Oracles for Checking Temporal Properties of Concurrent Systems*

Laura K. Dillon and Qing Yu
Department of Computer Science
University of California, Santa Barbara 93106

Abstract

Verifying that test executions are correct is a crucial step in the testing process. Unfortunately, it can be a very arduous and error-prone step, especially when testing a concurrent system. System developers can therefore benefit from oracles automating the verification of test executions.

This paper examines the use of Graphical Interval Logic (GIL) for specifying temporal properties of concurrent systems and describes a method for constructing oracles from GIL specifications. The visually intuitive representation of GIL specifications makes them easier to develop and to understand than specifications written in more traditional temporal logics. Additionally, when a test execution violates a GIL specification, the associated oracle provides information about a fault. This information can be displayed visually, together with the execution, to help the system developer see where in the execution a fault was detected and the nature of the fault.

1 Introduction

Testing is a necessary activity in all stages of system development to assure the final system is of high quality. While testing complex systems is extremely labor intensive and tedious, many of the tasks performed during testing are routine and repetitive. Thus, system developers can benefit from tools that automate various testing tasks.

One such task involves the verification of test executions. Once test executions are generated, they must be examined to determine if the system behaves correctly. If the test executions exhibit a fault, then the nature of the fault must be ascertained. Typically, large numbers of test executions must be examined in order to ensure sufficient 'coverage' of the space of possible executions of a system. When testing concurrent systems, the need to ensure coverage of the system's global synchronization structure dramatically increases the sizes of test sets. Moreover, verifying correctness of an execution of a concurrent system requires examination of the ordering and timing of events that occur throughout the potentially long execution. Thus, verification of test executions tends to be tedious and error-prone, especially when testing concurrent systems.

Test oracles based on formal specifications can automatically verify that test executions conform to the specifications [16]. In particular, oracles based on temporal specifications permit verification of the ordering of events in executions of concurrent systems.

This paper examines the use of Graphical Interval Logic (GIL) [5] for specifying temporal properties of concurrent systems and describes a method for constructing oracles from GIL specifications. The visually intuitive representation of GIL specifications makes them easier to develop and to understand than specifications written in more traditional (textual) temporal logics. Additionally, when a test execution violates a GIL specification, the associated oracle provides information about a fault. This information can be displayed visually, together with the execution, to help the system developer see where in the execution a fault was detected and the nature of the fault.

Previous work on GIL produced a toolset that supports the specification of systems in GIL and supports reasoning about properties guaranteed by GIL specifications [10]. More recently, researchers at the University of California, Irvine have produced a comprehensive toolkit, called TAOS (Testing and Analysis with Oracle Support) [15], that supports the definition of oracles in GIL.

The rest of the paper is organized as follows. Section 2 describes the textual logic on which GIL is based and then indicates how textual formulas are represented in GIL. Section 3 presents some sample specifications, which we use in Section 4 to demonstrate the capabilities of GIL-based oracles. Section 5 presents the algorithm for verifying the correctness of a test execution and Section 6 shows how information about a fault is displayed so as to help the system developer understand what has occurred. We discuss related work in Section 7 and future research directions in Section 8.

2 The Logic

We believe the visual representation of GIL formulas is more natural and intuitive than that of textual temporal logics. In particular, the timing-diagram look and feel of GIL corresponds closely to one's intuitive mental picture of time.

However, textual representations produce more concise formulas and facilitate definitions of semantics and of algorithms that manipulate formulas. Future Interval Logic (FIL) was invented to allow textual encoding of the semantic information in a GIL formula.[1] We use a fragment of FIL in Section 5 for describing the construction of GIL-based oracles. This section describes the FIL fragment used in this paper and then indicates how FIL formulas are represented graphically in GIL.

FIL is a propositional, linear-time temporal logic. A FIL formula is evaluated at a state in an infinite sequence of states with the semantics for the boolean connectives defined in the usual manner. A formula viewed as a specification must hold at the first state of all state sequences that represent possible executions of a system. (A finite execution determines a finite state sequence, which we identify with the infinite sequence obtained by stuttering the terminal state.) In the following informal description, we use the term "context" to mean a sequence of states obtained by extracting a subsequence of contiguous states from a given sequence and, if the subsequence is finite, stuttering its final state.

The key construct of FIL is the interval, which provides a context over which properties are asserted to hold. The typical interval modality in FIL has the form $[\theta_1|\theta_2)$, where θ_1 and θ_2 specify series of searches to states in the future at which designated target formulas hold.[2] The modality $[\theta_1|\theta_2)$ extracts the subcontext of the current context beginning with the state located by the searches specified in θ_1 and ending with that preceding the state located by the searches specified in θ_2. If this subcontext is finite, it is made infinite by stuttering the last state. If this subcontext can be constructed within the present context, then the FIL formula $[\theta_1|\theta_2)f$ requires that the formula f hold at the first state within the subcontext. If the target of a search in θ_1 or θ_2 cannot be located in the current context or if θ_2 locates the current state (the state at which θ_2 is evaluated), then $[\theta_1|\theta_2)$ cannot be constructed and the formula $[\theta_1|\theta_2)f$ is regarded to hold vacuously. For brevity, we refer to a state at which a target formula a holds as an a-state. We say that a search to a "succeeds," if the future within the present context contains an a-state (the search locates a state at which a holds), and "fails," otherwise. If a search fails, then all subsequent searches in a search pattern are also regarded to fail.

For example, in the modality $[\triangleright a_1, \triangleright a_2|\triangleright a_3)$, each $\triangleright a_i$ denotes a search to the formula a_i, $i = 1, 2, 3$, and the comma denotes composition of searches. The left endpoint of this interval is found by searching for the first a_1-state and then from that state for the first a_2-state. The right endpoint of the interval is found by starting at the left endpoint and searching for the first a_3-state.[3] The interval can be constructed if all three searches succeed and if a_3 does not hold at the state located by the search to a_2 (the interval is not empty). In this case, the formula $[\triangleright a_1, \triangleright a_2|\triangleright a_3)f$ asserts that f holds at the first state within the context that begins with the state located by the search to a_2 and that extends up to, but does not include, the state located by the search to a_3. If any of the searches fails or if the search to a_3 locates the current state, then $[\triangleright a_1, \triangleright a_2|\triangleright a_3)f$ holds vacuously.

FIL defines two trivial search patterns, in addition to the non-trivial search patterns obtained by composing one or more searches. The trivial search patterns are used in the following situations: The modality $[-|\triangleright\theta_2)$ attempts to construct a prefix of the present context (ending with the state preceding that located by θ_2), and the modality $[\theta_1|\triangleright)$ attempts to construct a suffix of the present context (beginning with the state located by θ_1).

The henceforth $\Box$ and eventually $\Diamond$ operators of Propositional Temporal Logic (PTL) are defined in FIL as derived operators: $\Box f$ abbreviates the formula

[1] "Future" because all searches are into the future of the current state.

[2] We adopt a non-strict interpretation of "future", which includes the present state, in order that FIL is insensitive to finite stuttering of states [11]. Thus, if the target of a search holds at the current state (the state at which the search is evaluated), then the search locates the current state.

[3] More generally, FIL permits the specification of independent searches, in which the searches for the right endpoint start at the same state as the searches for the left endpoint. However, we do not consider independent searches in this paper.

$[\triangleright\neg f|\triangleright)\mathit{false}$ and $\Diamond f$ abbreviates $\neg\Box\neg f$. Informally, $\Box f$ requires that the formula $\neg f$ never holds in the future of the current state (this interpretation follows from the 'default-to-true' semantics of the search to $\neg f$) and $\Diamond f$ requires that there is some state in the future at which f holds.

