Specification and Testing of Temporal Properties of Concurrent System Designs

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

Abstract

This paper investigates the use of Graphical Interval Logic for specification and testing of concurrent and real-time systems. Past research produced a toolset for creating specifications in GIL and for reasoning about properties of systems that satisfy their specifications. Current research is exploring the use of GIL specifications during system design and implementation.

Our work to date has focused primarily on using GIL specifications as oracles during testing of executable designs. The paper presents an example in which GIL specifications for an Ada design are used to uncover faults during testing. It also shows how the graphical representation of formulas can be used to advantage in displaying a trace that violates a specification. The testing methods have yet to be automated.

1 Introduction

This paper describes ongoing research on the use of formal specifications and testing during design of concurrent and real-time systems.

We use a highly intuitive visual notation, called Graphical Interval Logic (GIL) [7] for specifying temporal properties of designs. Specifications in GIL resemble the 'back-of-envelope' timing diagrams that system designers and software engineers often draw when reasoning about a system under construction. A prototype toolset allows designers to draw GIL specifications and check the validity of proofs in the logic [12]. In particular, a designer can verify that the design specifications imply critical correctness requirements (expressed in GIL), thereby certifying correctness of any design satisfying the specifications.

GIL specifications for a design describe constraints on the temporal sequencing of conditions that summarize important properties of execution histories. For example, if the conditions *Req* and *Lock* signify, respectively, that a client has requested a resource of a server and that the server has locked the resource, then a constraint might require that *Req* precedes *Lock*. Ideally, a design would be formally analyzed to determine if it satisfies its temporal specifications. However, formal verification of designs for even moderate-sized concurrent systems is often infeasible.

Testing of designs is required to validate the design specifications, as well as to validate requirements that cannot be inferred from the design specifications. A test execution that violates a specification reveals a fault in the design or a fault in the specification (i.e. the specification might not faithfully model the property that the designer intended). A test execution that violates a system requirement reveals a fault in the design and, if the design specifications imply the requirement, also a fault in the specifications.

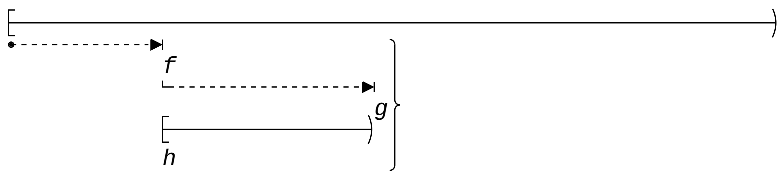
The paper is organized as follows. Section 2 provides a brief introduction to GIL. Ideas for using GIL specifications during testing of designs are outlined in Section 3 and restrictions on the form of GIL specifications that expedite testing are discussed in Section 4. Section 5 indicates how the methods discussed in Section 3 can be extended to permit testing of designs for real-time systems. Finally, Section 6 mentions related work and contains closing remarks.

2 Overview of Graphical Interval Logic

GIL is a linear-time temporal logic. Formulas are interpreted on sequences of states, which represent possible executions of a system. The key construct of GIL is the interval. An interval represents a sequence of contiguous states of a computation and provides a context within which properties can be asserted to hold. The progression of states in time is shown using the horizontal dimension. The vertical dimension shows the composition of formulas from subformulas. Formulas are read from top to bottom and left to right. The topmost interval represents an entire execution of the system.

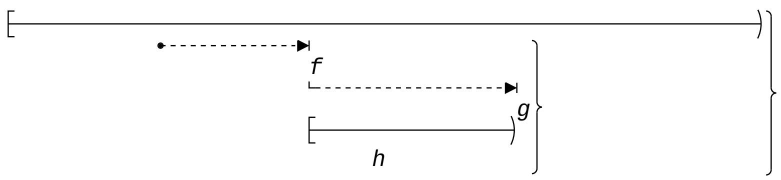
A typical interval is constructed by specifying two search patterns: one to locate the left endpoint of the interval and the other to locate its right endpoint. A search pattern consists of a sequence of searches. Searches are performed sequentially, with each new search starting at the state located by the previous search. A search within a search pattern locates the first state at which a target formula (left justified below the head of the search arrow) holds. An interval begins with the state located by the first search pattern and extends up to but does not include the state located by the second search pattern.

