8-2004

Generating Properties for Runtime Monitoring from Software Specification Patterns

Oscar Mondragon

Ann Q. Gates
The University of Texas at El Paso, agates@utep.edu

Oleg Sokolsky

Comments:
UTEP-CS-04-21.

Generating Properties for Runtime Monitoring from Software Specification Patterns

Oscar Mondragon and Ann Q. Gates
Department of Computer Science
The University of Texas at El Paso
oscar, agates@cs.utep.edu

Oleg Sokolsky
Department of Computer and Information Sciences
University of Pennsylvania
sokolsky@cis.upenn.edu

Abstract

The paper presents an approach to support run-time verification of software systems that combines two existing tools, Prospec and Java-MaC, into a single framework. Prospec can be used to clarify natural language specifications for sequential, concurrent, and nondeterministic behavior. In addition, the tool assists the user in reading, writing, and understanding formal specifications through the use of property patterns and visual abstractions. Currently, Prospec automatically generates a specification written in Future Interval Logic (FIL). The goal is to automate the generation of MEDL formulas that can be used by the Java-MaC tool to check run-time compliance of system execution to properties. The paper describes the mapping that translates FIL formulas into MEDL formulas and demonstrates its correctness.

1. Introduction

Verification of system properties at runtime provides an extra layer of assurance for software systems. Even though properties can be verified with formal techniques during the requirements phase, errors can be introduced at design, implementation, or maintenance phases of the software lifecycle, or by the environment in which the system runs. Runtime monitoring is one approach for detecting violations to properties during program execution. A major challenge in this approach and other formal techniques, however, is specifying properties. Formally specifying the behavior of a software system is a difficult task because it requires a high level of mathematical sophistication and training to accurately specify, read, and understand properties written in a formal language. Furthermore, it is difficult to specify complete and consistent requirements.

A tool called Property Specification (Prospec) [1, 2], which is built on the Specification Pattern System (SPS) [3] and composite propositions, provides visual and textual guidance for specifying properties of systems. Prospec steps the practitioner through elicitation and specification of properties and generates formal specifications in Future Interval Logic (FIL) and Linear Temporal Logic (LTL) that can be used by theorem provers and model checkers.

The motivation for the work reported in this paper is to extend the use of properties elicited and specified through Prospec to runtime monitors, in particular Java Monitoring and Checking (Java-MaC) system [4]. Java-Mac uses alarms written in a formal language named Meta Event Definition Language (MEDL) to determine whether a property is violated by a trace of computation. This paper presents a mapping that transforms FIL formulas into MEDL alarms.

Section 2 of the paper provides a background of SPS, Prospec, Java-MaC, MEDL, and FIL. Section 3 gives an overview of the mapping, describing the extent to which the mapping can be applied. Section 4 presents an high-level description of the translation from FIL to MEDL and then describes the rewriting rules. Section 5 demonstrates correctness of the mapping. It shows that the generated MEDL formulas are well-formed formulas that assert the violation of the FIL formula being translated. It also shows termination of the translation algorithm and describes the testing of the MEDL formulas generated by the mapping. The paper ends with a summary and related work.

2. Background

This section provides a brief description of the tools used in this work: Prospec and Java-MaC. In addition, it describes SPS, the underlying framework of Prospec, as well as the languages FIL and MEDL.

2.1. Specification Pattern System

The Specification Pattern System [3] provides patterns and scopes to assist the practitioner in formally specifying software properties. Specification patterns are high-level abstractions that provide descriptions of common properties. The main patterns defined by SPS are: universality, absence, existence, precedence, and response. Universality properties are true in every point of the execution; absence properties are never true during the execution; existence properties are true at some point in the execution; precedence properties require that a given state or event always occurs before a designated state or

event occurs; and response properties require that the occurrence of a given state or event be followed by a designated state or event. Response properties represent a temporal relation called cause-effect between two propositions.

In SPS, each pattern is associated with a scope that defines the extent of program execution over which a property pattern is considered. There are five types of scopes defined in SPS: global, before L, after L, between L and R, and after L until R. *Global* denotes the entire program execution; *before R* denotes the execution before the first R occurs; *after L* denotes execution after the first L occurs; *between L and R* denotes the execution between intervals defined by L and R; and *after L until R* denotes the execution between intervals defined by L and R and, in the case when R does not occur, until the end of execution.

The SPS website provides descriptions of the patterns, including intent, relationships, and known uses. After the user selects a pattern and a specification language, e.g., LTL or Graphical Interval Logic (GIL), the website displays a mapping for each scope in the chosen language.

2.2. Prospec

The Property Specification tool (Prospec) [1] assists users in the elicitation and specification of properties by providing guidance, definitions, and graphics for SPS patterns and scopes. Prospec generates a formal specification in FIL and LTL (translations to LTL are in progress). The tool extends the functionality of SPS by including composite propositions (CP). CP are classes of relations among multiple propositions that define the structure of sequential and concurrent behavior. CP defined as conditions are used to describe concurrency, and those defined as events are used to describe activation or synchronization of processes or actions. An informal description of CP classes follows, where subscript C denotes a condition and E denotes an event, G_S denotes a set of propositions and G_Q denotes a sequence of propositions: $\text{AtLeastOne}_C(G_S)$- at least one of the propositions in G_S holds; $\text{AtLeastOne}_E(G_S)$ - at least one of the propositions in G_S becomes true; $\text{Parallel}_C(G_S)$ - all propositions in G_S hold; $\text{Parallel}_E(G_S)$ - all propositions in G_S become true. $\text{Consecutive}_C(G_Q)$ - each proposition in G_Q is asserted to hold in a specified order, one at each successive state; $\text{Consecutive}_E(G_Q)$ - each proposition in G_Q becomes true in a specified order, one at each successive state, and once they become true, their true value does not matter; $\text{Eventual}_C(G_Q)$ - each proposition in G_Q is asserted to hold in a specified order and in distinct and possibly nonconsecutive states; $\text{Eventual}_E(G_Q)$ - each proposition in G_Q becomes true in a specified order and in distinct and possibly nonconsecutive states. Refer to [1] for the semantics of CP classes.

CP can be used to define boundaries of scopes and patterns with multiple propositions. For instance, an ordered sequence can define the left boundary of an *after L* scope, and multiple events can define the cause part of a *response* pattern.

