Efficient Specification-based Oracles for Critical Systems

Appeared in 1996 California Software Symposium

T. Owen O'Malley
Dept. of ICS
UC Irvine
Irvine, CA 92717 USA
(714) 824-4043
omalley@ics.uci.edu
http://www.ics.uci.edu/~omalley/

Debra J. Richardson
Dept. of ICS
UC Irvine
Irvine, CA 92717 USA
(714) 824-7353
djr@ics.uci.edu
http://www.ics.uci.edu/~djr/

Laura K. Dillon
Dept. of Computer Science
UC Santa Barbara
Santa Barbara, CA 93106 USA
(805) 893-3411
dillon@cs.ucsb.edu
http://www.cs.ucsb.edu/~dillon/

ABSTRACT

Effective testing of critical systems has been hampered by the lack of a cost-effective method for deciding the correctness of a program's behavior under test. Using formal specifications to describe the critical system properties and then checking test results against these specifications overcomes these problems. If these *test oracles*, which are mechanisms for determining whether a test passes or fails, are efficient, they can be combined with automatic test generation to cost-effectively automate the testing of large numbers of testcases that more adequately cover the system requirements and structure.

This paper presents a algorithm for automatically deriving efficient test oracles from Graphical Interval Logic (GIL) [5], which is a graphical temporal logic that is easier for non-experts to understand than many formal languages. To develop efficient test oracles from GIL, we convert the specifications into automata that can be checked in time linear in the length of the trace. Additionally, the automata can be checked incrementally to identify the failure when it occurs, which provides the opportunity to obtain useful information about the erroneous state of the system. We have implemented a tool that converts GIL specifications into automata based on the algorithms presented in this paper.

INTRODUCTION

Testing is a necessary activity to assess system behavior and quality. Unfortunately, it often yields very little trustworthy information about the correctness of a software system. One reason for this is the lack of an effective means for verifying that test executions are correct. The most common means of doing this is to manually check that test results are as expected or to specify the expected result for each test case and compare to the actual test result. Both means are extremely human-intensive and error-prone due to the repetitious and routine nature of the task, especially with a large test set developed to adequately cover a complex system. This is even further exacerbated for concurrent systems where the test set sizes must be larger for adequate coverage and the ordering and timing of events must also be examined. Testers could benefit greatly from technology that automates test result checking.

Specification-based test oracles are mechanisms derived from formal specifications for automatically checking that test execution behavior conforms to specified behavior [13]. Research about temporal logic specifications has focused on model checking [1, 4, 15], which determines whether it is possible to violate the specification on any execution, but require exploration of the entire state space and are inprecise. Specification-based oracles, on the other hand, only determine whether a particular execution violates the specification, but do not need to explore the entire state space and are precise. Temporal logic-based test oracles also permit verification of the ordering and timing of events in executions of concurrent systems. Dillon *et.al.* developed a method for constructing oracles from Graphical Interval Logic (GIL) specifications [7]. Here, we present a more general approach for deriving test oracles from GIL specifications.

GIL supports formal description of temporal properties of concurrent systems [5]. The visually intuitive nature of GIL specifications make them easier to develop and understand than traditional temporal logics, which are textual and dense. One might think of GIL specifications as stylized timing diagrams. Previous work on GIL produced a toolset that supports system specification and reasoning about properties guaranteed by the specification [11].

TAOS (Testing with Analysis and Oracle Support) is a comprehensive software testing environment that supports automatic test result checking with test oracles [12]. The support for test oracles is extremely general; in essence, it can be tailored to support oracles derived from any specification language as well as more basic types of oracles. TAOS currently supports a simple algorithm that checks test execution traces against GIL

This work is sponsored in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency, under the Contract Number F30602-94-C-0218 and by Hughes Aircraft Company and the University of California under MICRO Grant (Hughes UCM-20880, UC 95-125). The content does not necessarily reflect the position or policy of the U.S. Government, Hughes Aircraft Company or the University of California, and no official endorsement should be inferred.

specifications. We are currently implementing a tool supporting the algorithm discussed in this paper, which will provide more comprehensive support for GIL-based test oracles within the TAOS environment.