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. For example,



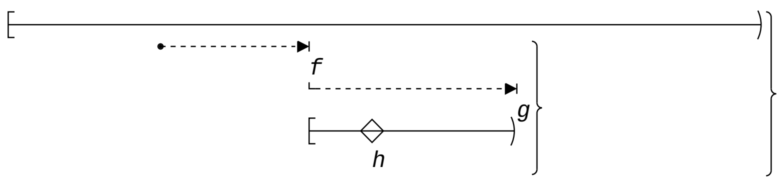
asserts that h holds at the first state in the interval that begins with the first state at which f holds and ends just prior to the next state at which g holds. For brevity, we often refer to a state at which a formula f holds as an f-state. The right brace alongside an interval formula helps visually to delimit the formula. In general, any GIL formula can serve as the target of a search operator. For the discussion that follows, however, we restrict our attention to GIL formulas in which search targets are strictly propositional. Section 4 discusses the reasons for this restriction.

To express an invariant (henceforth) property that holds throughout an interval, the invariant is positioned below the interval and indented to the right of the bracket representing the interval's left endpoint. For example,



asserts that h holds throughout every interval that begins with an f-state and extends up to, but does not include, the next g-state.

To express an eventuality property, a diamond is placed on the interval, with the formula that eventually holds left-justified below the diamond. For example,



(1)

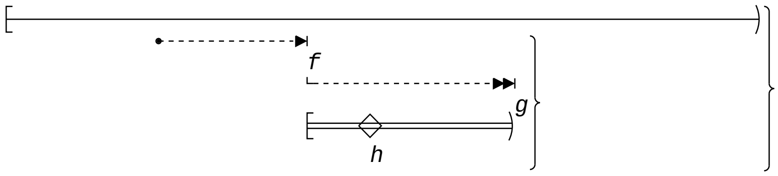
asserts that every interval that begins with an f-state and extends up to, but does not include, the next g-state contains an h-state.

If the target of a search does not hold at any state in the future of the state at which the search begins, the search to the formula fails.[1] If either of the searches for the left or right endpoints of

[1] GIL assumes a non-strict interpretation of the future, which includes the present. Thus, the future of a state begins with the state itself.

an interval fails or if the state located by the search for the right endpoint coincides with the state located by the search for the left endpoint, the interval cannot be constructed. By convention, every formula holds vacuously on an interval that cannot be constructed. Because of this, the searches and intervals in the previous examples are referred to as *weak* searches and *weak* intervals, respectively.

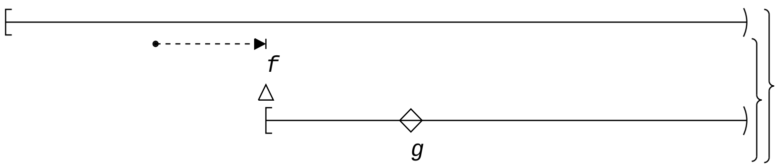
GIL also provides *strong* searches, denoted by double arrows, and *strong* intervals, denoted by double lines, as in



Here the strong search requires that, whenever the search to f succeeds, then the search to g must also. The strong interval requires that the interval is non-empty provided that the searches to f and g do not fail. This formula is thus stronger than (1) in that it also requires that every f-state leads to a subsequent g-state and that f and g never hold simultaneously. This formula illustrates the natural and visual manner by which an interval limits the scope of a nested formula. In comparison, the equivalent formula in Propositional Temporal Logic [8] is expressed

$$\Box(\neg f \ \mathrm{U}\ (f \wedge \neg g \ \mathrm{U}\ (h \wedge \neg g \wedge \Diamond g)))$$

GIL provides a *point* operator, denoted by a triangle, which locates a state specified by a search pattern and, when the state can be located, asserts that the formula left-justified below the point operator holds at that state. For example



asserts that every f-state leads to a g-state. In this example, searches are not required for specifying the endpoints of the nested interval because the interval starts immediately at the point located by the point operator and extends to the end of the enclosing interval.

GIL provides a special operator for expressing real-time properties. The *len* operator bounds the duration of the remaining context. Section 5 discusses this operator and gives an example real-time specification.