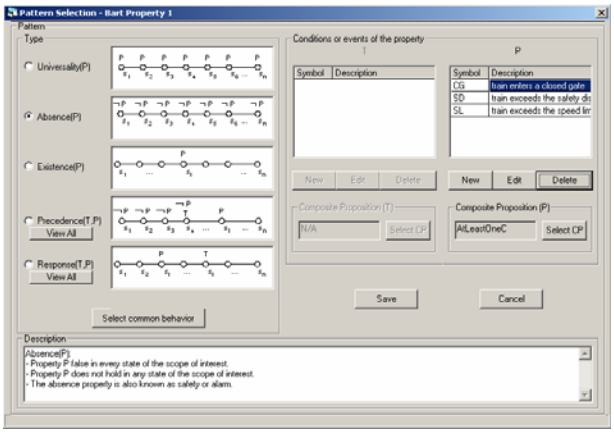

Figure 1. Prospec: pattern and proposition relations

For example, consider a property *P*: A train should not enter a closed gate, not exceed the safety distance limit of the train in front, and not exceed the speed limit of the track over which it is passing. The following propositions are identified: *CG*- train enters a closed gate; *SD*- train exceeds the safety distance limit; and *SL*- train exceeds the speed limit. Property *P* can be specified as an *absence* pattern (see Fig. 1) within *global* scope. What is needed next is to identify the relation among the propositions in the absence pattern. By following the guidance provided in Prospec, the responses lead to class $AtLeastOne_C$. The FIL specification generated by Prospec is (see Fig. 2):

$$\neg(\Diamond (CG \vee SD \vee SL)).$$

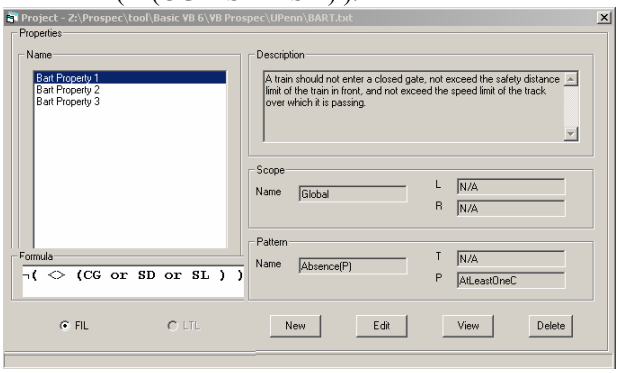

Figure 2. Property list and property detail description

A formal experiment conducted across three institutions evaluated the effects that Prospec and SPS have over the quality of the generated software property specifications. The results supported the hypothesis that users who specify software properties using Prospec correctly identify, on the average, more patterns and scopes than users who specify software properties using the SPS web site [5].

2.3. Java-MaC

Java-MaC is a tool that uses formally specified properties to monitor Java programs at runtime. Fig. 1 shows the overall architecture of Java-MaC. The architecture includes two main phases: a *static phase* (before a target program runs) and a *run-time phase* (while the target program executes). During the static phase, the run-time components (*filter*, an *event recognizer*, and a *run-time checker*) are automatically generated from a formal requirements specification. During the run-time phase, information about the execution of the target program is collected and checked against the given formal requirements specifications.

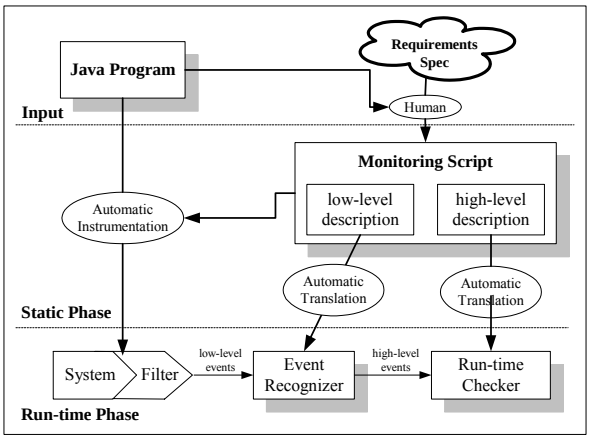

Figure 1. Java-MaC architecture.

The static phase of the MaC architecture starts with a formal requirements specification that is written in both high-level and low-level specifications. High-level specifications consist of required properties. Low-level specifications contain the definitions of primitive events and conditions used by these specifications. These definitions are given in terms of program entities such as program variables and program methods, and their purpose is to assign meanings to the program entities.

Once the specifications are written, the next task is to generate run-time components. Low-level specifications generate a filter that is inserted into the target program through the automatic instrumentation procedure. Also, they automatically generate an event recognizer. Similarly, a high-level specification generates the run-time checker.

During the run-time phase, the instrumented target program is executed while being monitored and checked with respect to a requirements specification. A *filter* is a collection of probes inserted into the target program. The essential functionality of a filter is to keep track of changes of monitored objects and send pertinent state information to the event recognizer. It is called a filter because it "filters" relevant information about the trace, and sends it to the checking routines. An *event recognizer* detects an event from the state information received from the filter. An event can be either a primitive event (such as a method call) or a change in the state of a condition. Events are recognized according to a low-level specification. Recognized events are sent to the run-time checker. A *run-time checker* determines whether or not the current execution history satisfies a requirements specification and raises an alarm if a violation is detected. The execution history is captured from a sequence of events sent by the event recognizer.

2.4. Meta Event Definition Language

The Meta Event Definition Language (MEDL) is the language used by Java-MaC. MEDL uses events and conditions to express safety properties. Intuitively, a condition is a state predicate and an event is an instantaneous state change. MEDL is based on a two–sorted logic of conditions and events. Conditions are associated with propositions that are evaluated at each state of the computation. Events denote a change of state in a condition from one value to another. Conditions and events are defined recursively as follows.

- Every proposition is a primitive condition.
- If C_1 and C_2 are conditions, then $!C_1$, C_1 && C_2, $C_1 \parallel C_2$, and $C_1 \Rightarrow C_2$ are conditions.
- If E_1 and E_2 are events, then $[E_1, E_2)$ is a condition.
- If C is a condition, then **start**(C) and **end**(C) are events.
- If E_1 and E_2 are events, then $E_1 \parallel E_2$ and E_1 && E_2 are events.
- If E is an event and C is a condition, then E **when** C is an event.

MEDL uses alarms to express a violation of a property. An alarm is an event that should not occur during an execution. If an event that is designated as an alarm occurs during an execution, a user notification is issued.

MEDL formulas are evaluated over an execution trace. Each state in an execution trace assigns values to each primitive condition. Boolean operations on conditions are interpreted classically. This paper uses the two-valued semantics of MEDL.

Event **start**(C) occurs in state s_i if condition C is *false* in s_{i-1} and is *true* in s_i, and conversely for **end**(C). Event E **when** C occurs in state s_i if E occurs at s_i and C is *true* at the same state. Conjunction and disjunction of events is interpreted classically over Boolean expressions. Note that negation of an event is not allowed in the language. Finally, $[E_1, E_2)$ holds in a state s_i if there is an occurrence of event E_1 in some past state s' and there is no occurrence of event E_2 in any state between s' and s_i. The complete description of MEDL and its semantics can be found in [4].