The default-to-true semantics of the interval and search modalities described above yields "weak" operators in the sense that the targets of searches need not be located and the specifications for the beginning and ending of an interval can designate an empty context. Strong versions of the interval and search operators are provided in FIL as derived operators, using definitions similar to those illustrated above for henceforth and eventually. A strong search to a formula a is denoted $\triangleright\!\!\triangleright a$, and a strong interval with endpoints specified by θ_1 and θ_2 is written $[\![\theta_1\|\theta_2)\!)$. Searches in the patterns θ_1 and θ_2 can be strong or weak. A weak search has the interpretation described above. A strong search is predicated on locating the targets of all prior searches in θ_1 and θ_2, if any prior searches exist. Under this condition, the strong search must not fail. Similarly, if the search targets in θ_1 and θ_2 are located, then $[\![\theta_1\|\theta_2)\!)$ requires the interval to be non-empty.

These conventions yield the following interpretations for the interval formulas $[\theta_1|\theta_2)f$ and $[\![\theta_1\|\theta_2)\!)f$. Both formulas require that

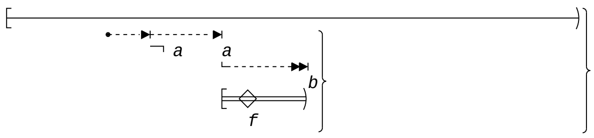
- if the interval modality can be constructed within the present context, then f holds at the first state within this interval, and
- a strong search in θ_1 or θ_2 can fail only if a prior weak search fails.

Additionally, $[\![\theta_1\|\theta_2)\!)$ requires that

- if the searches in θ_1 and θ_2 succeed, then θ_2 does not locate the current state (the interval is not empty).

Consider, for example, the formula $g \equiv \Box[\![\triangleright\neg a, \triangleright a\|\triangleright\!\!\triangleright b)\!)\Diamond f$. In effect, the weak searches predicate the invariant in g on locating a positive transition of the formula a. If a positive transition of a is located, the strong search guarantees the existence of a future b-state. However, b must not hold at the state following a positive transition of a since the strong interval precludes an empty context. Thus, g requires that every positive transition of a causes a (later) positive transition of b and that f holds at some state within every interval that begins with the state following a positive transition of a and ends with the state preceding the next positive transition of b.

The graphical representation of this formula in GIL is



Briefly, a GIL formula is read from top to bottom and left to right. The topmost interval represents the outermost context (implicit in the corresponding FIL formula) and the horizontal dimension represents the progression of states in time (time increases from left to right). Composition of searches is represented by horizontal concatenation of search arrows and search targets are left-justified below the arrowheads. The type of arrowhead indicates the type of search: a single arrowhead for a weak search and a double arrowhead for a strong one. Similarly, the type of line segment denoting an interval indicates the type of interval: a single line for a weak interval and a double line for a strong one.

To assert that a formula holds at the first state in an interval, the formula is drawn left-justified below the left endpoint of the interval. To assert that a formula is invariant over an interval, the formula is positioned below the interval and indented to the right of its left endpoint. To assert that a formula eventually holds within an interval, a diamond is placed on the interval and the formula is left-justified below the diamond. To assert that a formula holds at the first state of a suffix interval $[\theta_1|\triangleright)$, the formula is left-justified beneath a triangle positioned directly below the final arrowhead in the representation of θ_1. In all cases, the horizontal extent of a formula lies within the horizontal extent of the contexts in which the formula is nested.

Boolean composition can be laid out vertically in GIL; in vertical layout, a conjunction is indicated by stacking formulas one below the other without the conjunction operator. Right braces are used to visually delimit formulas and eliminate ambiguity.

In general, the formulas a, b, and f in the above examples can be arbitrary. In this paper, however, we require the targets of all searches to be strictly propositional and we do not permit nesting of intervals. A large range of temporal properties can be naturally expressed in the corresponding fragment of GIL. Moreover, the semantics of formulas containing temporal search targets or nested intervals can be counterintuitive due to the interaction of default valuations (when searches fail or intervals are empty). Further, state sequences are more simply and efficiently checked in the absence of these features.

A detailed definition of GIL can be found in [6], and of FIL in [13]. The former paper also defines the translation of graphical formulas to textual ones.

3 Example Specifications

This section illustrates the use of GIL for specifying temporal properties of a sample concurrent system. The specifications are for an Ada simulation of an automated gas station derived from that presented in [8]. In this simulation, an `Operator` task models a human operator who oversees the use of several gasoline pumps by multiple customers. The pumps are modeled by an array of `Pump` tasks and customers by an array of `Customer` tasks. For simplicity, we consider a version with a single `Pump` task. `Customers` is the name given to the array of `Customer` tasks.

A `Customer` prepays before pumping gasoline and collects change afterward. Prepayment is represented by a rendezvous on the entry `Operator.Pre_pay`. The amount paid and the index of the `Customer` task are passed to `Operator` in the rendezvous. A `Customer` calls `Pump.Start_pumping` to start pumping and `Pump.Finish_pumping` to finish, obtaining the amount charged from `Pump` as a result of the latter rendezvous. After pumping, a `Customer` receives change from `Operator` in a rendezvous on the `Customer`'s `Change` entry.

`Pump` is activated by a call from `Operator` to `Pump.Activate` for a limited amount of gasoline, after which `Pump` accepts a call to `Pump.Start_pumping` followed by a call to `Pump.Finish_pumping`. It then notifies `Operator` of the current charges in a call to `Operator.Charge` and waits to be reactivated.

`Operator` maintains a queue of prepayment records for `Customer` tasks that have prepaid and not yet received change. It activates `Pump` upon adding a `Customer` to an empty queue and after dispensing change if the queue is not empty. Upon accepting a call on its `Charge` entry, `Operator` dequeues the first prepayment record and dispenses change to the `Customer` indicated in the record.

Although the gas station example is a relatively simple concurrent program, it exhibits subtle temporal properties that are difficult to verify. The debugger described in [8] was used to discover deadlocks in two versions of the program. The third version, which we use below, prevents this particular deadlock from occurring but contains a different deadlock, which can only occur when more than two customers have prepaid for gasoline.

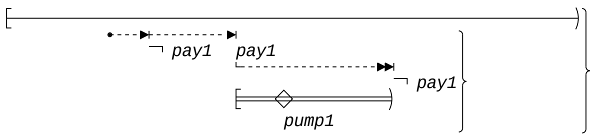
The primitive propositions in a GIL formula can be viewed as representing "conditions" that summarize important properties of execution histories. For the purposes of the example, we assume that conditions are triggered by the occurrence of specific events and cancelled by others, although more flexible definitions for conditions could be used. We further assume that the initiation and termination of rendezvous define events.

For GIL formulas to describe properties of executions of a system, definitions are needed for the conditions appearing in the formulas. Various methods could be used for defining conditions. For example, the programmer might annotate a program with formal comments specifying the initial values of conditions and the events that trigger and cancel conditions. For the gas station example, we assume definitions for two types of conditions: *pay$n* and *pump$n*, where $n represents an integer in the indexing range for `Customers`. By definition, all conditions are initially false; *pay$n* is triggered by initiation of a rendezvous with `Customers`(*$n*) on `Operator.Pre_pay` and cancelled by initiation of a rendezvous with `Operator` on `Customers`(*$n*)`.Change`; and *pump$n* is triggered by initiation of a rendezvous with `Customers`(*$n*) on `Pump.Start_pumping` and cancelled by termination of a rendezvous with `Customers`(*$n*) on `Pump.Finish_pumping`. Thus, *pay$n* is true when `Customers`(*$n*) has prepaid but not yet received change, and *pump$n* is true when `Customers`(*$n*) is pumping gasoline.

