Automata Representations for Concurrent METATEM

Adam Kellett and Michael Fisher

Department of Computing
Manchester Metropolitan University
Manchester M1 5GD
United Kingdom

EMAIL: {A.Kellett,M.Fisher}@doc.mmu.ac.uk

Abstract

Concurrent METATEM *is a language based on the execution of temporal logic formulae. The current implementations are based upon direct interpretation, yet are too slow for large applications. We present an approach to implementing the language by representing temporal formulae as finite-state automata. To combat the problems associated with constructing a single automaton containing all possible models of a formula, we partition the representation into three closely coupled automata. This provides structures small enough to be effectively used for larger applications, while still allowing increased performance over the direct interpretation of temporal formulae.*

CLASSIFICATION: *executable temporal logics, compilation mechanisms, automata-based representation.*

1 Introduction

Concurrent METATEM is a programming language for reactive systems [14] that has been shown to be particularly useful in representing and developing multi-agent systems [8]. It is based on the combination of two complementary elements: the direct execution of temporal logic specifications providing the behaviour of an individual object [9]; and a concurrent operational model in which such objects execute asynchronously, communicate via broadcast message-passing, and are organised using a grouping mechanism [7].

In this paper we consider the implementation of Concurrent METATEM objects situated in an unconstrained environment. In particular, we investigate alternative representations for the behaviour of an individual object based upon finite-state automata. The temporal specification of an object's behaviour directly corresponds to a formula in a suitable temporal logic and, since there is a close correspondence between temporal formulae and automata [18], it is not surprising that an automaton representation for an object's behaviour can be generated. However, as the size of this automaton is exponential in the length of the original temporal specification, it has been considered that the automata representation is likely to be too large for practical use, even when improved encoding techniques are employed [4]. Consequently, only the direct interpretation of the temporal specification has been so far developed as an implementation technique [10]. While the interpreted Concurrent METATEM system is appropriate for small and medium-size examples, it often becomes unacceptably inefficient for large-scale applications, such as those involving multi-agent systems. Thus, we here investigate improved automata representations with a view to utilising these in a compiler for Concurrent METATEM.

The structure of this paper is as follows. In §2, we provide a brief review of temporal logic and its relationship to finite automata, followed, in §3, by an introduction to the Concurrent METATEM language. In §4, the basic approach of constructing an automaton that represents a specific temporal formula is considered, together with a number of improvements designed to reduce both space and execution time. In §5, the relative merits of all these representations are considered. Finally, in §6, we summarize the contributions of the paper and outline future work.

2 Temporal Logic

Temporal logic can be seen as classical logic extended with various modalities representing temporal aspects of logical formulae [5]. The propositional temporal logic we use (called PTL) is based on a linear, discrete model of time. Thus, time is modelled as an infinite sequence of discrete states, with an identified starting point, called 'the beginning of time'. Classical formulae are used to represent constraints *within* states, while temporal formulae represent constraints *between* states. As formulae are interpreted at particular states in a sequence, operators which refer to both the past and future are required. Examples of

0-8186-7937-9/97 $10.00

such operators are as follows[1]: $\Diamond\varphi$ is satisfied now if φ is satisfied *sometime* in the future (often termed *eventualities*); $\Box\varphi$ is satisfied now if φ is satisfied *always* in the future; $\varphi\,\mathcal{U}\,\psi$ is satisfied now if φ is satisfied from now *until* a future moment when ψ is satisfied; $\bigcirc\varphi$ is satisfied now if φ is satisfied at the next moment in time; $\varphi\,\mathcal{S}\,\psi$ is satisfied now if ψ was satisfied in the past and φ was satisfied from that moment until (but not including) the present moment; $\bullet\varphi$ is satisfied if there was a previous moment in time and, at the moment, φ was satisfied; **start** is only satisfied at the beginning of time. Finally, '$\blacklozenge$' is the past-time analogue of '$\Diamond$', while '$\blacksquare$', is the past-time analogue of '$\Box$'.