2.5. Future Interval Logic

FIL interval formulas [6] assert properties within intervals of interest. The interval is defined using *search*

patterns α and β, also known as the left and right search patterns, respectively. A search pattern includes one or more searches. A search to a formula *g*, depicted as →*g*, identifies the first state in the computation at which *g* holds. A search to formula *g* fails if *g* does not hold at any state in the computation. Intervals are left-close and right-open so that they include the state found by the left search pattern and all consecutive states up to, but not including, the state found by the right search pattern. There are two special cases of intervals. A *prefix interval,* denoted [– | *β*), begins at the start of its parent interval. A *suffix interval,* denoted [*α* | →), terminates at the end of its parent interval. In both cases, if no parent interval is specified, the entire computation is used. A search to the end of an interval, denoted →, always succeeds.

Interval formula **[**α | β**)***p* holds if an interval is built and formula *p* holds at the first state of the interval, or if the interval cannot be built. An interval is built if: the left search pattern α succeeds, the right search β succeeds, and the state found by α precedes the state found by β. Ramakrishna [6] presents the syntax and semantics of FIL.

3. Overview of Translation

3.1. Goal

The goal of the work is to automate the generation of MEDL alarms that can be used by Java-MaC to determine whether a given trace of computation violates specified properties. This is needed because writing MEDL alarms for response and precedence properties that occur within an interval is a complex and error-prone task. The Prospec tool was developed to facilitate specification of such properties in FIL. An approach to accomplish our goal is to define a mapping that takes an FIL formula and returns an MEDL alarm.

Given a safety property and a trace, Java-MaC does not ask the question whether the trace satisfies the property. Rather, Java-MaC uses a satisfaction relation that checks whether the prefix of the trace ending at the given state violates the property. As a result, the mapping must change the satisfaction relation of the original FIL formula to express a violation of the formula.

3.2. Basics

Consider the following basic FIL interval formula:

$$[\rightarrow l \mid \rightarrow r)\, p \qquad (3.1)$$

where *l*, *r*, and *p* are propositions and →*l* and →*r* are search patterns. In the remainder of the paper, interval [→*l* | →*r*) is referred to as *interval lr*. FIL formula 3.1 denotes that interval *lr* is built and that *p* holds at the first state of the interval. Because MEDL asserts violations to properties, the MEDL formula must assert that the interval is built and that *p* does not hold at the beginning of the interval. Formula 3.2 represents a violation of Formula 3.1 as an MEDL alarm.

end([**start**(*l*) **when** !p, **start**(*r*)) (3.2)

A negated interval formula has the form ¬[→ *l* | → *r*)*p*. To handle more complex interval formulas such as nested intervals and negated interval formulas, Formula 3.2 is rewritten as the condition given in Formula 3.3.

[**end**([**start**(*l*) **when** !*p*, **start**(*r*))), **start**(*l*)). (3.3)

The second **start**(*l*) in Formula 3.3 defines the end of the condition and permits assertion of repeated intervals in a trace of computation. Formula 3.3 is next converted into an alarm by rewriting it as an event, i.e., asserting the **start** of the condition as follows:

start ([**end**([**start**(*l*) **when** !*p*, **start**(*r*))), **start**(*l*)). (3.4)

The alarm is raised when the start of the condition is asserted, i.e., when the event underlined in 3.4 occurs. Note that this event is the same as Formula 3.2.

Formula 3.4 follows the general structure for MEDL alarms generated by the mapping as given in Formula 3.5.

start ([**end**([e_1 **when** !*p*, e_2)), e_3)) (3.5)

Formula 3.5 asserts the start of the condition generated by events **end**([e_1 **when** !*p*, e_2) and e_3, where the former defines the end of interval and asserts that *p* does not hold at the end of this interval. Events e_1 and e_2 denote the start and end of the interval. In Formula 3.4, **start**(*l*) is event e_1, **start**(*r*) is event e_2, and the second **start**(*l*) is event e_3.

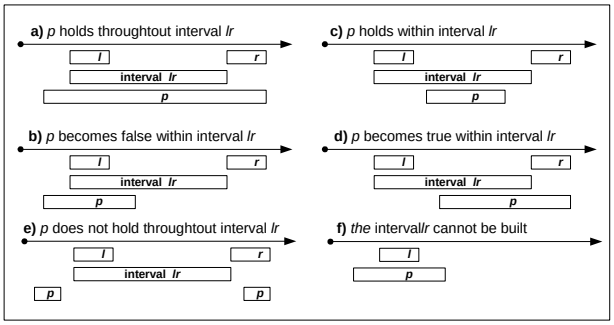

Figure 4. Traces of computation.

Fig. 4 depicts different traces of computation. In the figure, arrows denote traces of computation and rectangles denote subintervals (consecutive sequence of states) over which conditions hold. In traces *a* and *b* of Fig. 4, interval *lr* is built and *p* holds at the beginning of the interval; in traces *c, d,* and *e*, interval *lr* is built, but *p* does not hold at the beginning of the interval; and in trace *f* interval *lr* is not built.

Consider the FIL property given in Formula 3.1 and the MEDL alarm given in Formula 3.2. For traces *a* and *b* of Fig. 4, the FIL property is satisfied and the MEDL alarm is not raised, i.e., event **start**(*l*) **when** !*p* does not occur. For traces *c, d,* and *e*, the FIL property is not satisfied because *p* does not hold at the beginning of the interval, and the MEDL alarm is raised signaling that the property has been violated. For trace *f*, the FIL property holds by definition when the interval cannot be

constructed. Similarly, the MEDL alarm is not raised because the end of the interval cannot be asserted.

3.3. Scope of the Mapping

FIL can express safety and liveness formulas; however, runtime monitors can only verify safety properties. The mapping centers on a subset of safety formulas that can be monitored by Java-MaC. The mapping does not support formulas where the henceforth and eventually operators are used together, i.e., persistence and recurrence formulas denoted ◊□p and □◊p, respectively. Normally, liveness formula ◊*p* cannot be monitored at runtime. The mapping, however, can handle safety formulas that include the eventual operator. If an eventual formula is bounded within a prefix of the computation, then the formula can be monitored. For instance, consider asserting that proposition *p* eventually holds within an interval in which the left and right boundaries are defined by propositions *l* and *r,* respectively. If the monitor asserts *l* and some time in the future asserts *r* (i.e., the interval is built), the monitor can determine whether ◊*p* holds within the interval.

Table 1. Properties verifiable by Java-MaC.

Pattern Name	Scope				
	Global	Before R	After L	Between L and R	After L Until R
Universality	√	√	√	√	√
Absence	√	√	√	√	√
Existence	X	√	X	√	X
Precede	√	√	√	√	√
Response	X	√	X	√	X

The general response formula in FIL is □ ([→ p | →) ◊s), i.e., if proposition *p* holds, then some time in the future proposition *s* holds. The mapping applies only to response formulas that are bounded within a prefix of the computation (safety formulas).

The FIL formulas that are translated to MEDL formulas are those generated by the patterns and scopes given in Table 1 and marked with a check mark (√). This set of FIL formulas is called $L_{\text{FIL-SPS}}$.