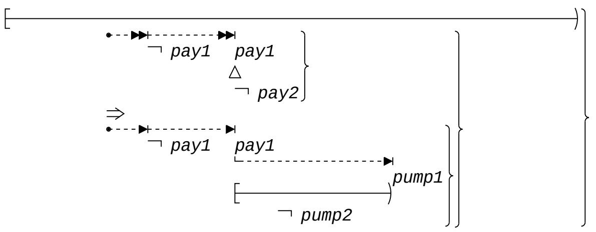
The following specifications are representative of the kinds of temporal properties that one might wish to verify about the gas station system.

SafePump1. `Customers`(1) pumps gasoline only when it has prepaid.

DoesPump1. Whenever `Customers`(1) prepays, it eventually pumps gasoline and, at some later point, it receives change.



Fair1.2.
If `Customers`(1) prepays before `Customers`(2), then `Customers`(2) does not gain access to the pump before `Customers`(1).



For convenience, we shorten the names of conditions (primitive propositions) in later sections of the paper. Using p for *pay1*, m for *pump1*, p' for *pay2*, and m' for *pump2*, we obtain the following FIL translations for the GIL specifications shown above:

- *SafePump1*: $\Box(m \Rightarrow p)$
- *DoesPump1*: $\Box[\![\rhd\neg p, \rhd p \| \rhd\!\!\rhd\neg p))\Diamond m$
- *Fair1.2*: $\Box([\rhd\!\!\rhd\neg p, \rhd\!\!\rhd p | \rhd)\neg p' \Rightarrow [\rhd\neg p, \rhd p | \rhd m)\Box\neg m')$

4 Using Test Oracles

GIL specifications describe properties of state sequences, whereas oracles check properties of system executions. To reconcile these two viewpoints, system executions are represented as traces of events. The definitions of conditions determine events that must be monitored during testing and the system developer might indicate additional events of interest. Traces generated during testing induce state sequences on which to evaluate specifications.[4] The traces satisfy the GIL specifications if the specifications are true when evaluated on these state sequences.

Figure 1 shows a prefix of an event trace that was generated by executing the Ada simulation of the automated gas station described above. We have traced only the events that affect conditions in the specifications. A vertical dashed line labeled `acc T1:T2.E` (`end T1:T2.E`) denotes the initiation (termination) of a rendezvous with `T1` on `T2.E`, where long identifiers are shortened to save space. The pseudo-event `initial` represents the start of the execution. The state sequence induced by the trace is displayed graphically below the trace. Shaded bars show where conditions are true and clear bars show where they are false. The full trace is easily seen to satisfy the specifications in Section 3.

If a trace violates a GIL specification, an oracle can construct a formula describing the trace and contradicting the specification. The contradiction produced by the oracle will either be the negation of the specification or a special case of said negation.[5] To help the system developer discover the nature and location of a fault, the oracle can display the trace and the contradiction one below the other and horizontally aligned, so as to show the points in the trace at which individual subformulas in the contradiction hold.

Figures 2 and 3 show faulty traces that were generated by executing the gas station program. We trace some call events in addition to rendezvous events to help in understanding the behavior in Figure 2. In this trace, the second rendezvous with `Customers`(1) on `Operator`.`Pre_pay` produces a positive transition of *pay1* that is never followed by a negative transition. Thus, the strong search in the invariant of *DoesPump1* fails when evaluated at any state between the first and second rendezvous with `Customers`(1) on `Operator`.`Pre_pay`. This is shown by the contradiction displayed below the trace. Alignment shows states at which subformulas hold. Inspection of the trace shows that the system is deadlocked at the end of the execution.

The trace in Figure 3 violates *Fair1.2*. As shown by the contradiction displayed below the state sequence, the first rendezvous with `Customers`(1) on `Operator`.`Pre_pay` produces a positive transition of *pay1* when *pay2* does not hold. However, *pump2* holds within the interval that begins after this transition and that extends up to the next *pump1*-state. Thus, the invariant in *Fair1.2* is violated at the first state of the execution.

5 Constructing Test Oracles

Automata theory provides the theoretical foundation for constructing oracles from GIL specifications. A GIL formula determines an equivalent finite state automaton, which accepts precisely those state sequences satisfying the formula. Thus, a state sequence s violates a GIL specification f if the automaton associated with $\neg f$ accepts s. By using the automaton for $\neg f$, rather than f, information in the accepting run can be used, in case s violates f, to align the trace that produced s with a formula contradicting f and satisfied by s. For convenience, we give all results in this section in terms of FIL formulas, which are obtained by translating (restricted) GIL specifications into FIL.

The automata constructed by the FIL decision pro-

[4] If a system is distributed, then an execution defines a partial order on events and a trace is any linear ordering that is consistent with this partial order.

[5] For brevity, we say "contradiction", rather than "contradiction to the specification", and do not mean "contradiction" in the sense of being unsatisfiable.

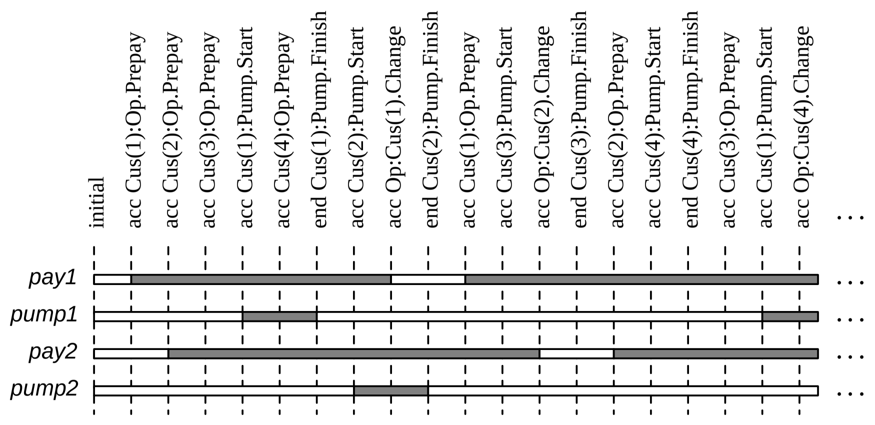


Figure 1: A sample trace and the associated state sequence.

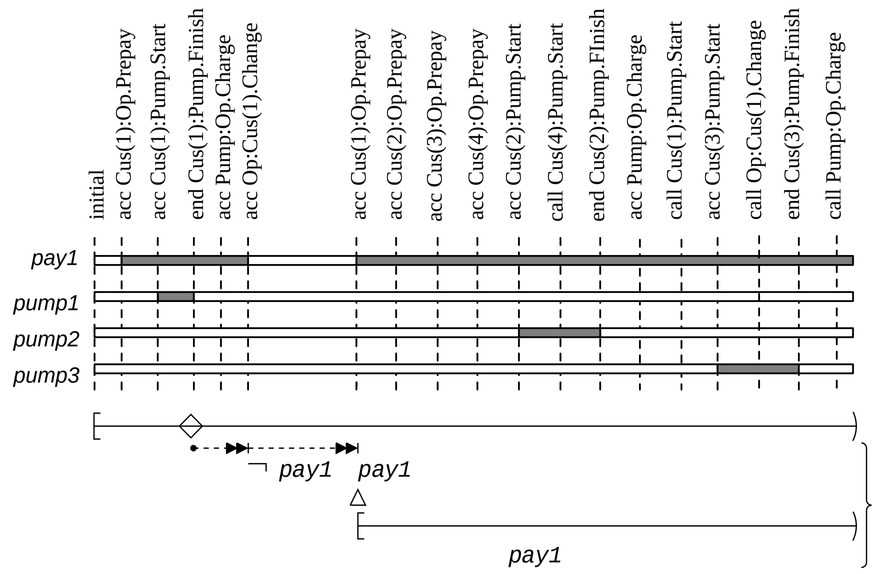


Figure 2: A trace violating *DoesPump1* and a formula that shows a fault.