To develop efficient test oracles from GIL, we convert the specifications into deterministic finite automata (DFA) that can be checked in time linear in the length of the trace. Additionally, the DFA can be checked incrementally so that a failure is identified when it occurs instead of at the end of the test when execution is complete. This enables testing of continuously running systems and provides the opportunity to obtain additional information about the erroneous state of the system that might be lost if the failure is not determined in a timely manner. Our approach to translating from GIL to DFA is based on using an intermediate hierarchal non-deterministic finite automata (NFA) form that is similar to StateCharts [9]. Dillon's previous translation method [7] used rewrite rules to produce DFAs, but could only handle specifications of a restricted form determined by the form of the rules used. Our new approach overcomes this limitation by using the intermediate NFA form.

Specification-based oracles can be thought of as consisting of a general purpose *oracle procedure* for a type of oracle and *oracle information* specifying desired behavior for specific test cases or a specific component under test [13]. The general approach for specification-based oracles is to convert a formal specification of required behavior into an internal representation (the oracle information) and then develop a checker for that internal representation (the oracle procedure). The DFA result checker works for all GIL specifications as well as for other specification languages that can be converted into DFAs, such as other temporal logics and other state-based specification-languages.

This paper briefly describes the GIL specification language; outlines our method for converting GIL specifications into automata, including description of the NFA formalism; and presents an example of the use of our approach illustrating its benefits. The use of the DFA as oracles is then discussed. Finally, we conclude with our plans for future work.

DESCRIPTION OF GIL

GIL [5] is a graphical interval logic that enables specifying the desired temporal properties of a system. This paper presents a brief description of GIL as necessary to clarify the transformation from GIL to an effective oracle procedure. Because GIL uses a graphical representation, it assists non-experts in understanding the specifications. The graphical editor and proof tools developed at UCSB form an effective specification language toolset that is easy to use and powerful [11].

GIL axioms are given over a global interval, which is displayed as a line across the top of the axiom, which represents the entire state trace. By default, GIL sub-expressions must hold in just the first state of the interval and so the sub-expression is placed on the left side of the interval, as shown in figure 1. Also supported are the logical operators *and* ($\wedge$ or simply vertical alignment), *or* ($\vee$), *not* ($\neg$), *implies* ($\Rightarrow$) and *equivalence* ($\Leftrightarrow$) and braces ($\}$) to display grouping. Some of these operators are shown in figure 2. GIL also includes the standard temporal operators always and eventually. Always is represented as putting the sub-expression in the center of the interval showing that it is true in each state throughout the interval, as in figure 3. Eventually means that the sub-expression becomes true in at least one of the states in the interval and is represented with the sub-expression in the center of the interval with a diamond above it, as in figure 4.

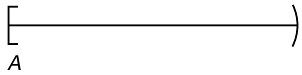


Figure 1: A simple expression in GIL

Figure 2: Logical operators in GIL

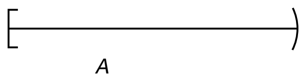


Figure 3: Always operator in GIL

In interval logics, such as GIL, the primary construct is an interval constructor, as shown in figure 5. In the example, the sub-expression C must hold between the first state where A is true to the first following state where B is true. A single line representing the interval means that if A and B become true simultaneously, the expression is true vacuously. Alternatively, a double line, as shown in figure 6, means that an empty interval does not satisfy the specification.

Intervals can also be built from more complex searches involving multiple search targets for either the start or the end of the interval, as in figure 7. In the figure, the start of the interval is the state where sub-expression B first becomes true after A first becomes true. Correspondingly, the end of the interval is the first state where

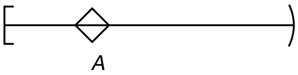


Figure 4: Eventually operator in GIL

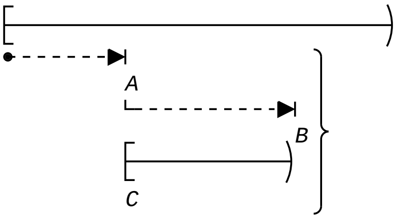


Figure 5: Interval construction in GIL

D becomes true, after C becomes true, after the start of the interval. Search targets may also be marked as forced or unforced searches. Forced searches are targets that must be found for the expression to succeed and are represented with double arrowheads, as shown in figure 8.

GIL also includes a shorthand for an expression that must hold after a search is satisfied. The point operator shown in figure 9 states that B is true after A becomes true.

CONVERSION TO AUTOMATA

Effective use of GIL specifications as oracles requires converting them into a form that can be checked against an execution trace efficiently. Our goal therefore is to find a representation for GIL that will be significantly faster to check against a trace, but which may be computationally expensive to generate. Our transformation is done in two phases: GIL is converted first to a powerful non-deterministic finite automata (NFA) formalism and secondly it is converted to a standard deterministic finite automata (DFA). Through out this discussion, we use the term *state* to refer to states in the trace and nodes to refer locations in the automata.