4. Translation from FIL to MEDL

4.1. Algorithm Map

Algorithm *Map* given in Fig. 5 translates an FIL formula $f \in L_{\text{FIL-SPS}}$, to a MEDL formula that asserts a violation if *f* does not hold. Refer to Table A-1 in the appendix for the mapping rules used in this translation. The translation applies the rules in a goal-directed fashion and is centered on applying mapping rule 8 to the basic interval formula.

```
INPUT: formula f ∈ L_FIL-SPS
OUTPUT: MEDL alarm

1. if f is a response formula then apply Rule 30 else {
2.     if f has a henceforth operator then apply Rule 28.
3.     if f has an eventually operator then apply Rule 29. }
4. if f has a prefix interval then apply Rule 4.
5. for each prefix subinterval in f apply Rule  5.
6. if f has a suffix interval then apply Rule 6
7. for each suffix subinterval in f apply Rule 7.
8. if f is a negated interval formula then apply Rule 2.
9. Apply Rule 8.
10 while f contains a μ function {
11.     if the formula has nested μ functions then
              select the innermost μ function
12.     else Select any μ function
13.          apply the matching transformation rule. }
14. Alarm ← start( + <transformed formula> + )
```

Figure 5. Algorithm *Map* for transforming FIL to MEDL.

4.2. Basic Rules

The algorithm first determines if the formula is a response formula. In this case, it translates the formula by applying rule 30, yielding $[\rightarrow p \mid \rightarrow s^r)$*false*. Proposition s^r is translated via rule 19d. Next, the algorithm translates derived operators *henceforth* and *eventually*, if present, via rules 28 and 29. Formula $[\rightarrow l \mid \rightarrow r)\ \Box p$ asserts that *p* is *true* within the interval, and rules 28 and 7 rewrite this formula. Formula $[\rightarrow l \mid \rightarrow r)\ \Diamond p$ asserts that p occurs within the interval, and rules 24 and 5 rewrite this formula.

Mapping rules 1, 2, and 3 define translations for formulas that use propositional logic. Rules 4 and 6 apply to prefix and suffix intervals, respectively. A prefix interval in FIL denotes that the interval starts at the beginning of the computation, which translates to **start(true)** for e_1 in Formula 3.5. A suffix interval in FIL denotes that the interval finishes at the end of the computation, which translates to **start(false)** for e_2 in Formula 3.5. Rule 5 applies to prefix subintervals (e.g., $[\rightarrow l_1 \mid \rightarrow r_1)\ [-\mid \rightarrow r_2)p$), where the left search of the subinterval is assigned the left search of its parent interval. Rule 7 is for suffix subintervals (e.g., $[\rightarrow l_1 \mid \rightarrow r_1)\ [\rightarrow l_2 \mid \rightarrow)p$), where the right search of the subinterval is assigned the right search of its parent interval.

Rule 8 transforms interval formula into a rule that enforces the structure of Formula 3.5. The following part of rule 8: μ(left(μ(lInterval([α | β)*p*)))) is transformed into e_1 **when** !*p* by applying rules 9, 14, 15, 2, 24, and 1. Note that e_1 **when** !*p* asserts the negation of *p* either at the beginning of the interval or at the beginning of the intersection of all nested intervals (see Section 4.3). The following part of rule 8: μ(right(μ(rInterval([α | β)*p*)))) is transformed into e_2 by applying rules 11, 14, 18, 21 and 24. Event e_2 asserts the end of the interval and that all subintervals end before its parent interval. Event e_3 is **start**(μ(last(α))) in rule 8.

4.3. Nested Intervals

The mapping considers nested-interval formulas that can be derived from a basic interval formula. For example, consider the following FIL interval formula of depth 2, where l, r, and p are propositions: $[\rightarrow l_1 \mid \rightarrow r_1)$ $[\rightarrow l_2 \mid \rightarrow r_2)p$. This formula is read as: nested interval l_2r_2 must hold within parent interval l_1r_1 and property p must hold at the beginning of interval l_2r_2. This formula can be depicted as a basic interval formula: $[\rightarrow l_1 \mid \rightarrow r_1)q$, where $q = [\rightarrow l_2 \mid \rightarrow r_2)p$. Using this approach, the mapping first creates a list of intervals (rules 9-14) and then intersects the nested intervals (rules 15-20) to determine the point at which the property should be asserted. The steps for creating the intersection follow.

The first step is to apply rules 9-12 to create a list of intervals by using search patterns. Rules 9 and 10 create a list of search patterns that will be used to determine the beginning of the nested intervals. The last element of the list is the property to be asserted and the other elements are the search patterns that define the inner intervals. The first pair of search patterns in the list represents the parent interval. Rules 11 and 12 are used to create a list of the search patterns that will be used to determine the end of the nested intervals, where the first pair in the list corresponds to the parent interval.

Next the translation applies rules 13 and 14. Rule 13 creates an event that occurs when the beginning of the intersection of the nested interval is identified. Rule 14 creates an event that occurs when the end of the parent interval is identified and the nested intervals are built.

Rules 15-17 build the beginning of the nested interval. Rule 15 creates a condition using a pair of search patterns. This rule is used when the FIL formula has no nested intervals. Rule 16 creates a condition that asserts the start of the intersection between the parent interval and the nested interval. Rule 17 considers the case where a FIL formula has more than two nested intervals.

Rules 18-20 build the end of the nested intervals. Rule 18 generates constant *true* when the FIL formula has no nested intervals. Rule 19 creates a condition that asserts the end of the nested interval within its parent interval. Rules 19c and 19d are special cases for eventual and response formulas, respectively. Rule 20 considers the case where a FIL formula has more than two nested intervals.

4.4. Search Patterns

A search pattern is a sequence of searches $s_1, s_2, \ldots, s_n$. Search s_1 starts at the beginning of the parent interval. Search s_{i+1} starts at the state found by search s_i. Recall that a search to a formula f finds the first state in the computation where f holds.

For search pattern $\rightarrow g$, $\rightarrow h$, a search to formula h starts only if a search to formula g succeeds. The above search pattern succeeds at the same state if g and h hold at the same state. Nested intervals and search patterns share some properties, i.e., a nested interval may start at the same state as its parent interval. The approach used to convert FIL formulas with nested intervals is adopted for search patterns. Rule 21 considers search patterns with one search. Rule 22 considers search patterns with two searches, and rule 24 considers search patterns with more than two searches. In rule 24, *head* returns the formula of the first search within a search pattern. In rule 25, *last* returns the formula of the last search in a search pattern.

5. Verification of the Map Algorithm

The proofs are based on the formal semantics of MEDL [4] and FIL [6]. The FIL semantics uses the models relation $\models$ for a model M and a pair of indices i, j, where i and j must be between 0 and |M| inclusive, or $\perp$. Function *locate* returns the index of the first state at which a proposition holds within an interval and returns $\perp$ if i or j is $\perp$ or if the proposition does not hold. The MEDL semantics uses the models relation $\models$ for a model M and an index j that denotes the time at which the state s_j occurs (recall that MEDL uses a prefix of the computation). Because of space limitations, this section provides outlines of the proofs.