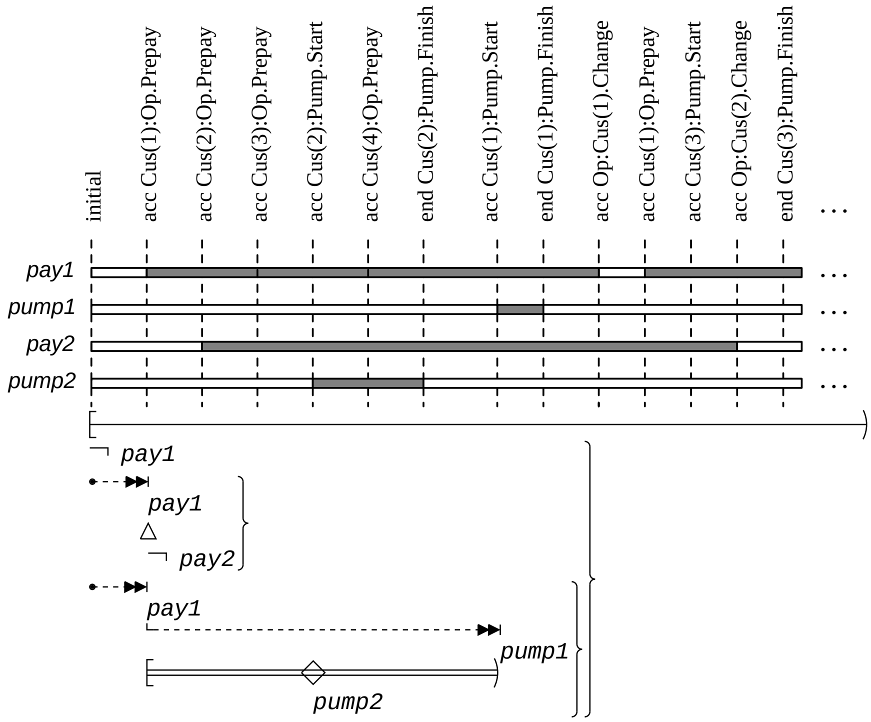


Figure 3: A trace violating *Fair1.2* and a formula that shows a fault.

cedure [13] do not provide a practical foundation for an oracle procedure. In the first place, the decision procedure assumes formulas are expressed exclusively in terms of weak searches, weak intervals and propositional connectives. Higher level temporal operators must be translated into weak searches and weak intervals. This increases the size of formulas and the depth of interval nesting. A minimal set of temporal primitives simplifies the definition of the automaton for a formula, but generally requires a larger, more complex formula, which in turn produces a (exponentially) larger automaton.

Additionally, the FIL decision procedure requires Bucchi automata [3] because FIL formulas are interpreted over infinite state sequences. However, the event traces generated during testing are finite. Thus, oracles do not require the power, or additional complexity, of Bucchi automata.

This section describes an algorithm for associating finite state automata with FIL formulas that provides a more practical foundation for oracles. The algorithm requires a FIL formula in which negation has been pushed into propositions; the only boolean connectives are conjunction, disjunction and negation; search targets are strictly propositional; and the maximum depth of interval nesting is one. Given a formula f of the required form, the algorithm builds a deterministic finite state automaton (DFA) that accepts precisely those *finite* state sequences satisfying f. We denote the DFA for a formula f as D_f.

The DFA for a formula is produced using a variation of the well-known tableau method[6] that simultaneously builds and determinizes the automaton for restricted FIL formulas. The nodes of D_f are annotated with sets of formulas that are derived from f through a series of reductions.[7] Intuitively, the sets annotating a node can be viewed as imposing alternative requirements on the future of a state sequence: The remaining state sequence must satisfy the formulas in at least one of the 'requirement sets' associated with the node.

5.1 Definitions and terminology

We adhere to the following notational conventions in the description below. The symbols p, q (with or without subscripts) denote primitive propositions; a, b denote propositional formulas; and θ denotes a search pattern, either a (possibly empty) sequence of

[6]See, for example, the satisfiability algorithm for PTL presented in [12].

[7] We refer to the states of the automaton as 'nodes', rather than 'states', to avoid confusion between the states of the automaton and the states of a concurrent system, which comprise the automaton's input alphabet.

searches or a trivial search pattern. The symbols f, g denote FIL formulas of the required form. We assume that all formulas are syntactically well-formed, so that the context in which a symbol appears may impose additional restrictions on its form. We write $\{\theta_1|\theta_2\}$ to represent either of the interval modalities $[\theta_1|\theta_2)$ or $[\![\theta_1\|\theta_2)\!)$ and $*a$ to represent either of the search modalities $\triangleright a$ or $\triangleright\!\triangleright a$.

The next operator $\odot$ of PTL is not expressible in FIL. However, tableau methods use the next operator to describe requirements that must be checked at the next state of a sequence of states. We therefore use the next operator, in this section only, in describing the construction of the DFA. As in PTL, the formula $\odot f$ holds at the i^{th} state of a sequence of states if f holds at the $i+1^{th}$ state of the sequence. A formula of the form $\odot f$ is said to be "deferred" and f is called the "core" of $\odot f$. We refer to a formula that does not contain the next operator as a "basic formula."

Tableau methods typically represent formulas as sets. The manipulation of formulas is then easily described using set operations. In the following, we regard a set X of formulas as representing the conjunction of its elements, $\wedge_{f\in X} f$, and a set Y of sets of formulas as representing the disjunction of (the formulas represented by) its elements, $\vee_{X\in Y}(\wedge_{f\in X} f)$. We abuse notation slightly, identifying a set representing a formula with the formula itself. Thus, for instance, we say that X and Y are equivalent if the formulas that they represent are equivalent. By convention, the empty set represents *true*.

We refer to primitive propositions and their negations as "atomic formulas." A set of atomic formulas that does not contain a primitive proposition and its negation is said to be "consistent." The symbol A denotes a consistent set of atomic formulas regarded as "assumptions" that are made about the current state. The identification of A with the conjunction of its elements permits us to write $A \Rightarrow a$ to signify that the assumptions in A guarantee that the (propositional) formula a holds in the current state.

A family of parameterized reduction relations are used in generating the requirement sets for the successors of a node in a DFA. Intuitively, if f is a requirement that must hold at the current state, a is assumed to hold in the current state, and a reduces f to g, written $f \underset{a}{>} g$, then we may substitute g, a boolean combination of syntactically simpler formulas and deferred formulas, for f.

Definition 1 The relations $\underset{a}{>}$ are defined to be the smallest relations satisfying the following rules.

- $\Box f \underset{true}{>} (f \wedge \odot\Box f)$ and $\Diamond f \underset{true}{>} (f \vee \odot\Diamond f)$
- $a \underset{a}{>} \odot true$ and $a \underset{\neg a}{>} \odot false$
- $\{\theta_1|\theta_2\}f$:
  - $\{*a,\theta_1|\theta_2\}f \underset{a}{>} \{\theta_1|\theta_2\}f$, if θ_1 is not empty
  - $\{*a|\theta_1\}f \underset{a}{>} \{-|\theta_1\}f$
  - $\{*a,\theta_1|\theta_2\}f \underset{\neg a}{>} \odot\{*a,\theta_1|\theta_2\}f$
- $\{-|\theta_1\}f$:
  - $\{-|*a,\theta_1\}f \underset{a}{>} \{-|\theta_1\}f$, if θ_1 is not empty
  - $[-|*a)f \underset{a}{>} \odot true$ and $[\![-\|*a)\!)f \underset{a}{>} \odot false$
  - $\{-|*a,\theta_1\}b \underset{\neg a\wedge b}{>} \odot[-|*a,\theta_1)true$
  - $\{-|*a,\theta_1\}b \underset{\neg a\wedge\neg b}{>} \odot[\![-\|*a,\theta_1)\!)false$
  - $\{-|*a,\theta_1\}\Box f \underset{\neg a}{>}$ $\{-|*a,\theta_1\}f \wedge \odot[-|*a,\theta_1)\Box f$
  - $\{-|*a,\theta_1\}\Diamond f \underset{\neg a}{>}$ $\{-|*a,\theta_1\}f \vee \odot[\![-\|*a,\theta_1)\!)\Diamond f$
  - $\{-|\triangleright\}f \underset{true}{>} f$
- $f \underset{b}{>} g$, if $f \underset{a}{>} g$ and $b \Rightarrow a$

The reduction of f to g under the assumption a is formally justified by observing that $f \underset{a}{>} g$ only if the conjunctions $f \wedge a$ and $g \wedge a$ are equivalent.