GIL formulas can be combined using standard logical infix operators laid out vertically. In vertical layout a conjunction is indicated by stacking the formulas one below the other without the conjunction operator. Braces are used to disambiguate formulas. The syntax and semantics of GIL are described in more detail in [7]

A prototype toolset supports the use of GIL for specifying and reasoning about properties of concurrent systems. The current prototype provides a syntax-directed editor for constructing and editing graphical formulas on a workstation display; an automated proof-checker for checking the validity of inferences in GIL and for producing counterexamples to invalid inferences; and a proof management system for tracking proof dependencies and for storing and retrieving graphical formula. The GIL graphical editor interfaces with all tool components, so that the software engineer works exclusively with graphical formulas. The GIL toolset is the topic of [12]. Experiences with the toolset are reported in [6, 12, 16, 19].

3 Using Specifications During Testing of Designs

Testing is a very laborious process, particularly when concurrency is involved. Ideally, some criteria is used for selecting test executions that provide sufficient 'coverage' of the space of possible executions of a design. Coverage criteria for designs of concurrent systems must ensure coverage of the design's global synchronization structure [27, 28], which dramatically increases the size of test sets. The test executions must then be examined for correctness. Determining correctness of an execution of a concurrent system requires examination of the temporal ordering and timing of events that occur during the execution. Manually checking that the test executions are correct is error-prone, and not likely to be practical.

This section presents an example that illustrates ideas for using GIL specifications during design testing. In particular, the example shows how oracles developed from GIL specifications can be used to automatically determine if a test execution satisfies necessary temporal properties. In cases where a fault is detected, the faulty execution and a formula describing the fault can be displayed so as to help the software engineer understand the nature of the fault and where it occurred.

For the example, we hypothesize a high-level design for a simple elevator system written in Ada. The appendix gives designs for the two main tasks: `Control` and `Elevator`. Line numbers are provided for reference purposes, and are not part the design. We assume the tasks are declared in a main procedure that issues repeated calls to `Control.Req` with random values (in the range

Cond.	ON		OFF	
	Event	Precond.	Event	Precond.

at$n	`e33:Control.At($n,_)`	`true`	`e20:Floor:=Floor+1`
			`e23:Floor:=Floor-1`
			`e27:Floor:=Floor-1`
			`e30:Floor:=Floor+1`
				`true`

req$n	`c15:accept Req($n)`	`not open($n)`	`e35:Open:=true`	`at($n)`
open$n	`e35:Open:=true`	`at$n`	`e40:Open:=false`	`true`
goingup	`c29:Dir:=Up`	`true`	`e22:Dir:=Down`	`true`

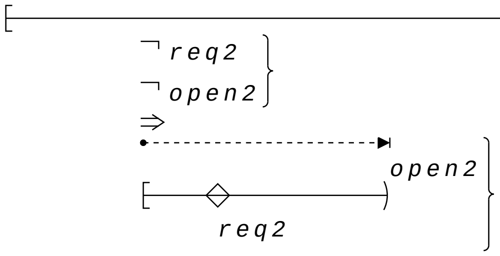
Table 1: Condition definitions (`$n: Floor_range`)

`Floor_range`) for `For_floor`, modeling passengers requesting service at arbitrary floors. `Control` keeps track of outstanding requests from passengers. It calls `Elevator.Disp` to dispatch the elevator, informing it of the range of floors with outstanding requests. For simplicity, we do not distinguish requests made inside the elevator from those made at a floor, nor do we associate a direction with requests. `Elevator` calls `Control.At` when it arrives at a floor to see if the floor needs to be serviced. When `Elevator` is next dispatched, it determines the direction of movement based on the value of `Dir` and the range of floors that require service.