NFA formalism

Our NFA formalism is based on StateCharts [9]. StateCharts is a hierarchical state machine representation

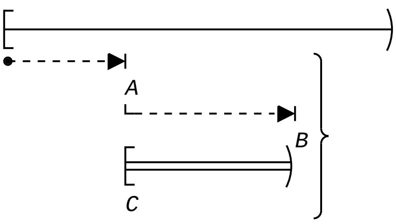


Figure 6: Strong interval construction in GIL

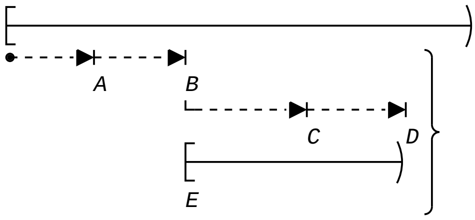


Figure 7: Two target search interval construction in GIL

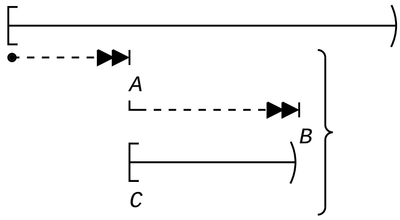


Figure 8: Forced search interval construction in GIL

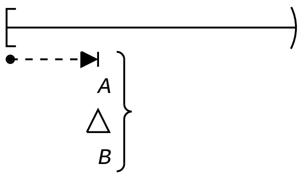


Figure 9: Point operator in GIL

that is typically used to describe reactive systems. Hierarchical state machines extend traditional automata by allowing sub-automata as nodes. *Orthogonal products* are automata composed from a set of children automata that are interpreted in parallel. Since the children are interpreted in parallel, each child evaluates each trace state. Thus, the state space of the orthogonal product is the cross product of each of the children automata. Orthogonal products are represented as sub-automata with dashed lines between the children. StateCharts also have predicates, such as *in 9*, which are true if the automata is in node 9, that enable the parts of orthogonal products to communicate.

We added three extensions to StateCharts to form our NFA. These extensions allow syntactic transformations of GIL to NFA. The extensions are:

- Conditions on transitions from the NFA nodes are not required to be mutually exclusive. Since more than one condition can be satisfied, the automata are non-deterministic. If any path through the automata satisfies the trace, it must be accepted.

- Because we are analyzing program traces instead of modeling a system that runs indefinitely, termination conditions must be considered. Some of the nodes are marked as final nodes and the automata accept a trace if and only if the automata is in a final node during the final state of the trace. We use the standard representation of double circles for final nodes. Orthogonal products accept a trace only if all of their children are in final nodes.

- We need additional control over the trace state so divide the automata edges into two kinds: delta and non-delta edges. Delta edges check their condition against the "current" trace state and then go on to the next trace state. Non-delta edges check their condition against the current trace state and do not move the trace to the next state. Therefore, two non-delta edges, with labels A and B respectively, put end to end are equivalent to a single non-delta edge with $A \wedge B$. Non-delta edges are a generalization of epsilon edges from automata theory.

Converting GIL specifications into NFA

GIL specifications[1] can be converted syntactically into NFA. We will present the transformations of the various operators, but note that all of the transformations may be nested to arbitrary depths. This section uses the figures from the section describing GIL to show how the GIL expressions are converted into NFA.

[1] currently with a restriction against having temporal properties as search goals

To simplify the transformation process, we transform implication and equivalence to ands, ors and nots and push all of the nots down to lowest possible level. Because all of negations of the GIL temporal operators can also be expressed in GIL, all of the negations end up over the non-temporal state predicates.

The basis case is any non-temporal state predicate as in figure 1, which is transformed to a transition between non-final and final nodes, as shown in figure 10.

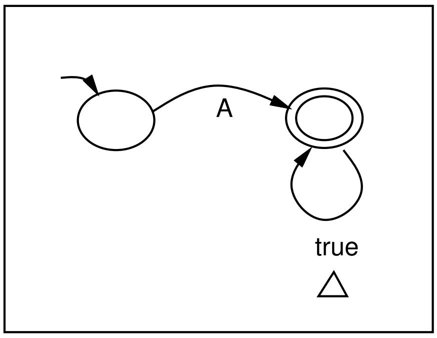


Figure 10: Non-temporal predicate as a NFA (from figure 1)

The logical operators *or* and *and*, as in figure 2, are converted into non-deterministic choice or orthogonal products, respectively, provided they have temporal subexpressions. Figure 11 is the resulting automata.