The formulas used in representing requirements can be characterized as follows.

Definition 2 Given a basic FIL formula f, C_f is defined to be the smallest set of formulas such that

- $f \in C_f$;
- if $g_1 \in C_f$, the main connective of g_1 is a boolean operator or the next operator, and g_2 is an operand of said connective, then $g_2 \in C_f$; and
- if $g_1 \in C_f$ and g_1 reduces to g_2 (under any assumption) then $g_2 \in C_f$

A "requirement set," denoted by the symbol R, is a subset of C_f that is regarded as constraining the future of the current state: The conjunction of the formulas in R must hold at the current state. A set of requirement sets, denoted by the symbol S, is regarded as imposing alternative requirements on the future: At least one of the requirement sets in S must hold at the current state. We say R is "deferred" if all formulas in R are deferred. If R is deferred, then we define $Core(R) \equiv \{\, f \mid \ \odot f \in R \,\}$. Similarly, we say S is deferred if all requirement sets in S are deferred.

We extend the scheme for representing formulas as sets to also specify a pairing of formulas regarded as expressing assumptions with those regarded as expressing requirements. If X is a set representation for a formula and A is a set of assumptions, we identify the pair (A, X) with the conjunction $A \wedge X$, and we identify a set Y of such pairs with the disjunction $\vee_{(A,X) \in Y}(A \wedge X)$. Intuitively, a pair $(A, X) \in Y$ represents one way of satisfying Y: Assuming that A is *true* in the current state, Y holds at the current state if X does.

The symbol N denotes a node of a DFA. The input alphabet for D_f is the set of states (interpretations for primitive propositions). However, we label transitions in D_f with propositional formulas, rather than states, and interpret the label on a transition as describing the states on which the transition is taken. Thus, a transition in D_f corresponds to multiple transitions in a conventional DFA representation (in which input symbols label transitions). A transition from N_i to N_j with label a is denoted (N_i, a, N_j). The labels on the transitions leaving a node "partition" the state space, in the sense that one and only one of the labels holds in any state.

5.2 The Decomposition Procedure

We first describe the decomposition procedure that is the heart of the algorithm for constructing D_f. This procedure decomposes a set of requirement sets S into a set of assumption-requirement pairs $\{ (A_i, S_i) \mid 1 \leq i \leq k \}$, for some $k \geq 1$, with the following properties

(1) S is equivalent to $\vee_{1 \leq i \leq k}(A_i \wedge \odot S_i)$;

(2) the assumptions A_i, $1 \leq i \leq k$, partition the state space; and

(3) either $(\Box A_i) \Rightarrow S_i$ or $(\Box A_i) \Rightarrow \neg S_i$, $1 \leq i \leq k$.

If S is the annotation for a node N, properties (1) and (2) justify a transition with label A_i from N to a node N_i that is annotated with S_i. Property (3) permits N_i to be classified as accepting (final) or not. If A_i holds in the last state of an input sequence, then $(\Box A_i) \Rightarrow S_i$ guarantees that S_i holds at the last state, so that N_i is a final node. If $(\Box A_i) \Rightarrow \neg S_i$, then N_i is not accepting.

We define a predicate that is used to ensure (3). The rules in Figure 4 define *Accept*(a, f), for a basic formula f and propositional formula a, provided that a is strong enough to evaluate the conditions in the rules. It can be shown that (3) is implied by *Accept*(A_i, S_i).

The decomposition procedure makes use of the following operations, which are defined for a set X that pairs assumptions with (flat) requirement sets.

f	$Accept(a, f)$
$g_1 \wedge g_2$	$Accept(a, g_1) \wedge Accept(a, g_2)$
$g_1 \vee g_2$	$Accept(a, g_1) \vee Accept(a, g_2)$
$\Box g$	$Accept(a, g)$
$\Diamond g$	$Accept(a, g)$
$\{*b, \theta_1 \mid \theta_2\}g$	$Accept(a, \{\theta_1 \mid \theta_2\}g)$, if $a \Rightarrow b$, and θ_1 is not empty
$\{*b \mid \theta_2\}g$	$Accept(a, \{- \mid \theta_2\}g)$, if $a \Rightarrow b$
$\{\triangleright b, \theta_1 \mid \theta_2\}g$	*true*, if $a \Rightarrow \neg b$
$\{\triangleright\!\!\triangleright b, \theta_1 \mid \theta_2\}g$	*false*, if $a \Rightarrow \neg b$
$\{- \mid *b, \theta_2\}g$	$Accept(a, \{- \mid \theta_2\}g)$, if $a \Rightarrow b$, and θ_2 is not empty
$[- \mid *b)g$	*true*, if $a \Rightarrow b$
$[\![- \,\|\, *b)\!)g$	*false*, if $a \Rightarrow b$
$\{- \mid \triangleright b, \theta_2\}g$	*true*, if $a \Rightarrow \neg b$
$\{- \mid \triangleright\!\!\triangleright b, \theta_2\}g$	*false*, if $a \Rightarrow \neg b$
$\{- \mid \triangleright\}g$	$Accept(a, g)$
b	*true*, if $a \Rightarrow b$
b	*false*, if $a \Rightarrow \neg b$

Figure 4: Rules for evaluating *Accept*(a, f)

Each operation can be shown to transform a given assumption-requirement pairing into any equivalent one.

Boolean decomposition. If the pair $(A, R) \in X$ and if the formula $f_1 \wedge f_2 \in R$, then replace (A, R) in X with $(A, R - \{ f_1 \wedge f_2 \} \cup \{ f_1, f_2 \})$.

If $(A, R) \in X$ and if $f_1 \vee f_2 \in R$, then replace (A, R) in X with $(A, R - \{ f_1 \vee f_2 \} \cup \{ f_1 \})$ and $(A, R - \{ f_1 \vee f_2 \} \cup \{ f_2 \})$.

Requirement reduction. If the pair $(A, R) \in X$, $f_1 \in R$, and $f_1 \underset{A}{>} f_2$, then replace (A, R) in X with $(A, R - \{ f_1 \} \cup \{ f_2 \})$.

Assumption introduction. If $(A, R) \in X$, $A_1 \vee A_2 =$ *true*, and $A \cup A_1$ and $A \cup A_2$ are consistent, then replace (A, R) in X with $(A \cup A_1, R)$ and $(A \cup A_2, R)$.

Subsumption. If $(A_1, R_1) \in X$, $(A_2, R_2) \in X$, $R_1 \Rightarrow R_2$, and $A_1 \Rightarrow A_2$, then delete (A_1, R_1) from X.

If $(A, R) \in X$, $f_1 \in R$, $f_2 \in R$, and $f_2 \Rightarrow f_1$, then replace (A, R) in X with $(A, R - \{ f_1 \})$.