5.1. Proofs

Lemma 1. Interval formula $[\rightarrow l \mid \rightarrow r)$ p holds within a trace of computation when either of the following properties hold:

P1. An interval is built and formula p holds at the first state of interval lr, where an interval is built if:
  P2. The left search pattern $\rightarrow l$ succeeds.
  P3. The right search pattern $\rightarrow r$ succeeds.
  P4. The state found by search l precedes the state found by search r.
P5. Interval lr cannot be built.

Outline of proof. Lemma 1 is true by definition of FIL semantics. Specifically, if $i \neq \perp$ and $j \neq \perp$ or $i < j$, then $(M,i,j) \models [\rightarrow l \mid \rightarrow r)p$ iff $(M, \text{locate}(l,M,i,j), \text{locate}(r,M,i,j)) \models p$, which states that: if l can be asserted at k_1 and r can be asserted at k_2 such that $i \leq k_1 < k_2 \leq j$, then $(M,k_1,k_2) \models p$. If $i = \perp$ or $j = \perp$ or $j \leq i$, then $(M,i,j) \models [\rightarrow l \mid \rightarrow r)\, p$. ■

Lemma 2. MEDL event **end([start**(l) **when** p, **start**(r))**)** occurs at s_i if the condition **([start**(l) **when** p, **start**(r))**)** changes from *true* to *false* at s_i, i.e.,

Q1. proposition r changes from *false* to *true* at s_i,
Q2. there exists a state s_j, where $0 \leq j < i$, such that proposition l is *false* at s_{j-1} and *true* at s_j and p is *true* at s_j,
Q3. **start**(r) is *false* for all s_k, where $j \leq k < i$.

Outline of proof. Lemma 2 is true by definition of MEDL semantics. Specifically, $(M,j) \models$ **end([start**(l) **when** p, **start**(r))**)** iff $(M,j) \not\models$ **[start**(l) **when** p, **start**(r)**)** $\wedge$ $(M,j\text{-}1) \models$ **[start**(l) **when** p, **start**(r))**,** i.e., this event occurs when

condition (**[start**(l) **when** p, **start**(r)**)** changes from *true* to *false*. (M,j) $\models$ **[start**(l) **when** p, **start**(r)**)** iff $\exists\ i \le j \bullet$ (M,i) $\models$ **start**(l) **when** $p \wedge \forall k \in i..j$ (M,k) $\not\models$ **start**(r), i.e., **[start**(l) **when** p, **start**(r)**)** is *true* from **start**(l) **when** p until **start**(r). This condition represents interval lr (see Q3). (M,i) $\models$ **start**(l) **when** p iff (M,i) $\models$ **start**(l) $\wedge$ (M,i) $\models$ p, i.e., **start**(l) occurs when p is *true*. This ensures that p holds at the beginning of interval lr (see Q2). (M,i) $\models$**start**(l) iff (M,i) $\models$ **start**(l) $\wedge$ (M,i-1)$\not\models$ **start**(l), i.e., l changes from *false* to *true* at s_i. This identifies the first state at which l holds. Similarly, **start**(r) identifies the first state which r holds (see Q1). ■

Lemma 3. In traces of computation in which lr can be built, FIL interval formula $f = [\rightarrow l \mid \rightarrow r)\ p$ holds iff MEDL formula g= **end([start**(l) **when** p, **start**(r)**)**) occurs.

Outline of proof. If f holds then g occurs. By Lemma 1, property P1 holds. Because the left search pattern for l succeeds in FIL, **start**(l) occurs in MEDL. As a result, l is *true* at some state s_i and *false* at s_{i-1}. Because the right search pattern for r succeeds, **start**(r) occurs. This means that r is *true* at some state s_j and *false* at s_{j-1}. A search to formula r in FIL finds the first state at which r holds. Proposition r is *false* for all states s_k where $i \le k < j$. As a result, **start**(r) must be *false* for all s_k, where $i \le k < j$. Because search l precedes the state found by search r, event **start**(l) precedes event **start**(r) and i must be less than j. Because property p holds at the first state of the interval in f, then p must be *true* when **start**(l) occurs, i.e., at s_i. Because the properties of Lemma 2 hold, g occurs.

If g occurs, then f holds. By Lemma 2, properties Q1 to Q3 hold. Because proposition l becomes *true* at s_i, r becomes *true* at s_j and r is *false* for all s_k where $i \le k < j$, it follows that interval $[\rightarrow l \mid \rightarrow r)$ can be built and that the state found by search l precedes the state found by search r. Because p holds at s_i, then p holds at the first state of interval $[\rightarrow l \mid \rightarrow r)$. Because the properties of Lemma 1 hold, f holds. ■

Theorem 1. Given an FIL formula $f \in L_{FIL\text{-}SPS}$ and a trace of computation T, Map(f) holds in T iff f does not hold in T.

Outline of proof. The proof is by induction on the depth of nested FIL interval formulas.

Basis step: Interval formula of depth 1. Consider interval formula $f = [\rightarrow l \mid \rightarrow r)p$, where l, r, and p are propositions. By FIL semantics, f does not hold means that the interval is built and p does not hold at the beginning of the interval. By Lemma 3, if interval lr can built, f holds iff **end([start**(l) **when** p, **start**(r)**)**) occurs. It follows that the MEDL alarm that occurs when f does not hold is:

end([start(l) **when** ! p, **start**(r) **)**). (5.1)

Table A-2 in the appendix enumerates the steps that lead to the transformation in the more general case. Because of space limitations, the paper does not include the lemma that shows the equivalence of Formulas 5.1 and 3.4 (refer to Section 3.2 for a discussion of the construction of Formula 3.4).

Induction Hypothesis. Given an FIL formula f of depth n and a trace of computation T, Map(f) occurs in T iff f does not hold in T. Nested interval formula $f = [\rightarrow l_1 \mid \rightarrow r_1)\ [\rightarrow l_2 \mid \rightarrow r_2) \ldots [\rightarrow l_n \mid \rightarrow r_n)p$. Map($f$) yields MEDL wff:

[**end**([**start**([**start**([**start**(l_1), **start**(r_1) **) &&** l_2), **start**(r_1)) && …&& [**start**([**start**(l_{n-1}), **start**(r_{n-1}) **) &&** l_n), **start**(r_{n-1}))) **when** !p, **start**(r_1) **when** [**end**([**start**(l_2), **start**(r_2) **)) when** [**start**(l_1), **start**(r_1) **)**, **end**([**start**(r_1), **start**(l_1) **))))) &&…&&** [**end**([**start**(l_n), **start**(r_n) **)) when** [**start**(l_{n-1}), **start**(r_{n-1}) **)**, **end**([**start**(r_n), **start**(l_n) **))))), start**(l_1)

Induction Step. The proof outline below argues the if direction, i.e., given a trace of computation T and an FIL formula f of depth $n+1$ that does not hold in T, then Map(f) occurs in T. The proof for the only-if direction uses similar reasoning.