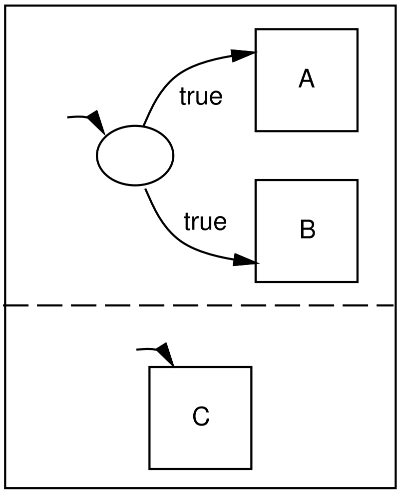


Figure 11: Logical operators as a NFA (from figure 2)

Always A, as shown in figure 3, requires that the predicate A is true in the current state and that always A is true in each following state. Since orthogonal products require all of their children to accept, A must be true for each state for the NFA in figure 12 to accept the trace.

Eventually A, shown in figure 4, is modeled in figure 13 by making a non-deterministic choice between satisfying

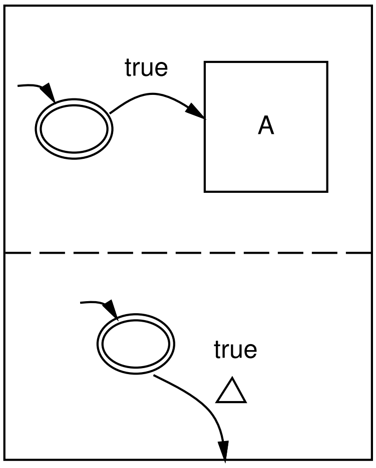


Figure 12: Always as a NFA (from figure 3)

the predicate A or going to the next state. Since NFAs always accept if any path accepts, if A holds in any state, the eventually NFA will accept the trace.

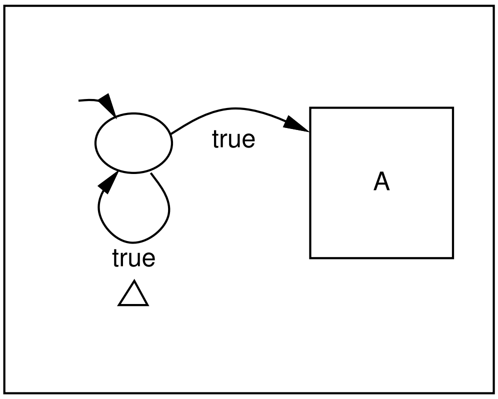


Figure 13: Eventually as a NFA (from figure 4)

The simplest search in GIL is A point B, shown in figure 9, that finds the first state where A is true and requires B be true. Its translation is shown in figure 14.

Interval construction is a complex transformation including the sub-expression and both of the searches for the start and end of the interval. In figure 5 the interval is between A and B with the sub-expression C. The top level automata in figure 15 is composed of four places: the start node represents the state before A is found, the sub-automata checks the states during the interval and the nodes on the left represent success or failure of the expression. The top of the sub-automata checks for the end of the interval while the bottom is marked C' because it is the automata for the C with all of the nodes marked as final. The condition C-Final is a node predicate that represents when C accepts the trace.

Forced searches, as in figure 8, change the terminal

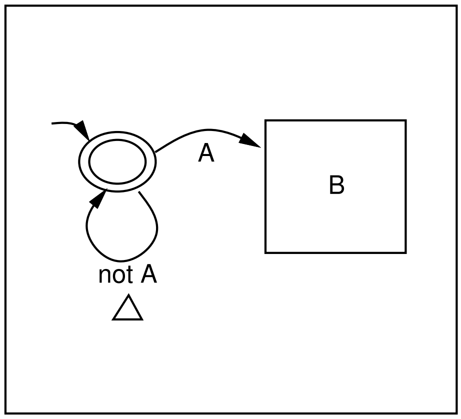


Figure 14: Point as a NFA (from figure 9)