The decomposition procedure is shown in Figure 5. Step 1 initializes X to be equivalent to S. X is then decomposed in steps 2 and 3 using the operations defined above to obtain an equivalent representation of S in which all requirements are deferred, distinct assumptions are pairwise inconsistent, and assumptions are strong enough to determine acceptance. In step 4, sets of requirement sets are formed from the cores of requirements sets that are associated with identical assumptions. Clearly, the formula $\vee_i(A_i \wedge \odot S_i)$, Input: A set S of requirement sets

Output: A set Y of the form $\{\, (A_i, S_i) \mid 1 \le i \le k \,\}$, $k \ge 1$, such that $\vee_{1 \le i \le k} A_i = \mathit{true}$; $A_i \cup A_j$ is inconsistent, if $1 \le i < j \le k$; $\mathit{Accept}(A_i, S_i)$ is defined, for $1 \le i \le k$; and S is equivalent to $\vee_{1 \le i \le k}(A_i \wedge \odot S_i)$.

1. Set $X := \{\, (\emptyset, R) \mid R \in S \,\}$.
2. Repeatedly apply operations to X until all requirement sets appearing in X are deferred.
3. Continue applying assumption introduction to X until distinct assumptions appearing in X are pairwise-inconsistent and $\mathit{Accept}(A, \mathit{Core}(R))$ is defined, for each $(A, R) \in X$.
4. Assuming X has the form $\left\{\, (A_i, R_i) \;\middle|\; 1 \le i \le k' \,\right\}$, $k' \ge 1$, set $Y := \cup_{1 \le i \le k'} \{\, (A_i, S_i) \,\}$, where $S_i \equiv \cup_{j \in J_i} \{\, \mathit{Core}(R_j) \,\}$, $J_i \equiv \{\, j \mid A_i = A_j \,\}$.

Figure 5: The decomposition procedure

where i ranges over representative elements from distinct J_i, is equivalent to X, and, hence, also to S. The disjunction of the assumptions in Y is *true* since the only operations that affect assumptions are assumption introduction and subsumption, and these operations preserve the disjunction of the assumptions (which is initially *true*). Thus, the decomposition procedure is correct.

Termination of step 2 follows from the observations that pairing a requirement that is not deferred with strong enough assumptions permits it to be decomposed using boolean decomposition or requirement reduction, and that assumptions can be strengthened to whatever degree is necessary using assumption introduction. (If $(A, R) \in X$ and A does not assume a valuation for p, then (A, R) can be split into $(A \cup \{\, p \,\}, R)$ and $(A \cup \{\, \neg p \,\}, R)$.) The conditions for termination of step 3 are met if assumptions are introduced for each atomic formula in f, although assumptions seldom need to be split to this extent. Thus, the decomposition procedure terminates. The subsumption operations can be used to streamline the decomposition procedure, but are not needed for correctness.

The size of Y is sensitive to the choice of operations that are applied to X and to the order in which they are applied. In particular, assumption introduction should be used, in step 2, only when needed to reduce a requirement and, in step 3, only when necessary to obtain mutually inconsistent assumptions or to determine acceptance. Subsumption can help reduce the sizes of requirement sets, as well as the number of requirement sets that are generated. However, if applied inconsistently, subsumption can increase the size of the DFA and, if applied indiscriminately, the cost of checking the conditions for subsumption can be prohibitive. The development of heuristics for streamlining the decomposition procedure is a topic of ongoing research.

5.3 The DFA construction

Figure 6 shows the algorithm for constructing D_f from a formula f of the required form.

Input: A basic FIL formula f in which negation has been pushed into propositions; the only boolean connectives are $\wedge$, $\vee$, and $\neg$; search targets are strictly propositional; and the maximal depth of interval nesting is 1

Output: A set *NODES* of nodes
A set *START* of start nodes,
A set *FINAL* of accepting nodes,
A set *TRANS* of transitions
An annotation function $S\colon \mathit{NODES} \to 2^{C_f}$

1. Set $\mathit{TRANS} := \mathit{FINAL} := \emptyset$.
2. Create a (unmarked) node N_0.
3. Set $\mathit{NODES} := \mathit{START} := \{\, N_0 \,\}$ and $S(N_0) := \{\, \{\, f \,\}\,\}$.
4. While some node in *NODES* is not marked:
   a. Select an unmarked node $N \in \mathit{NODES}$ and mark it.
   b. Use the procedure in Figure 5 to decompose $S(N)$ into the pairs (A_i, S_i), $1 \le i \le k$, $k \ge 1$.
   c. For each i, $1 \le i \le k$:
      If $\mathit{Accept}(A_i, S_i)$ then
         If some $N_j \in \mathit{FINAL}$ satisfies $S(N_j) = S_i$,
            add (N, a_i, N_j) to *TRANS*;
         else create a (unmarked) node N_i,
            add N_i to *NODES* and to *FINAL*,
            add (N, a_i, N_i) to *TRANS*, and
            set $S(N_i) := S_i$;
      elseif some $N_j \in \mathit{NODES} - \mathit{FINAL}$ satisfies $S(N_j) = S_i$,
         add (N, a_i, N_j) to *TRANS*;
      else create a (unmarked) node N_i
         add N_i to *NODES*,
         add (N, a_i, N_i) to *TRANS*, and
         set $S(N_i) := S_i$;

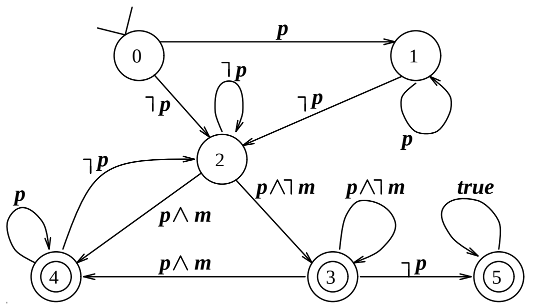
Figure 6: The DFA construction algorithm

A start node is created in step 2 and annotated with the single requirement set $\{\, f \,\}$ in step 3, indicating that f must hold at the first state of the input sequence. Step 4 generates the transitions leaving a node and the corresponding successor nodes, repeatedly, until all successors of existing nodes have been generated. A marking scheme is used to distinguish nodes whose successors have yet to be calculated (unmarked nodes) from nodes that have been selected for said calculation (marked nodes). Step 4 is guaranteed to terminate because C_f is finite.

We use the negation of *DoesPump1* to illustrate the DFA constructed by this algorithm. Pushing negation into an interval formula changes the modes (strong to weak and weak to strong) of the interval and of the searches for its endpoints. Thus, we write the negation of *DoesPump1* as

$$f_1 \equiv \Diamond[\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m$$

and then apply the algorithm of Figure 6 to f_1. This produces the automaton shown in Figure 7, where the nodes of D_{f_1} are annotated with the requirement sets shown below the automaton. Nodes 3–5 each contain an accepting requirement set, indicated in the diagram by an "(A)." These are, therefore, final nodes.



Node 0

$\{\ \Diamond[\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$

Node 1

$\{\ [\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$
$\{\ \Diamond[\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$

Node 2

$\{\ [\triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$
$\{\ \Diamond[\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$

Node 3

$\{\ [-|\triangleright\neg p)\Box\neg m\ \}$ (A)
$\{\ [\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$
$\{\ \Diamond[\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$

Node 4

$\{\ [\![-\|\triangleright\neg p))\mathit{false}\ \}$ (A)
$\{\ [\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$
$\{\ \Diamond[\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m\ \}$

Node 5

$\{\ \mathit{true}\ \}$ (A)

Figure 7: The DFA for $f_1 \equiv \Diamond[\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m$.

In the worst case, the size of the DFA constructed by the algorithm in Figure 6 can be exponentially larger than that of the corresponding nondeterministic finite automaton (NFA) produced by the GIL proof checker. However, preliminary investigation suggests that the DFAs for formulas expressing standard safety and liveness requirements are typically much smaller than the corresponding NFAs and that, on such formulas, the running time of the algorithm for building DFAs is much better. For example, from the formula f_1, the GIL proof checker first builds a 'local automaton' containing 63 nodes and 75 transitions. Elimination of unsatisfiable eventualities reduces these numbers by approximately 50%. Similarly, the DFA $D_{\neg Fair1.2}$ constructed using the algorithm in Figure 6 contains 8 nodes and 27 transitions, while the corresponding local automaton produced by the GIL proof checker contains 677 nodes and 1,055 transitions, which numbers are then reduced by approximately 85%.