The construction of the MEDL alarm from formula f follows. The mapping transforms the intervals from left to right (parent interval to the deepest nested interval) as consecutive pairs of intervals, i.e., $[\rightarrow l_1 \mid \rightarrow r_1)$ and $[\rightarrow l_2 \mid \rightarrow r_2)$, $[\rightarrow l_2 \mid \rightarrow r_2)$ and $[\rightarrow l_3 \mid \rightarrow r_3)$,…,$[\rightarrow l_{n-1} \mid \rightarrow r_{n-1})$ and $[\rightarrow l_n \mid \rightarrow r_n)$. Adding interval $l_{n+1}r_{n+1}$ requires applying mapping rules 17 and 20 in Step 11 of the *Map* algorithm. Rule 17 adds the following condition to the MEDL formula:

[**start**([**start**(l_n), **start**(r_n) **) &&** l_{n+1}), **start**(r_n)).

This condition intersects the beginning of interval l_nr_n with $l_{n+1}r_{n+1}$. This condition is added to the formula by intersecting it with the condition that asserts the beginning of intervals $l_{n-1}r_{n-1}$ and l_nr_n. Rule 20 adds the following condition:

[**end**([**start**(l_{n+1}),**start**(r_{n+1}) **)) when start**(l_n),**start**(r_n)**)**, **end**([**start**(r_{n+1}), **start**(l_{n+1}) **)**])).

This condition asserts the end of interval $l_{n+1}\ r_{n+1}$ within interval l_nr_n, and it is added by intersecting it with a condition that handles intervals $l_{n-1}\ r_{n-1}$ and l_nr_n. Map(f) yields:

[**end**([**start**([**start**([**start**(l_1), **start**(r_1) **) &&** l_2), **start**(r_1)) && …&&

[**start**([**start**(l_{n-1}), **start**(r_{n-1}) **) &&** l_n), **start**(r_{n-1})) &&

[**start**([**start**(l_n), **start**(r_n) **)&&** l_{n+1}), **start**(r_n))) **when** !p, **start**(r_1) **when** [**end**([**start**(l_2), **start**(r_2) **)) when**

[**start**(l_1),**start**(r_1)), **end**([**start**(r_1), **start**(l_1) **)))))**

&&…&& [**end**([**start**(l_n), **start**(r_n) **)) when** [**start**(l_{n-1}), **start**(r_{n-1}) **)**, **end**([**start**(r_n), **start**(l_n) **))**

[**end**([**start**(l_{n+1}), **start**(r_{n+1}) **)) when** [**start**(l_n), **start**(r_n) **)**, **end**([**start**(r_{n+1}), **start**(l_{n+1}) **))))**), **start**(l_1)).

The underlined (solid and dashed) portions of the alarm are those added to the alarm given in the induction hypothesis. By the semantics of FIL, it is known that a formula f with nested intervals of depth $n+1$ does not hold when: 1) all intervals are built; 2) interval $n+1$ holds within interval n and interval n holds within interval

n-1 … and interval 2 holds within interval 1, and 3) *p* does not hold at the beginning of interval *n+1*. From the induction hypothesis, Map(*f*) yields a MEDL alarm that occurs in T for nested intervals of depth *n*. The condition distinguished by the solid underline is added to the alarm given in the induction hypothesis. This condition intersects the beginning of interval *n+1* with *n*, ensuring that that the beginning of interval *n*+1 follows the beginning of interval *n*. By MEDL semantics, !*p* is asserted at the start of the intersection of the beginning of all intervals. The condition distinguished by the dashed underline in the MEDL alarm is also added to the alarm from the induction hypothesis. This condition intersects the end of interval *n+1* with interval *n*, ensuring that the end of interval *n+1* precedes the end of interval *n*. It follows that Map(*f*) occurs in T at the end of the parent interval. ■

Theorem 2. Given an FIL formula $f \in L_{\text{FIL-SPS}}$, algorithm *Map(f)* terminates.

Outline of proof. The transformations provided by *Map* divide into four groups: 1) those that remove derived operators and special symbols from an FIL formula, 2) those that define the structure of the MEDL formula, 3) those that recursively refine the MEDL formula, and 4) those that provide supporting functions.

Transformations of type 1 are handled in Steps 1 - 8 of *Map* given in Fig. 5. Step 9 applies mapping rule 8, a transformation of type 2. Because these steps apply a sequence of non-recursive rules, rules 2, 4-8, and 28-30 are applied at most once; thus, *Map* makes progress.

Steps 10 -14 in *Map* make transformations of types 2-4. The claim is that these transformations lead to removal of all μ functions from the formula, resulting in termination of the *while* condition. Transformations of type 2 (rules 1, 4, 14-15 and 18) and transformations of type 4 (rules 24-27) are applied once. Rules 14 and 15 provide structure to the generated MEDL condition creating e_1 *when !p* and e_2, respectively.

Transformations of type 4 (rules 9-10, 11-12, 16-17, 19-20, and 21-24) are recursive transformations. Each call to a rule reduces the size of the formula, progressing toward the base case. Rules 9 and 10 and rules 11 and 12 generate a list of pairs of search patterns. These rules transform one interval at a time decreasing the list of intervals in each call. Rule 15 creates a condition and rules 16 and 17 intersect the beginning of two intervals, or multiple intervals, respectively, creating a condition for each pair of intervals taken from left to right. One interval is removed from the list in each recursive call. Rule 19 creates a condition that asserts the end of nested interval within parent interval. While rule 19 handles two intervals, rule 20 handles multiple intervals. Rule 20 creates a condition for each pair of intervals taken from left to right. One interval is removed from the list in each recursive call. Rules 21 to 24 define a recursive transformation for search patterns. Rule 21 and 22 handle the base cases. Rule 24 handles the case for multiple searches, where one search is removed from the list in each recursive call. Because each call to a recursive mapping rule decreases its parameter list one unit, the translation procedure is guaranteed to terminate. ■

5.2. Testing the Translation

We ran tests to check that the MEDL formulas generated from all basic FIL formulas $f \in L_{\text{FIL-SPS}}$ detected violations when *f* does not hold. All tests were successful. One such test evaluated an existence property pattern for formula *p* with a *Between L and R* scope. Let *p*, *l*, and *r* be propositions and *l* and *r* define interval boundaries *L* and *R*, respectively. The FIL formula for this property pattern is: $[\rightarrow l \mid \rightarrow r)\ \Diamond p$. The generated MEDL formula is:

[end([**start**([**start**(*l*), **start**(*r*) **)**),
start(*r*) **when !** [**start**(p && [**start**(*l*), **start**(*r*) **)**),
end([**start**(*r*), **start**(*l*) **)**)))), **start**(*l*))

The condition in the first line of the formula asserts the start of the scope. The second line asserts both the end of the scope and the negation of a condition, which asserts the intersection of proposition *p* within the scope. That is, if property p does not hold sometime within the scope, then a violation will be raised.