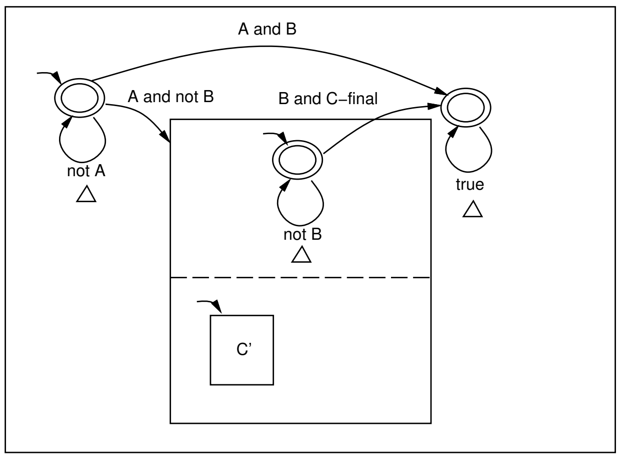


Figure 15: Simple interval as a NFA (from figure 5)

nodes in the search automata to non-terminals, as shown in figure 16, because the expression is not satisfied if they are not found.

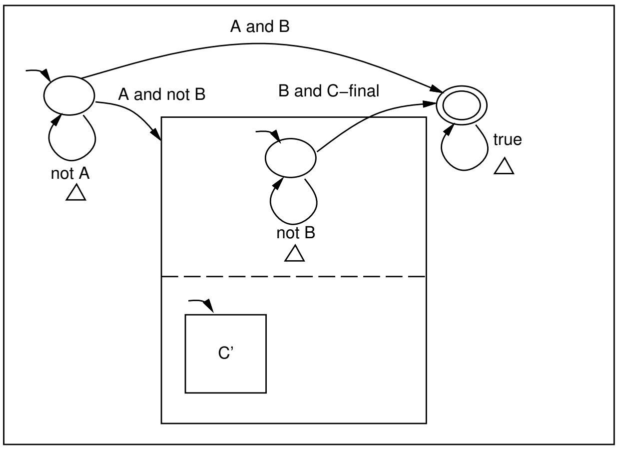


Figure 16: Forced searches as a NFA (from figure 8)

The strong interval in figure 6 prohibits the empty interval, so the equivalent NFA (figure 17) does not have the transition from the start node to the final node.

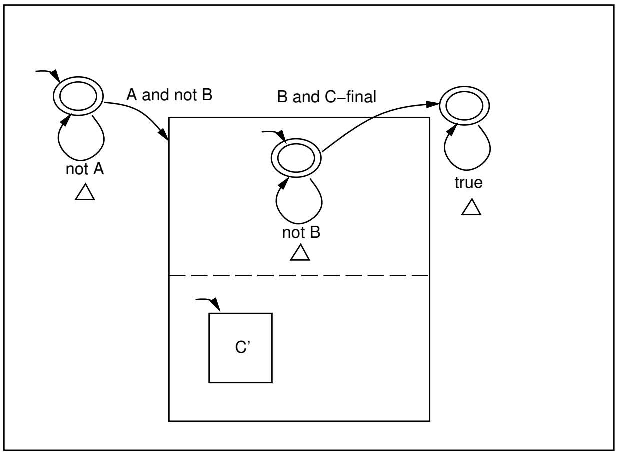


Figure 17: Strong interval as a NFA (from figure 6)

GIL searches are not limited to single goals, but may have an arbitrary number of goals, as in figure 7. Each goal is represented as a node, with a self edge with the negation of the goal and a continuation edge that requires the goal be true. The short-circuit edge in a multiple goal search is the conjunction of all the goals that close the interval. An example transformation with two goal searches is shown in figure 18.

By composing these transformations, all GIL specifications can be transformed to NFA.

Conversion to a DFA

These extended NFA may be converted into standard DFA through a transformation that is similar to the standard one between NFA and DFA, where nodes in the DFA correspond to sets of potentially reachable nodes in the NFA. Each of the NFA extensions require modifications to the transformations: Next, we will discuss these modifications.