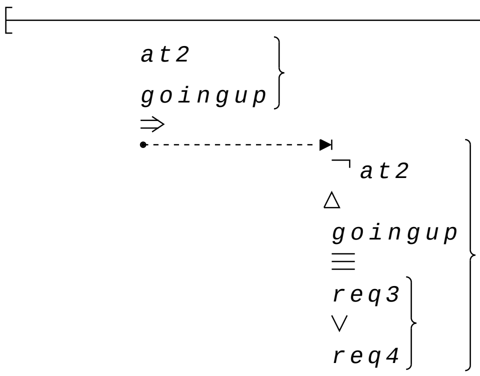
GIL specifications are expressed in terms of conditions that summarize important properties of execution histories. For the purposes of the example, we assume that conditions are triggered by the occurrence of specific events and cancelled by others, although more flexible methods for defining conditions could be used. We define an event to be an execution of an atomic operation subject to certain preconditions. For simplicity, we regard assignments and the evaluation of guards to be atomic. We also assume that issuing an entry call, initiating a rendezvous and completing a rendezvous are atomic operations. Table 1 defines conditions of interest for the elevator example. The first three rows of the table use an abbreviation for defining multiple conditions, determined by substituting values from `Floor_range` for '*$n*'. Thus, for example, *at1* is turned on by a call from `Elevator` to `Control.At` with the value 1 for `Curr_floor` and it is turned off by an assignment of a new value to `Floor`. Similarly, a rendezvous on `Control.Req` turns on *req1* when the value 1 is passed in for `For_floor` and *open1* is off, and executing the assignment `e35:Open:=true` turns *req1* off when *at1* is on.

Given these conditions, the designer might define the following design specifications. These specifications are not meant to be comprehensive, but rather representative of different types of temporal properties.

ReqService2: The doors open at floor 2 only in response to a request for service at the floor.

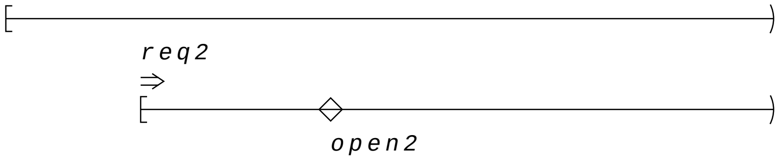


ContinueUp2: If the elevator was going up when it arrived at floor 2, then it continues going up precisely if a passenger requires service at a higher floor by the time the elevator leaves floor 2.



After augmenting these specifications with specifications for other expected liveness and fairness properties, the designer might verify that the design specifications ensure the following requirement, as illustrated in [6].

ServeRequest2: The elevator eventually responds to every call for service.



The method for producing an oracle from a GIL formula (which may represent a design specification or a system requirement) is straightforward. Event traces are generated from the test executions, where the definitions for the conditions appearing in the formula determine the events to be traced.[2] The traces are then checked to determine if they satisfy the formula in question. Section 4 discusses the algorithm that is used for checking traces.

[2] If a design is distributed, then an execution defines a partial order on events. In this case, a trace is any linear ordering that is consistent with this partial order.

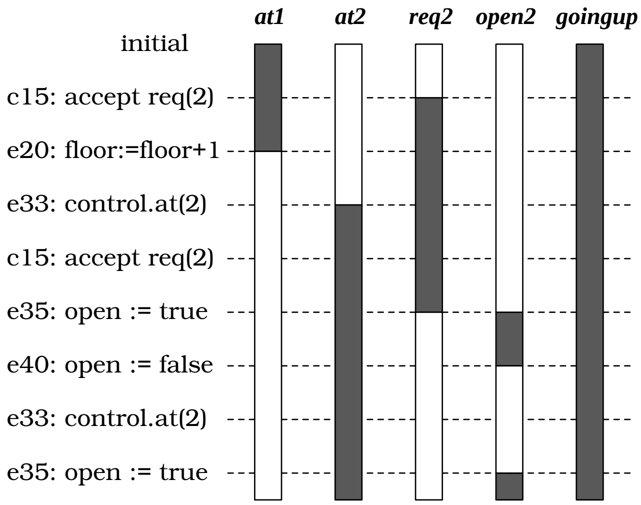
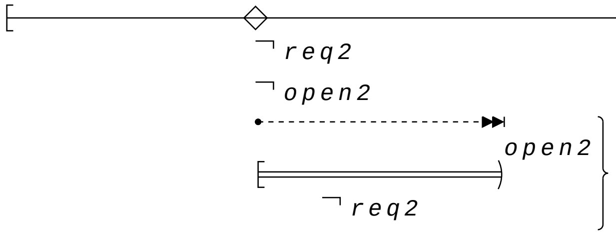


Figure 1: A sample trace

Figure 1 shows a sample trace representing an execution of the design for the elevator example. The event trace is shown to the left of the bar lines, with the pseudo-event `initial` representing the start of the execution. The relevant conditions are listed along the top of the bar lines. Shading shows where conditions are on. This trace is easily shown to satisfy *ContinueUp2* and *ServeRequest2*. However, it violates *ReqService2*.