The traces given in Fig. 4 were used as test cases to check the existence property. Traces *a-d* satisfied the property, while traces *e-f* resulted in a violation. A description of the cases follows: a) condition *p* holds throughout the interval; b) condition *p* holds within the interval; c) condition *p* becomes *true* before the start of the interval and becomes *false* within the interval; d) condition *p* becomes *true* within the interval and becomes *false* after the end of the interval; e) condition *p* does not hold throughout the interval; f) the interval cannot be built.

6. Summary

We have presented an approach to improve correctness assurance of software systems. The approach combines two well-established tools in a single framework. On the one hand, the technology of SPS, implemented in the Prospec tool, is a proven way to specify subtle correctness properties of a system. On the other hand, run-time verification technology, implemented in the Java-MaC tool, has been shown to be useful in demonstrating at run time that the system satisfies its properties. In this paper, we demonstrate how to use pattern-based properties in run-time verification. The centerpiece of this work is the mapping that provides for the translation of pattern-based properties, expressed in FIL, into the monitoring language MEDL.

Related work. Havelund and Rosu [7, 8] have investigated the use of several commonly used logics in the run-time verification context. Their approach avoids the translation process and evaluates formulas directly on

an execution trace. However, a custom implementation of the evaluation algorithm is needed for each logic. We think that it is advantageous to use an existing tool that has been proven to be robust by a number of case studies.

Propel [9] is a tool that enhances SPS by identifying ambiguities in the intent of the patterns. It makes use of disciplined natural language and extended FSA.

Future work. Prospec is being modified to include MEDL. The next step is to extend the mapping to include CP to specify sequential and concurrent behavior for defining scope boundaries and patterns. Simplification of the translated MEDL formulas is another improvement that needs to be made. We believe the output of the translation contains redundancies that may slow down the run-time evaluation of the formulas. Eliminating the redundancies is not straightforward and requires further investigation into the equivalence of MEDL formulas.

References

[1] Mondragon, O. and A. Q. Gates, "Supporting Elicitation and Specification of Software Properties through Patterns and Composite Propositions," *Intl. Journal SEKE*,14(1):21-41, 2004.

[2] Mondragon, O., A. Q. Gates, and S. Roach, ``Prospec: Support for Elicitation and Formal Specification of Software Properties," *Electronic Notes Theoretical Computer Science*, O. Sokolsky and M. Viswanathan (eds.), 2004.

[3] Dwyer, M. B., G. S. Avrunin, and J. C. Corbett, "Property Specification Patterns for Finite-State Verification," 2nd Workshop on Formal Methods in Software Practice. Clearwater Beach, Florida, 1998, 7-15.

[4] Kim, M., S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, " Java-MaC: a Run-time Assurance Tool for Java Programs," in Havelund, K. and Rosu, G. (Eds.) Proc. the Runtime Verification Workshop, 55(2), Jul. 2001, 97-104.

[5] Mondragon, O., "Elucidation and Specification of Software Properties through Patterns and Composite Propositions to Support Formal Verification Techniques," Ph. D. Dissertation, Computer Science Department, University of Texas at El Paso, May 2004.

[6] Ramakrishna, Y. S., P.M. Melliar-Smith, L. E. Moser, L. K. Dillon, and G. Kuttty. "Interval Logics and their Decision Procedures. Part I: An Interval Logic," *Theoretical Computer Science,* 166(1-2), Oct. 1996, 1-47.

[7] Havelund, K. and G. Rosu , "Monitoring Programs using Rewriting," in Proc. Intl. Conf. ASE 01, Nov. 2001, 145-144.

[8] Havelund, K. and G. Rosu , "Synthesizing Monitors for Safety Properties," in *Lecture Notes in Computer Science*, 2280, Apr. 2002, 442-456.

[9] Smith, R., Avrunin, G., Clarke, L., and Osterweil, L., "PROPEL: An Approach Supporting Property Elucidation," *in Proc. ICSE*, Orlando, FL, USA, May 2002, 11-21.

Appendix: Mapping Rules and Transformations

The variable types used in the mapping rules in Table A-1 are as follows:

p, s: Proposition; f, g, k : IntervalFormula; Ω, Ψ : IntervalSequence;
α : LeftSearchPattern; β : RightSearchPattern; β^r: ResponseCase; β^e: EventuallyCase.

Table A-1. Mapping Rules for FIL to MEDL Translation.

1	$\mu(p)$	$\stackrel{def}{\equiv} p$ *// primitive condition*
2	$\mu(\neg f)$	$\stackrel{def}{\equiv} !\ \mu(f)$
3	$\mu(\ f \vee g\ )$	$\stackrel{def}{\equiv} (\mu(f) \parallel \mu(g))$
4	$\mu(\ [- \mid \beta)f\ )$	$\stackrel{def}{\equiv} \mu(\ [\rightarrow true \mid \beta)f\ )$ *// prefix interval*
5	$\mu(\Omega\ [\alpha_i \mid \beta_i)\ [- \mid \beta_{i+1})\Psi f)$, where $i \in \mathbf{N}_1$	$\stackrel{def}{\equiv} \mu(\ \Omega\ [\alpha_i \mid \beta_i)\ [\alpha_i \mid \beta_{i+1})\Psi f)$ *// prefix subinterval*
6	$\mu(\ [\alpha \mid \rightarrow)f\ )$	$\stackrel{def}{\equiv} [\ \mu(\text{left}(\mu(\text{lInterval}([\alpha \mid \rightarrow \text{false})f)))), \text{start}(\text{false})\ )$ *// suffix interval*
7	$\mu(\ \Omega[\alpha_i \mid \beta_i)[\ \alpha_{i+1} \mid \rightarrow)\Psi f)$, where $i \in \mathbf{N}_1$	$\stackrel{def}{\equiv} \mu(\Omega\ [\alpha_i \mid \beta_i)[\ \alpha_{i+1}\ \mid \beta_i)\Psi f)$ *// suffix subinterval*
8	$\mu(\ [\alpha \mid \beta)f\ )$	$\stackrel{def}{\equiv} [\ \text{end}([\ \mu(\text{left}(\mu(\text{lInterval}([\alpha \mid \beta)f)))),$ $\mu(\text{right}(\mu(\text{rInterval}([\alpha \mid \beta)f))))),\ \ \text{start}(\mu(\text{last}(\alpha)))\ )$
9	$\mu(\text{lInterval}([\alpha \mid \beta)p))$	$\equiv \alpha;\ \beta;\ p$
10	$\mu(\text{lInterval}([\alpha \mid \beta)f))$	$\equiv \alpha;\ \beta;\ \mu(\text{lInterval}(f))$
11	$\mu(\text{rInterval}([\alpha \mid \beta)p))$	$\equiv \alpha;\ \beta$
12	$\mu(\text{rInterval}([\alpha \mid \beta)f))$	$\equiv \alpha;\ \beta;\ \mu(\text{rInterval}(f))$
13a	$\mu(\text{left}(\alpha_1;\ \beta_1;\ldots,\ ;\alpha_n;\ \beta_n;\ \text{false}))$, where $n \geq 1$	$\equiv$ **start**$(\mu(\text{leftCond}(\alpha_1;\ \beta_1;\ldots,\ \alpha_n;\ \beta_n)))$
13b	$\mu(\text{left}(\alpha_1;\ \beta_1;\ldots,\ ;\alpha_n;\ \beta_n;\ p))$, $n \geq 1$	$\equiv$ **start**$(\mu(\text{leftCond}(\alpha_1;\ \beta_1;\ldots,\ \alpha_n;\ \beta_n)))$ **when** $!p$
14	$\mu(\text{right}((\alpha_1;\ \beta_1;\ldots;\alpha_n;\ \beta_n)),\ n \geq 1$	$\equiv$ **start**$(\mu(\beta))$ **when** $\mu(\text{rightCond}((\alpha_1;\ \beta_1;\ldots,\ \alpha_n;\ \beta_n))$