2.1 Separated Normal Form

As an object's behaviour is represented by a temporal formula, we can transform this formula into Separated Normal Form (SNF) [13]. This not only removes the majority of the temporal operators, but also translates the formula into a set of *rules* suitable for direct execution. Each of these rules is of one of the following forms.

$$\textbf{start} \Rightarrow \bigvee_{j=1}^{r} m_j \qquad \text{(an } \textit{initial} \Box\text{-rule)}$$

$$\bullet \bigwedge_{i=1}^{q} k_i \Rightarrow \bigvee_{j=1}^{r} m_j \qquad \text{(a } \textit{global} \Box\text{-rule)}$$

$$\textbf{start} \Rightarrow \Diamond l \qquad \text{(an } \textit{initial} \Diamond\text{-rule)}$$

$$\bullet \bigwedge_{i=1}^{q} k_i \Rightarrow \Diamond l \qquad \text{(a } \textit{global} \Diamond\text{-rule)}$$

where each k_i, m_j or l is a literal. Note that the left-hand side of each *initial* rule is a constraint only on the *first* state, while the left-hand side of each *global* rule represents a constraint upon the previous state. The right-hand side of each $\Box$-rule is simply a disjunction of literals referring to the current state, while the right-hand side of each $\Diamond$-rule is a single eventuality (i.e., '$\Diamond$' applied to a literal).

2.2 Automata and Temporal Logic

Automata are structures based on *states* and *transitions*. Finite-state automata have been shown to have a close correspondence to formulae of temporal logic [18]. In particular, it has been shown that any PTL formula can be modeled by a ω-automaton [16]. Each state of such an automaton represents a possible interpretation for a temporal state, while transitions between states determine valid successors. An ω-automaton consists of

$$(S, s_0, \Sigma, \Delta, F)$$

Here, the alphabet, Σ, is just the set of propositions, L_p, used in a formula. Each state, $s \in S$, represents the valuation for a temporal state, i.e. an assignment of truth values to each $p \in \Sigma$. The initial state, s_0, is distinguished as the beginning of time, specifying the first state of any model. The set of transitions, Δ, is a subset of $S \times S$, linking states to produce full sequences. The set of final states, F, represents states which must be visited infinitely often, effectively those in which eventualities can be satisfied.

The finite model property of PTL allows an automaton representing a PTL formula to model every potential temporal sequence. PTL formulae can therefore be fully specified by an automaton. Unfortunately, not only is the construction of such an automaton time consuming, but it is generally much larger than the original formula.

3 Concurrent METATEM

The motivation for the development of Concurrent METATEM [14] has been provided from many areas. Being based upon executable logic, it can be utilised as part of the formal specification and prototyping of reactive systems. In addition, as it uses *temporal*, rather than classical, logic the language provides a high-level programming notation in which the dynamic attributes of individual components can be concisely represented [1]. This, together with its use of a novel model of concurrent computation, ensures that it has a range of applications in distributed and concurrent systems [7].

Concurrent METATEM is an object-based programming language comprising two distinct aspects:

1. the fundamental behaviour of a single object is represented as a temporal formula and animation of this behaviour is achieved through the direct execution of the formula [9];

2. objects are placed within an operational framework providing both asynchronous concurrency and broadcast message-passing.

While these aspects are, to a large extent, independent, the use of *broadcast* communication provides a natural link between them as it represents both a flexible communication model for concurrent objects [3] and a natural interpretation of distributed deduction [15]. Thus, these features together provide an coherent and consistent programming model within which a variety of reactive systems can be represented and implemented.

3.1 Objects

The basic elements of Concurrent METATEM are objects. These are considered to be encapsulated entities, executing independently, and having complete control over their own internal behaviour. There are two elements to

[1] Due to lack of space, the full syntax and semantics of this temporal logic is omitted — see, for example, [5] for details.

each object: its *interface definition* and its *internal definition*. The definition of which messages an object recognises, together with a definition of the messages that an object may itself produce, is provided by the interface definition for that particular object. The internal definition of each object is provided by a temporal specification.