If a trace violates a specification, then it satisfies the negation of the specification. Thus, the trace in Figure 1 satisfies



(2)

The designer is better able to see how and where the trace violates the specification if the trace and the negated specification are displayed one below the other and aligned so as to show the points at which subformulas in the negated specification are satisfied. For example, the trace in Figure 1 can be drawn horizontally, rather than vertically, with (2) positioned below it so that the outer context line spans the entire trace, the diamond is aligned with (the event) `e40:Open:=false`, and the head of the strong search arrow is aligned with second occurrence of `e35:Open:=true`.

This information can help the designer discover the reason behind the fault. In this case, the problem is caused by the second occurrence of `c15:accept Req(2)`, which takes place after `e33:Control.At(2,_)`. Conceptually, the designer considers this request to have been served at the next occurrence of `e35:Open:=true`, which models the opening of the elevator doors. However, `Control` clears the requests for a floor when the elevator arrives at the floor instead of when the doors close. At this point the designer must decide if the trace represents an acceptable execution and the specification is too strong, or if the design must be corrected.

If the design is modified so that `Elevator` calls an entry in `Control` to signal when it is closing its doors, similar race conditions can result in traces that violate *ServeRequest2*. At any rate, a physical elevator system will necessarily exhibit certain race conditions, such as a passenger pressing the button for a floor while the doors are closing. A more realistic specification for an elevator system would take into account the passage of time, and not just the temporal ordering of events. The specification and testing of real-time properties of designs are discuss below.

4 Restrictions on GIL Specifications

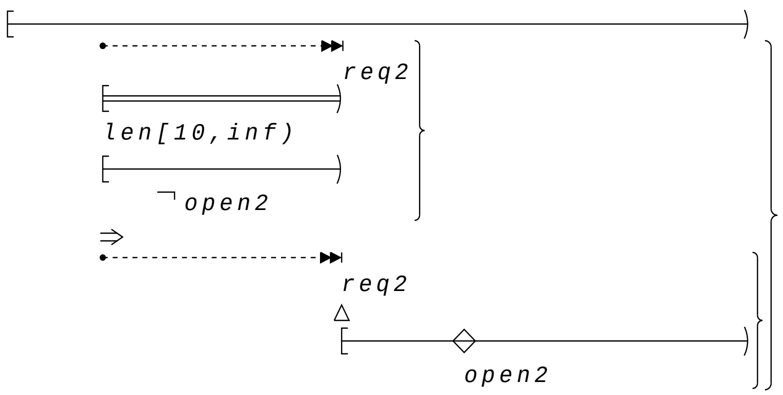
We have never found reason to use the full generality of GIL in numerous experiments that involved the specifications of a variety of different systems [5,6,11,12,16]. On the contrary, we have found that deeply nested intervals and searches to temporal formulas can result in counterintuitive formulas, due to the interaction of default valuations (when searches fail or intervals are empty). Moreover, it appears that most properties of interest for concurrent systems can be expressed very naturally without searching to temporal formulas.

Limiting the form of search targets to propositional formulas also expedites trace checking. The trace checking algorithm requires a formula be put into a disjunctive normal form, with negation pushed into subformulas and strong searches converted into eventualities. The algorithm is based on reducing formulas that are required to hold at a given state to simpler ones under the assumption that certain additional formulas hold at the next state. A reduction can introduce case-splits, corresponding to different ways in which a formula can be satisfied. Limiting the targets of searches to propositional formulas reduces the incidence of case-splitting. Intuitively, the restriction on search targets simplifies checking of interval formulas because it permits us to determine if a search succeeds at a state without looking into the future.

When a trace violates a specification, we would like to display the trace in a manner that helps the software engineer see where the fault occurred. Appropriately aligning subformulas of a negated specification with states in a trace often helps to show how the trace violates the specification. However, when a GIL formula involves searches to interval formulas, it can be very difficult to align the formula with the trace in a meaningful fashion. Thus, the restriction on search targets also facilitates the display of faults.