6 Displaying a Fault

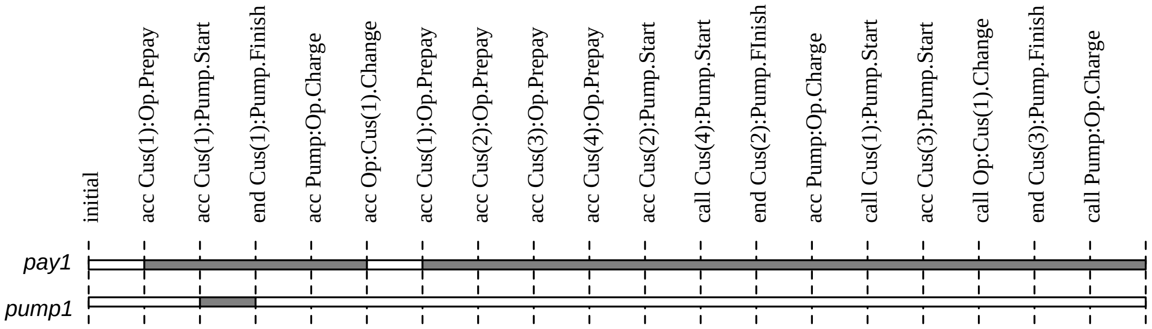
If a trace violates a specification, the oracle should provide the system developer with information about where a fault was detected and the nature of the fault. The graphical representation of GIL permits a formula describing a trace and contradicting a specification to be displayed, appropriately aligned with the trace, in order to help the system developer see what has occurred.

While we have yet to develop a systematic method for displaying faults that can be shown to work in all cases, we have obtained hueristics that appear to work well for specifications of standard system properties. This section presents an example illustrating the basic technique.

We consider the trace in Figure 2, which is reproduced in Figure 8. Node numbers below the vertical dashed lines in Figure 8 indicate the nodes encountered in the run of D_{f_1} on this state sequence. For convenience in the discussion below, we write N for the sequence of nodes in the accepting run and $N(i)$ for the ith node in N, $i = 0 \ldots 19$. Thus, $N(0)$ is Node 0, $N(1)$ is Node 2, and so on. Similarly, we write s for the state sequence induced by the trace and $s(i)$ for the ith state in the state sequence. Thus, $s(i)$ is represented by the bar lines between the dashed lines over $N(i)$ and $N(i+1)$, $i = 0 \ldots 18$. We also use the following abbreviations for various formulas in C_{f_1}

$$\begin{aligned}
f_1 &\equiv \Diamond[\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m \\
f_2 &\equiv [\triangleright\!\triangleright\neg p, \triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m \\
f_3 &\equiv [\triangleright\!\triangleright p|\triangleright\neg p)\Box\neg m \\
f_4 &\equiv [-|\triangleright\neg p)\Box\neg m
\end{aligned}$$

The method for building the contradiction begins by choosing a path R through the requirement sets in



N:	0	2	3	4	4	4	2	3	3	3	3	3	3	3	3	3	3	3	3	3
R:	$\{f_1\}$	$\{f_1\}$	$\{f_1\}$	$\{f_1\}$	$\{f_2\}$	$\{f_2\}$	$\{f_3\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$	$\{f_4\}$
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Figure 8: Run of D_{f_1} on the state sequence generated by the trace in Figure 2 and the sequence of requirement sets used for displaying the contradiction.

N that generates the accepting requirement set $\{ f_4 \}$ in the final node of N. A path is generated in reverse order. We start by taking $R(19) \equiv \{ f_4 \}$ and, given suitable choices for $R(i+1), R(i+2), \ldots, R(19)$, choose $R(i) \in N(i)$ such that $(A(i), \odot R(i+1))$, where A_i denotes the label of the transition from $N(i)$ to $N(i+1)$, is generated by decomposition of the assumption-requirement pair $(\emptyset, R(i))$, for $i = 18 \ldots 0$. For the example, we take R to be the sequence of requirement sets shown below N in Figure 8, that is, we let $R(i)$ denote the requirement set directly below $N(i)$, $i = 0 \ldots 19$.

The requirement sets extracted from the run of D_{f_1} on s indicate states at which formulas in C_{f_1} hold. We use the requirement sets in R to determine how to display a contradiction that shows a fault in the trace. We can assert f_2 at any state i such that $f_2 \in R(i+1)$. For the example, we choose the first such i. We therefore align a diamond with $s(3)$ and look for a formula that can be drawn at $s(3)$ to assert f_2. The next two changes in the requirement sets are brought about by reducing $f_2 \in R(5)$ by $\neg p \in s(5)$ to f_3 and by reducing $f_3 \in R(6)$ by $p \in s(6)$ to f_4. These reductions signify that, when f_2 is evaluated at $s(3)$, the strong searches to $\neg p$ and to p locate, respectively, $s(5)$ and $s(6)$. The failure to reduce $\{ f_4 \}$ indicates that the last search to $\neg p$ ($\neg$*pay1*) fails, i.e. p is invariant over the future of $s(6)$. Thus, we obtain the contradiction displayed in Figure 2 using the requirement sets shown in Figure 8.

Different paths through the requirement sets in the accepting run can produce other contradictions or alter the manner in which a contradiction is displayed. For example, we can take

- $R(i) \equiv \{ f_1 \}$, for $i = 0 \ldots 5$
- $R(6) \equiv \{ f_3 \}$, for $i = 6$ and
- $R(i) \equiv \{ f_4 \}$, for $i = 7 \ldots 19$.

With this definition for R, it takes two steps to generate $R(6)$ from $R(5)$. The reduction of $R(5) \equiv \{ f_1 \}$ by *true* generates $\{ f_2 \}$, which $\neg p$ then reduces to $R(6) \equiv \{ f_3 \}$. The first of these reductions causes the diamond to be aligned with $s(6)$ and the second, which corresponds to locating the first search target in f_2 at $s(6)$, causes $\neg p$ and f_3 to be drawn at this point. This latter choice of requirement sets thus results in the contradiction being displayed as shown in Figure 9.

7 Related Work

Model checking is typically used for verifying that a concurrent system satisfies temporal specifications [1, 4]. However, because model checking requires examination of all reachable system states, it suffers from state explosion. Such exhaustive techniques are computationally infeasible except for special classes of systems, and so their potential for practical use is also limited. Dynamic analysis methods, such as testing and run-time monitoring, must be used when exhaustive analysis techniques are impractical. Moreover, even when model checking is feasible, testing is needed to check that the model analyzed is an accurate representation of the final system.