In order to animate the behaviour of an object, we choose to execute its temporal specification directly [9]. Execution of a temporal formula corresponds to the construction of a model for that formula and, in order to execute a set of SNF rules representing the behaviour of a Concurrent METATEM object, we utilise the *imperative future* [2] approach. This evaluates the SNF rules at every moment in time, using information about the history of the object in order to constrain future execution. Thus, a *forward-chaining* process is employed to produce a model for a formula; the underlying (sequential) METATEM language [1] exactly follows this approach.

The operator used to represent the basic temporal indeterminacy within the SNF rules is the *sometime* operator, '$\Diamond$'. When $\Diamond\varphi$ is executed, the system must try to ensure that φ *eventually* becomes true. As such eventualities might not be able to be satisfied immediately, we must keep a record of the unsatisfied eventualities, retrying them as execution proceeds. It should be noted that the use of temporal logic as the basis for the computation rules gives an extra level of expressive power over the corresponding classical logics. In particular, operators such as '$\Diamond$' give us the opportunity to specify future-time (temporal) indeterminacy. Transformation to SNF allows us to capture these expressive capabilities concisely.

As an example of a simple set of rules which form a fragment of an object's description, consider the following.

$$\begin{aligned} \textbf{start} &\Rightarrow \neg \texttt{moving} \\ \bullet\texttt{go} &\Rightarrow \Diamond\texttt{moving} \\ \bullet(\texttt{moving} \wedge \texttt{go}) &\Rightarrow \texttt{overheat} \vee \texttt{fuel} \end{aligned}$$

Here, we see that `moving` is false at the start of execution and, whenever `go` is true in the last moment in time, a commitment to eventually make `moving` true is made. Similarly, whenever both `go` and `moving` are true in the last moment in time, then either `overheat` or `fuel` must be made true.

3.2 Concurrency and Communication

It is fundamental to our approach that all objects are (potentially) concurrently active. In particular, they may be asynchronously executing. Each object, in executing its temporal formula, independently constructs its own temporal sequence. Within Concurrent METATEM, a mechanism is provided for communication between separate objects which simply consists of partitioning each object's propositions into those controlled by the object and those controlled by its environment. To fit in with this logical view of communication, whilst also providing a flexible and powerful message-passing mechanism, *broadcast* message-passing is used to pass information between objects. Here, when an object sends a message it does not send it to a specified *destination*, it merely sends it to its environment where it can be received by *all* other objects. Although broadcast is the basic mechanism, both multicast and point-to-point message-passing can be defined on top of this [7]. Finally, the default behaviour for a message is that if it is broadcast, then it will *eventually* be received at all possible receivers. Also note that, by default, the order of messages is not preserved, though such a constraint can be added, if required.

3.3 Applications and Implementation

The combination of executable temporal logic, asynchronous message-passing and broadcast communication provides a powerful and flexible basis for the development of reactive systems. Concurrent METATEM is being utilised in the development of a range of applications in areas from distributed artificial intelligence [11], concurrent theorem-proving [15], agent societies [12], and transport systems [6]. A survey of some of the potential applications of the language is given in [7],

The current implementation is based upon the direct interpretation of Concurrent METATEM rules. It is written in C++ and incorporates many of the features of interpreters developed for sequential METATEM [10].

4 Compilation of Concurrent METATEM

The purpose of the compiler is to produce a representation for the compact and efficient execution of Concurrent METATEM programs. Having a close relationship to temporal logic, finite-state automata are a natural choice on which to base these structures [19, 18]. The product of a Concurrent METATEM execution is a model of the temporal formula corresponding to the program. By representing a program as an automaton all models, and therefore all possible execution sequences, are determined during compilation. The effect is to reduce execution to a process of traversing the automata, as directed by environment interaction.

In this section we introduce the framework for the compiled Concurrent METATEM system. The generation of automata representing both propositional and first-order programs is described, including a number of approaches designed to reduce the space requirements of the compiled representation. In §5, we examine the effect of our model on both the compilation and execution behaviour.

4.1 Compilation Framework

For our approach, the METATEM language consists of a compiler, translating from program formulae to an

automata-based representation, and an execution mechanism able to interpret these structures. The compiler guarantees that for any model of the program formula, there exists a path in the automaton which represents this model. The execution mechanism takes a compiled representation as input and produces a single infinite execution sequence, by traversing the automaton and producing a temporal state valuation for each moment in time. Execution is directed by environmental interaction and heuristics for the satisfaction of eventualities [10].