5 Testing Designs of Real-time Systems

GIL provides a special operator for expressing real-time properties of designs. The *len* operator bounds the duration of the remaining context, i.e. of the future within the enclosing interval. Thus, for example, the specification



asserts that, provided sufficient time elapses between the doors closing at floor 2 and a request for service at floor 2, the request is eventually serviced. In this example, the predicate $len(10, inf]$ bounds the duration of the strong interval from below by 10 time units. (An upper bound is specified by replacing *inf*, which can be viewed as denoting infinity, by a numeric value.)

To use specifications that involve the *len* operator as oracles, the events in traces must be time-stamped. Future research will extend the algorithm for checking timeless traces to timed traces. The extension will entail the introduction of ‘clocks’, modeled after the clocks described in [1, 18], which are started upon locating the left endpoint of an interval that contains a timing constraint. Each clock that is active in a state has an associated ‘clock condition’, which is checked at each state of the interval being timed. A clock is deactivated upon reaching the interval’s right endpoint.

6 Concluding Remarks

This paper has considered the use of GIL for specification and testing of concurrent and real-time systems. Past research produced a toolset for creating specifications in GIL and for reasoning about properties of systems that satisfy their specifications. Current research is addressing problems of using GIL specifications during the later phases of system development.

Our work to date has focused primarily on using GIL specifications as oracles during testing of executable designs. The paper presents an example in which GIL specifications for an Ada design are used to uncover faults during testing. It also shows how the graphical representation of formulas can be used to advantage in displaying a trace that violates a specification. To avoid overwhelming the designer with all the information in a trace, the designer might be permitted to indicate events to be traced and conditions to be displayed. The methods illustrated in the examples have yet to be automated.

Model checking is typically used for verifying that a concurrent system does not violate specifications expressed using temporal logics [2,4]. However, because model checking requires examination of all reachable system states, it runs into the state explosion problem.

Oracles for concurrent systems can be based on any formal specification language with a suitable decision procedure. The IDD debugger for distributed programs uses a subset of Interval Logic (IL) [25] for expressing synchronization constraints. IDD checks assertions at runtime, stopping execution if an assertion is violated [9]. The TSL Runtime System monitors executions of Ada tasking programs and checks them for conformance with specifications written in an event-based notation for specifying sequencing constraints [23,24]. Oracles for testing communication protocols can be produced from state transition diagrams [3]. A very powerful (but undecidable) extension of IL, called Real-Time Interval Logic (RIL), was designed for use in testing real-time systems [20]. Richardson, Aha and O'Malley argue that oracles based on multiparadigm and multilingual specifications are needed due to the multiparadigm nature of requirements [22]. They use RTL to specify temporal requirements of an elevator system and Z to describe the scheduling of services.

We believe that designers of concurrent systems will find GIL easier to use and understand than either IL or RTL due to the intuitive graphical representation of GIL formulas and the simpler, more uniform semantics of GIL. The graphical representation also provides us the opportunity to assist the designer in understanding faults discovered during testing. We are currently investigating tradeoffs between expressiveness and analyzability to find a compromise that permits common properties to be naturally expressed and also allows efficient trace checking.

Since concurrent systems are often distributed and are inherently nondeterministic, collecting traces and regenerating executions pose difficult research problems [13, 14, 17, 26]. We do not plan to develop new techniques in this area, but will evaluate current techniques to find one that is suitable for our purposes.

Specifications can also provide a basis for developing criteria for assessing the adequacy of test sets and for guiding the generation of test data [21]. The size of the test sets required when criteria are based on covering the concurrency states of a program grows exponentially with the number of tasks (processes) [27, 28]. Information in a specification can be used to generate more tractable test sets. For example, coverage criteria based on the structure of specifications might ensure that important synchronization patterns have been adequately tested, without examining all synchronization patterns. Potential faults detected during analysis of designs might also be used to identify specific synchronization patterns that warrant additional testing. Of particular interest in this regard is the work of Luo, Bochmann and Petrenko [15], in which they generate test sequences for concurrent programs and communication protocols from nondeterministic finite state machines.

References

[1] R. Alur and D. Dill. Automata for modelling real-time systems. In *Proc. 17th ICALP*, pages 322–335, 1990. LNCS 443.