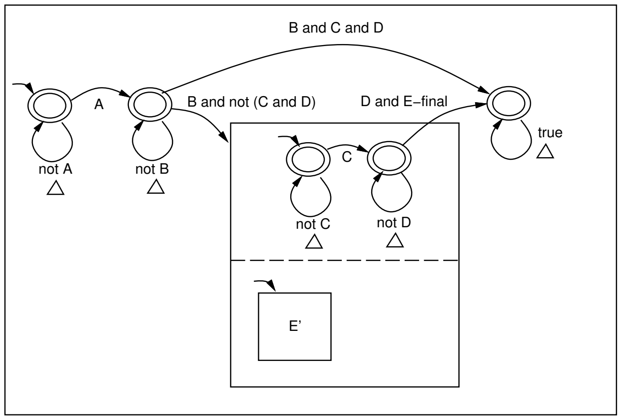


Figure 18: Multiple search conditions in interval construction as a NFA (from figure 7)

Orthogonal products replace a single node with a cross product consisting of one node from each child. For example, in a orthogonal product with two children one with node 1 and the other with node 9, a valid state would be `(1,9)`. Thus, the set of potential states can contain node cross products as well as nodes.

Recursive loops in the automata, which are created from an always operator, as shown in figure 12, create a set of iteration states. When the automata loops back to the parent automata, it starts a new iteration. Each iteration is independent of the others and they must all accept the trace. The size of the iteration set is bounded because there are a bounded number of distinct iterations. Identical iterations do not need to be considered independently. For example, we represent an iteration set with the members `(1,9)` and `(2,9)` as `Itr{(1,9), (2,9)}`.

Since the alphabet of our automata is boolean expressions, they are not necessarily disjoint. To ensure that the DFA is deterministic, we must check each pair of edges A and B and generate all of the combinations where at least one edge is enabled: $A \wedge B$, $A \wedge \neg B$ and $\neg A \wedge B$.

The distinction between delta transitions and non-delta transitions is removed by "collapsing" a sequence of non-delta transitions with a final delta transition into a single transition in the new automata. As the edges are collapsed, the conditions on the edges should be checked for consistency to ensure that we only explore feasible transitions. To generate the potential next states, we choose an arbitrary node in the state and take all the feasible transitions by replacing the edge's source node with the destination node. If a transition is a delta edge, that node is done. We repeat this process until every node in the state is done.

Using the DFA as an oracle

To use the generated DFA as an oracle, we instrument the program to generate a trace of the specification states. Each specification state determines the next state in the DFA by comparing the state to the condition on each of the DFA edges. If at some point there are no valid edges or the trace ends in a non-final node, a failure has occurred.

The generated trace may either be stored for later analysis or analyzed concurrently with test execution to check the specification as the program is executing. Generating the trace allows the tester to examine the complete history of the program, but also requires large amounts of memory or disk space. Checking the oracle during execution also allows the testing system to capture state information from the program that would otherwise be lost, including the recent specification trace information. Incremental oracles are critical for continually executing or concurrent systems, because they typically generate large amounts of trace information.

Other potential uses of the DFAs include static dataflow analysis to determine which actions the program may or may not perform [8]; structural specification-based test selection and measurement [3]; combining static and dynamic analysis [14]; and incorporation of the oracle into a debugging system [10] or a robust self-checking program [2].

EXAMPLE

To illustrate our algorithm, we choose the GIL axiom DoesPump, as shown in figure 19, from the gas station problem presented in [7]. DoesPump requires that if a user pays, she will eventually pump gas and finally get her change back.

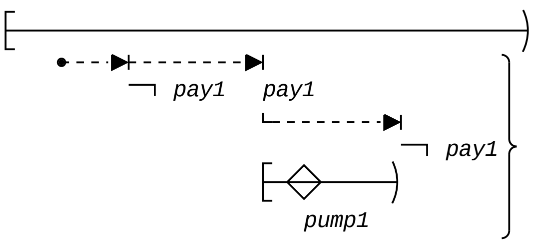


Figure 19: GIL axiom does pump

We transform DoesPump to an NFA, shown in figure 20, by starting at the top level always operator and working inwards, applying a straightforward transformation at each step. Thus, the outermost automata is identical to the always NFA in figure 12. Notice that the automata containing nodes 4 through 7 represents the interval and its exit condition is $\neg pay1 \wedge in9$, which is the conjunction of the search goal that ends the interval and the node predicate that represents *eventually pump*1.