15	μ(leftCond(α; β))	≡ μ(search(α; head(β)))
16a	μ(leftCond((α$_1$; β$_1$;α$_2$; β$_2$))	≡ [**start**(μ(search(α; head(β$_1$))) **&&** μ(α$_2$)), **start**(μ(β$_1$)))
16b	μ(leftCond(**α**;β$_1$; **α**;β$_2$;…;**α**;β$_n$)), n≥2	≡ μ(search(α; head(β$_1$))) *// same left boundary*
17	μ(leftCond(α$_1$; β$_1$; α$_2$; β$_2$; …; α$_n$; β$_n$)), n ≥ 1	≡ [**start**(μ(search(α$_1$; head(β$_1$))) **&&** μ(α$_2$)), **start**(μ(rParent( )))) **&&** μ(leftCond(α$_2$; β$_2$;…; α$_n$; β$_n$))
18	μ(rightCond(α; β))	≡ **true**
19a	μ(rightCond(α$_1$; **β**; α$_2$; **β**))	≡**true** *// same right boundary*
19b	μ(rightCond(α$_1$; β$_1$; α$_2$; β$_2$))	≡ [**end**(μ(search(α$_2$; head(β$_2$)))) **when** μ(search(α$_1$; head(β$_1$))), **end**(μ(search(β$_1$; head(α$_1$)))))
19c	μ(rightCond(α$_1$; β$_1$; α$_2$; β$_2$^{e}))	≡**!** [**start**(μ(β$_2$) && μ(search(α$_1$; head(β$_1$)))), **end**(μ(search(β$_1$; head(α$_1$))))) *// eventual*
19d	μ(rightCond(α$_1$; β$_1$; α$_2$; β$_2$^{r}))	≡ (μ(search(α$_2$; head(β$_2$))) && !μ(β$_2$)) ‖ *// response* [**start**(μ(search(!β$_2$; head(β$_2$))) **&&** μ(α$_2$)), **start**(μ(α$_2$)))
20a	μ(rightCond(α$_1$; β$_1$; α$_2$; β$_2$; …; α$_n$; β$_n$)), n ≥ 1	≡ [**end**(μ(search(α$_2$; head(β$_2$)))) **when** μ(search(α$_1$; head(β$_1$))), **end**(μ(search(rParent( ); head(lParent( )))))) **&&** μ(rightCond(α$_2$; β$_2$; …;α$_n$; β$_n$))
20b	μ(rightCond(α$_1$; β$_1$; α$_2$; β$_2$^{e}; …; α$_n$; β$_n$)), n ≥ 1	≡ ! [**start**(μ(β$_2$) && μ(search(α$_1$; head(β$_1$)))), **end**(μ(search(rParent( ); head(lParent( )))))) **&&** μ(rightCond((α$_2$; β$_2$; …;α$_n$; β$_n$)) *// eventual*
21	μ(search(→g, k))	≡ [**start**(μ(g)), **start**(μ(k)) **)**
22	μ(search((→g$_1$,→g$_2$),k))	≡ [**start**(μ(g$_1$)), **start**(μ(k)) **) &&** μ(g$_2$)
23	μ(search((→g$_1$,→g$_2$, …, →g$_n$),k)), n ≥ 3	≡ ([**start**(μ(g$_1$)), **start**(μ(k))**) &&** μ(g$_2$)) **&&** μ(search((→g$_2$, …,→g$_n$),k))
24	head(→g$_1$, →g$_2$, …, →g$_n$), n ≥ 1	≡ g$_1$
25	last(→g$_1$, …, →g$_n$), n ≥ 1	≡ g$_n$
26	lParent(α$_1$;β$_1$; α$_2$;β$_2$; …; α$_n$; β$_n$), n≥1	≡ α$_1$
27	rParent(α$_1$;β$_1$; α$_2$;β$_2$; …; α$_n$;β$_n$), n≥1	≡ β$_1$
28	□ (f)	≡ [→¬ (f)	→) false *// henceforth*
29	◊ (f)	≡ [–	→ f^e) false *// eventually*
30	□ (p → ◊s)	≡ [→p	→ s^r) false *// response*

Table A-2. Transformation steps for [→*l* | →r)*p*

N	TRANSFORMED FORMULA	JUSTIFICATION
1.	μ([→l |→r)p)	Initial formula
2.	[end([μ(left(μ(lInterval([→l|→r)p))), μ(right(μ(rInterval([→l|→r)p)))))), start(μ(last(→l))))	Rule 8 is applied to Step 1 to create the structure of the main condition.
3.	[end([μ(left(→l; →r; p)), μ(right(→l; →r)))), start(μ(l)))	Rules 9 and 11 are applied to Step 2 to convert the interval into a list of pairs of search patterns.
4.	[end([start(μ(leftCond(→l; →r))) when !p start(μ(r)) when μ(rightCond(→l; →r)))), start(l))	Rules 14b and 14 are applied to Step 4 to intersect *p* with interval *lr* and assert the end of the interval.
5.	[end([start(μ(search(→l; head(→r)))) when !p, start(r) when true)), start(l))	Rules 15 and 18 are applied to Step 4 to intersect the beginning of nested intervals and assert *true* since there are no nested intervals.
6.	[end([start(μ(search(→l; r))) when !p, start(r) when true)), start(l))	Rule 24 is applied to Step 5 to obtain the first search of the right search pattern.
7.	[end([start([start(μ(l)), start(μ(r)))) when !p, start(r))), start(l))	Rule 21 is applied to Step 6 to convert a pair of search patterns into a condition.
8.	start([end([start([start(l), start(r))) when !p, start(r))), start(l)))	Rule 1 is applied to Step 7 to transform the primitive propositions and step 14 of algorithm Map.