Interval logic (IL) [18] largely inspired the design of FIL. However, there are several presentational as well as semantic differences between IL and FIL, which

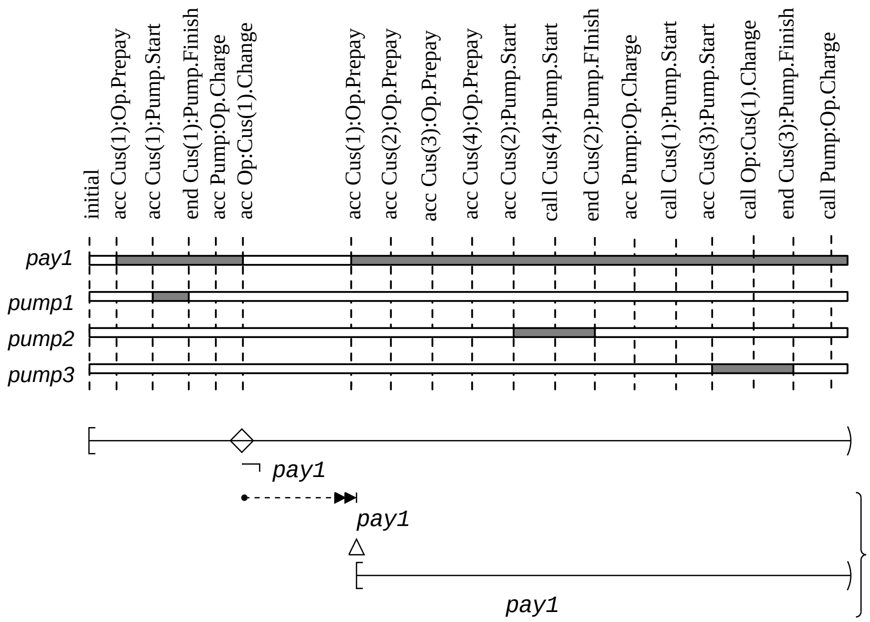


Figure 9: Contradiction displayed using a different sequence of requirement sets than that in Figure 8.

makes them different with respect to expressiveness, decision procedures and complexity. A detailed comparison of IL and FIL can be found in [13].

Oracles for concurrent systems can be based on any formal specification language with a suitable procedure for checking traces. The IDD debugger for distributed programs uses a limited subset of IL for expressing synchronization constraints. IDD checks assertions at runtime, stopping execution if an assertion is violated [7].

Real-Time Interval Logic (RIL) [14] was designed to provide oracles for real-time systems. RIL is undecidable, but permits traces to be efficiently checked. An RIL trace checker is currently being integrated into the TAOS toolkit for use with RIL oracles.

Sequencing constraints expressed in TSL (Task Sequencing Language) are used as oracles by the TSL Runtime System, which monitors executions of Ada tasking programs [17]. Also, state transition diagrams have been interpreted as describing sequencing constraints and used as oracles for testing protocols [2]. However, TSL and state transition diagrams are not formal logics and do not support reasoning about properties of systems to the same degree as GIL.

TAOS supports the use of GIL oracles by means of a very simple prototype trace checker [15]. The checker scans the trace several times to locate intervals and evaluate formulas within them. Thus, it does not scale for use with long traces and does not support on-line checking.

In contrast, the approach described in this paper depends on DFAs that are derived from GIL formulas (via translation to FIL). It permits traces to be checked as they are generated and a violation of a specification to be reported as early in a trace as possible. Consequently, our approach can support run-time monitoring and debugging, in addition to postexecution checking of logged traces. Also, when a trace violates a specification, we are able to construct a formula that describes a fault in the trace, providing valuable feedback to the user. The formula and trace can be displayed graphically to help the user see the violation that was detected.

Since concurrent systems are often distributed and are inherently nondeterministic, collecting traces and regenerating executions pose difficult research problems. We obtained the traces for this paper by manually instrumenting the Ada source program and executing it on a uniprocessor. Delay statements were inserted to induce different behaviors.

8 Future Directions

Automating the techniques described in this paper will permit us to undertake a longer term study aimed at assessing the effectiveness of GIL-based oracles. We therefore plan to automate the algorithm for checking traces described in Section 5. Our tools will build on the existing GIL tools and be integrated into the TAOS toolset, which provides comprehensive support for testing, including generation of traces and management of test artifacts.

To automate the display of faults, it will be necessary to refine the heuristics illustrated in Section 6 into a systematic procedure. We are also investigating more effective methods for displaying traces. Display tools should support different views of traces at different levels of abstraction in order to help the user interactively examine sections of a trace.

Studies involving substantial applications will also permit us to investigate tradeoffs between expressiveness and efficiency of analysis. The current reduction rules can be extended to accommodate arbitrary GIL formulas with no restrictions on search targets or nesting of intervals. We have not found reason to use such features in numerous experiments with GIL. However, these experiment involved relatively small-scale applications. Complex properties of larger systems may be more naturally expressed in less restrictive subsets of GIL, which admit suitable trace checking procedures. For example, searches to interval formulas might be prohibited and nested intervals otherwise permitted up to a limited depth. We hope to achieve a tenable balance between expressiveness and analyzability through such investigation.

Acknowledgements: The authors would like to acknowledge John Gannon for his advice and helpful comments on the first draft of this paper.

References

[1] J M Atlee and J Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. Software Engineering*, 19(1):24–40, Jan 1993.

[2] G V Bochmann, R Dssouli, and J R Zhao. Trace analysis for conformance and arbitration testing. *IEEE Trans. Software Engineering*, 15(11):1347–1355, Nov 1989.

[3] J R Büchi. On a decision method in restricted second order arithmetic. In *Proc. 1960 Congress Logic, Methodology and Philosophy of Science*, pp 1–11. Stanford University Press, 1960.

[4] E M Clarke, E A Emerson, and A P Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2):244–263, Apr 1986.

[5] L K Dillon, G Kutty, L E Moser, P M Melliar-Smith, and Y S Ramakrishna. Graphical specifications for concurrent software systems. In *Proc. 14th IEEE Inter. Conf. Software Engineering*, pp 213–224, Melbourne, May 1992.

[6] L K Dillon, G Kutty, L E Moser, P M Melliar-Smith, and Y S Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Trans. Software Engineering and Methodology*. To appear.

[7] P K Harter, D M Heimbigner, and R King. Idd: An interactive distributed debugger. In *IEEE Inter. Conf. Distributed Computer Systems*, pages 498–506, 1985.

[8] D P Helmbold and D C Luckham. Debugging Ada tasking programs. *IEEE Software*, pp 47–57, Mar 1985.

[9] R A Kemmerer, ed. *Proc. SIGSOFT '89 3rd ACM Symp. Testing, Analysis and Verification*, Key West, FL, Dec 1989. *ACM SIGSOFT Software Engineering Notes* 14(8).

[10] G Kutty, Y S Ramakrishna, L E Moser, L K Dillon, and P M Melliar-Smith. A graphical interval logic toolset for verifying concurrent systems. In *Proc. 4th Conf. Computer Aided Verification, LNCS #697*, pp 138–153, Jul 1993. Springer-Verlag.

[11] L Lamport. What good is temporal logic? In *Proc. IFIP Congress*, pp 657–668, Paris, 1983.

[12] Z Manna and P Wolper. Synthesis of communicating processes from temporal logic specifications. In *Proc. Work. Logics of Programs, LNCS*. Springer-Verlag, 1981.

[13] Y S Ramakrishna, L K Dillon, L E Moser, P M Melliar-Smith, and G Kutty. An automata-theoretic decision procedure for future interval logic. In *Proc. 12th Conf. Foun. Software Technology and Theoretical Computer Science, LNCS #652*, pp 51–67, Dec 1992. Springer-Verlag.

[14] R R Razouk and M M Gorlick. A real-time interval logic for reasoning about executions of real-time programs. In Kemmerer [9], pp 10–19.

[15] D J Richardson. TAOS: Testing with analysis and oracle support. In *Proc. Inter. Symp. Software Testing and Analysis*, Seattle, WA, Aug 1994.

[16] D J Richardson, S L Aha, and T O O'Malley. Specification-based test oracles for reactive systems. In *Proc. 14th Inter. Conf. Software Engineering*, pp 105–118, Melbourne, AUS, May 1992.

[17] D Rosenblum. Specifying concurrent system with TSL. *IEEE Software*, pp 52–61, May 1991.

[18] R L Schwartz, P M Melliar-Smith, and F H Vogt. An interval logic for higher-level temporal reasoning. In *Proc. 2nd ACM Symp. Principles of Distributed Computing*, pp 173–186, Montreal, Canada, Aug 1983.