$$true \Rightarrow \texttt{Itr\{(1,9)\}}$$

$$\Downarrow \text{(node 1)}$$

$$true \Rightarrow \texttt{Itr\{(2,9)\}}$$

$$\Downarrow \text{(node 2)}$$

$$\begin{array}{c} pay1 \Rightarrow \texttt{Itr\{(2d,9)\}} \\ \neg pay1 \Rightarrow \texttt{Itr\{(3,9)\}} \end{array}$$

$$\Downarrow \text{(node 3)}$$

$$\begin{array}{c} pay1 \Rightarrow \texttt{Itr\{(2d,9)\}} \\ \neg pay1 \Rightarrow \texttt{Itr\{(3d,9)\}} \end{array}$$

$$\Downarrow \text{(node 9)}$$

$$\begin{array}{c} pay1 \Rightarrow \texttt{Itr\{(2d,*), (1d,9d)\}} \\ \neg pay1 \Rightarrow \texttt{Itr\{(3d,*), (1d,9d)\}} \end{array}$$

Figure 21: Extending from the initial state of figure 20

Next, we convert the NFA to a DFA by exploring the potential states. Derivation of transitions from the initial state is shown in figure 21. Nodes that are done are marked with a 'd' and nodes that have recursed and have started a new iteration are represented with '*'. As the figure shows, the initial DFA node has two transitions: $pay1$ leads to `Itr{(2,*), (1,9)}` and $\neg pay1$ leads to `Itr{(3,*), (1,9)}`. The complete DFA is given in figure 22.

Using DoesPump's DFA from figure 22, if the program generated the trace in figure 23, the DFA-based oracle would detect a failure in State 3 because there are no enabled transitions from node D. In this case, the failure is caused by the end of the interval without finding Pump1.[2]

CONCLUSION

This paper has presented a method for converting GIL specifications to DFA for arbitrary GIL specifications, with the restriction of non-temporal search targets. This restriction can be removed if the semantics of the NFA formalism is changed so that the trace state is restored when a sub-automata is exited. Then any predicate can be represented as an edge to an automata that checks the predicate and another edge out.

GIL provides an effective and readable language for spe-

[2]Recall that GIL intervals do not include the state where the final goal becomes true.

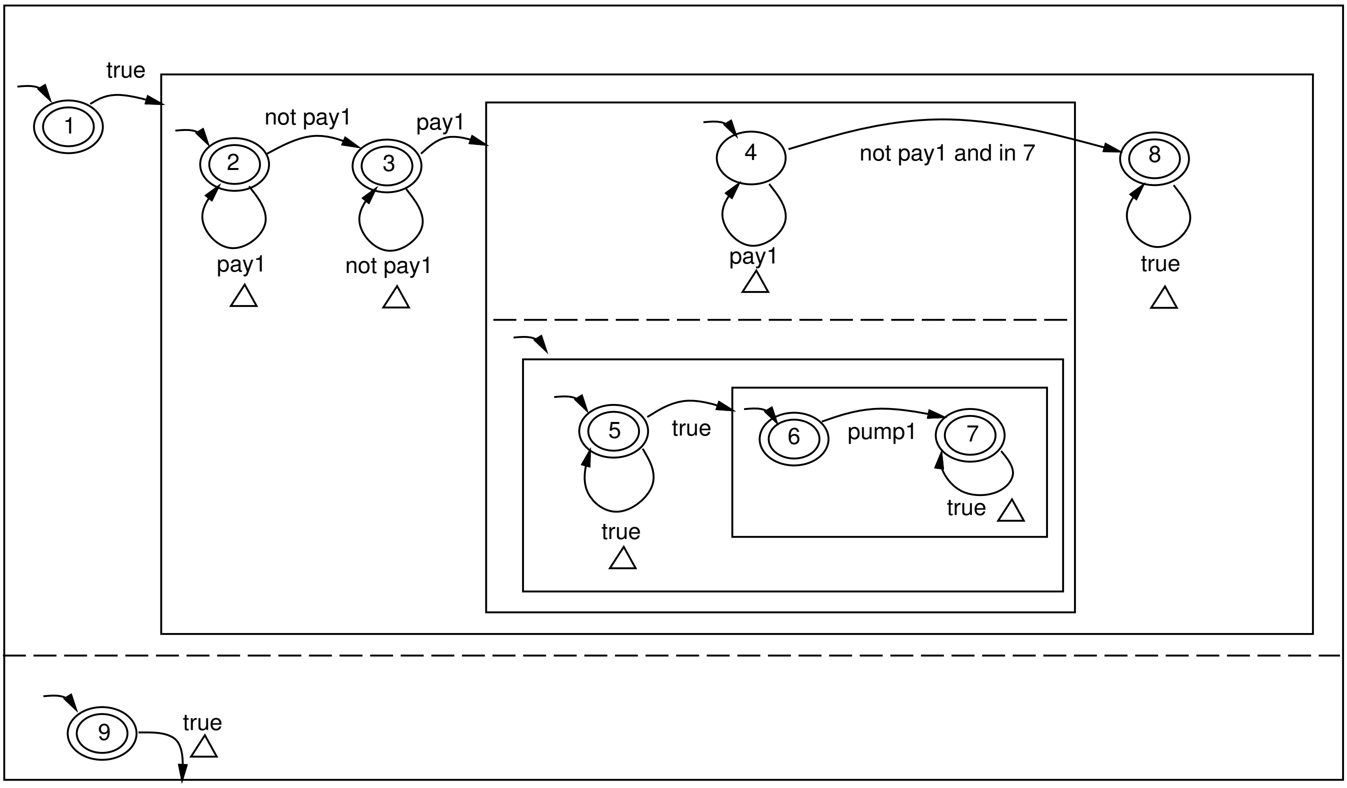
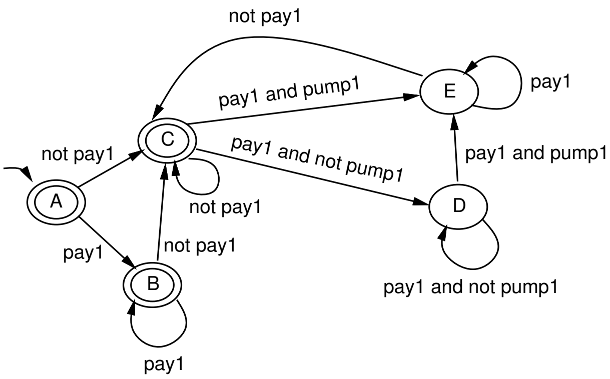


