 1305–1320 (1991)
12. Backes, J., Cofer, D., Miller, S., Whalen, M.W.: Requirements analysis of a quad-redundant flight control system. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 82–96. Springer, Heidelberg (2015)
13. Murugesan, A., Heimdahl, M.P., Whalen, M.W., Rayadurgam, S., Komp, J., Duan, L., Kim, B.G., Sokolsky, O., Lee, I.: From requirements to code: model based development of a medical cyber physical system. SEHC (2014)
14. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 126–140. Springer, Heidelberg (2012)
15. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, 1st edn. Addison-Wesley Professional, Reading (2012)
16. CESAR: The CESAR project (2010). http://www.cesarproject.eu/
17. Dutertre, B., Sorea, M.: Timed systems in SAL. Technical report, SRI International (2004)
18. Pike, L.: Real-time system verification by *k*-induction. Technical report, NASA (2005)
19. Gao, J., Whalen, M., Van Wyk, E.: Extending lustre with timeout automata. In: SLA++P (2007)
20. Gacek, A., Backes, J., Whalen, M.W., Cofer, D.: AGREE Users Guide (2014). http://github.com/smaccm/smaccm
21. Gómez, R., Bowman, H.: Efficient detection of zeno runs in timed automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 195–210. Springer, Heidelberg (2007)
22. Gafni, V., Benveniste, A., Caillaud, B., Graf, S., Josko, B.: Contract specification language (CSL). Technical report, SPEEDS Deliverable D.2.5.4 (2008)
23. Pike, L.: Modeling time-triggered protocols and verifying their real-time schedules. In: Formal Methods in Computer-Aided Design, pp. 231–238 (2007)
24. Sorea, M., Dutertre, B., Steiner, W.: Modeling and verification of time-triggered communication protocols. In: 2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pp. 422–428. IEEE (2008)
25. Li, W., Grard, L., Shankar, N.: Design and verification of multi-rate distributed systems. In: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 20–29 (2015)

http://www.springer.com/978-3-319-40647-3

NASA Formal Methods
8th International Symposium, NFM 2016, Minneapolis,
MN, USA, June 7-9, 2016, Proceedings
Rayadurgam, S.; Tkachuk, O. (Eds.)
2016, XIX, 396 p. 129 illus., Softcover
ISBN: 978-3-319-40647-3