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

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

[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, April 1986.

[5] Laura K. Dillon, George Kutty, P. Michael Melliar-Smith, Louise E. Moser, and Y. S. Ramakrishna. Visual specifications for temporal reasoning. *Journal of Visual Languages and Computing*, to appear.

[6] Laura K. Dillon, George Kutty, Louise E. Moser, P. Michael Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. Submitted.

[7] Laura K. Dillon, George Kutty, Louise E. Moser, P. Michael Melliar-Smith, and Y. S. Ramakrishna. Graphical specifications for concurrent software systems. In *Proc. 14th IEEE Inter. Conf. Software Engineering*, pages 213–224, Melbourne, May 1992.

[8] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. Elsevier Science, 1990.

[9] Paul K. Harter, Dennis M. Heimbigner, and Roger King. Idd: An interactive distributed debugger. In *IEEE Inter. Conf. Distributed Computer Systems*, pages 498–506, 1985.

[10] Richard A. Kemmerer, editor. *Proc. SIGSOFT '89 3rd ACM Symp. Testing, Analysis and Verification*, Key West, FL, December 1989. ACM Press. *ACM SIGSOFT Software Engineering Notes* 14(8).

[11] George Kutty, Y. S. Ramakrishna, Laura K. Dillon, Louise E. Moser, and P. Michael Melliar-Smith. Specification of a communication protocol in graphical interval logic. In *Proc. IEE Inter. Conf. Information Engineering*, pages 432–441, Singapore, December 1991.

[12] George Kutty, Y. S. Ramakrishna, Louise E. Moser, Laura K. Dillon, and P. Michael Melliar-Smith. A graphical interval logic toolset for verifying concurrent systems. In *Proc. 4th Conf. Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 138–153, Elounda, Greece, July 1993. Springer-Verlag.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *ACM Symp. Theory of Computing*, C-36(4):471–482, April 1987.

[15] G. Luo, G.V. Bochman, and A.Petrenko. Test selection based on communicating nondeterministic finite state machines using a generalized wp-method. *IEEE Trans. Software Engineering*, 20(2):149–162, February 1994.

[16] Louise E. Moser, Y. S. Ramakrishna, George Kutty, P. Michael Melliar-Smith, and Laura K. Dillon. A graphical environment for design of concurrent and realtime systems. Submitted.

[17] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. In *ACM SIGPLAN/SIGOPS Work. Parallel and Distributed Debugging*, pages 124–129, 1988.

[18] Y. S. Ramakrishna, Laura K. Dillon, Louise E. Moser, P. Michael Melliar-Smith, and George Kutty. A real-time interval logic and its decision procedure. In *Proc. 13th Conf. Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 173–192, Bombay, India, December 1993. Springer-Verlag.

[19] Y. S. Ramakrishna, P. Michael Melliar-Smith, Louise E. Moser, Laura K. Dillon, and George Kutty. Really visual temporal reasoning. In *Proc. 14th IEEE Real-Time Systems Symp.*, pages 262–273, Raleigh-Durham, NC, December 1993.

[20] Rami R. Razouk and Michael M. Gorlick. A real-time interval logic for reasoning about executions of real-time programs. In Kemmerer [10], pages 10–19. *ACM SIGSOFT Software Engineering Notes* 14(8).

[21] Debra Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification and testing of temporal properties of concurrent system designs. In Kemmerer [10], pages 86–96. *ACM SIGSOFT Software Engineering Notes* 14(8).

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

[23] David Rosenblum. Specifying concurrent system with TSL. *IEEE Software*, pages 52–61, May 1991.

[24] David S. Rosenblum and David C. Luckham. Testing the correctness of tasking supervisors with TSL specifications. In Kemmerer [10], pages 187–196.

[25] 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*, pages 173–186, Montreal, Canada, August 1983.

[26] K.C. Tai, R.H. Harver, and E.E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Trans. Software Engineering*, 17(1):45–63, January 1991.

[27] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Software Engineering*, 18(3):206–215, March 1992.

[28] Stewart N. Weiss. A formal framework for the study of concurrent program testing. In *Proc. SIGSOFT '88 2nd ACM Symp. Software Testing, Analysis, and Verification*, July 1988.