Figure 20: Does pump NFA (from figure 19)



DFA node	NFA node expression
A	`Itr{(1,9)}`
B	`Itr{(2,*), (1,9)}`
C	`Itr{(3,*), (1,9)}`
D	`Itr{((4,5),*), (2,*), (1,9)}`
E	`Itr{((4,7),*), (2,*), (1,9)}`

Figure 22: Does pump DFA (from figure 19 and figure 20)

State	Pay1	Pump1	DFA Node
0			A
1	false	false	C
2	true	false	D
3	false	true	failure
4	false	false	

Figure 23: Example DoesPump trace

cifying system properties. Converting GIL into DFA generates efficient and effective test oracles that can be checked with minimal overhead, thereby increasing the confidence in the software quality gained via the testing process.

We have developed tool support for the GIL → NFA → DFA conversion presented in the paper as well as a general-purpose test result checker for DFAs. The software is available free of charge from `http://www.ics.uci.edu/~softtest/gil.html`. This technology will be integrated into the TAOS software testing environment, which also provides support for test artifact development and maintenance, automatic test data generation, test adequacy criteria definition and coverage measurement, and test process management [12]. TAOS is available from `http://www.ics.uci.edu/~softtest/taos.html`.

Other current work on GIL includes a method for generalizing the process of using re-write rules to create automata to allow arbitrary re-write rules [6]. This work would simplify the process of experimenting with different re-write rules and/or changes to the semantics of the language.

REFERENCES

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

[2] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *Proc. Symposium of Foundations of Computer Science*, 1994.

[3] J. Chang, D. J. Richardson, and S. Sankar. Structural specification-based testing with ADL. In *Proc. International Symposium on Software Testing and Analysis*, pages 62–79, San Diego, CA, January 1996.

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

[5] L. K. Dillon, G. Kutty, L. E. Moser, and P. M. Melliar-Smith. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[6] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal specifications. Submitted.

[7] L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *Proc. 2nd ACM SIGSOFT Symposium Foundations of Software Engineering*, pages 140–153, New Orleans, December 1994.

[8] M. Dwyer and L. Clarke. A flexible architecture for bulding data flow analyzers. In *Proc. 18th International Conference on Software Engineering*, Berlin, Germany, March 96.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[10] P. K. Harter, D. M. Heimbigner, and R. King. IDD: An interactive debugger. In *IEEE International Conference on Distributed Computer Systems*, pages 498–506, 1985.

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

[12] D. J. Richardson. Taos: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 138–153, Seattle, August 1994. ACM Press.

[13] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.

[14] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

[15] M. Young, R. N. Taylor, K. Forester, and D. A. Brodbeck. Integrated concurrency analysis in a software development environment. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 200–209, Key West, Florida, December 1989. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).