Although ω-automata provide a mechanism for representing temporal formulae, the size of the structures produced can quickly become unmanageable. As the formulae of METATEM programs can be complex, the specification of many applications would be prohibited by storage requirements. For this reason we have developed an approach to reduce the size of the representation produced. This is based on the removal of some subset of the program formula from the interpretation of automaton states. Based on the set of propositions L_p, three subsets are defined, and each is treated separately by the compilation process:

L_e: The set of environment propositions representing messages received by an object.

L_{ev}: The set of propositions specified as eventualities.

L_a: The set of propositions represented by $L_p - (L_e \cup L_{ev})$.

The constraints on each subset are represented by an automaton. The internal evaluation of a program is represented by a cyclic automaton generated for the set L_a. The automata for L_e and L_{ev} are used to integrate the satisfaction of any members of these sets with the internal automaton. This approach reduces the maximum number of states used by the compiled representation from

$$2^{\overline{L_e \cup L_a \cup L_{ev}}} \quad \text{to} \quad 2^{\overline{L_a}} + 2^{\overline{L_e}} + 2^{\overline{L_{ev}}}$$

Thus, much of the work described in this paper ensures that sufficient structure is present within the separate automata to ensure that execution is not prohibitively expensive.

4.2 Automata and METATEM

Concurrent METATEM actually executes formulae of both propositional and first-order temporal logic. A set of SNF program rules for the first-order temporal logic FML are of the form[2]

$$\bigcirc \bigwedge_{i=1}^{n} p_i(\bar{x}) \Rightarrow \bigvee_{j=1}^{m} q_j(\bar{y})$$

[2]Again, the full syntax and semantics of the first-order temporal logic FML are omitted (they are standard — see, for example [2]).

where each p_i, q_j is a predicate symbol, and $\bar{x}, \bar{y}$ are tuples of terms. From an arbitrary FML program formula, the representation of all grounded atomic formulae derived by assigning elements of the domain to variable symbols is impractical. An automaton must therefore provide a more general representation of first-order programs. This requires a framework in which variable assignment can be performed at a later stage (i.e. run-time). First-order temporal models are generated by combining an automaton with a variable assignment produced during execution. A finite number of automaton states can therefore represent a general interpretation of temporal states over an unconstrained domain.

We propose an automata representation for first-order logic in which an automaton state is a general representation of a set of temporal states. An automaton state provides a binary value for each predicate symbol represented by the program. For any element of the set of predicates, L_{pr}, an assignment $p \mapsto$ **true** indicates *at least one* ground predicate $p(\bar{x})$ in a temporal state generated during execution. The interpretation of an automaton state is therefore analogous to that in the PTL model, by treating predicate symbols as propositions.

To determine the set of reachable states, the program formula is evaluated for each automaton state. As grounded instantiations of predicates are not used during compilation, we may only determine rules for which the past-time formulae are *potentially* satisfied during execution. This is defined as any rule for which each positive literal used in the past-time formula has the same value in the current automaton state. A state determines a set of successors, representing potential interpretations for the next moment in time. As interpretations may only be valid under some constraint on the value of terms, a transition label specifies the conditions under which a transition may be used.

4.3 Internal Evaluation of Program Formulae

As a structure capable of representing both PTL and FML programs, we use the predicate automaton describe above. Three automata, representing subsets of the program formula, are used to limit the size of the compiled representation. In effect, the *internal* automaton models the execution sequences of programs, while the *environment* and *eventuality* automata provide direction for the execution mechanism. We define transitions in the internal automaton to reflect any value derivable from these external structures.

From the definition of the set of predicates L_a given above, a set of program rules R_a is defined. This consists of every rule for which any predicates of the past-time formula, where $p \mapsto$ **true**, belong to L_a. No rule therefore requires the receipt of environment predicates or the satis-

faction of an eventuality to be used in a temporal state.

Concurrent METATEM prohibits the use of any member of the set of environment predicates in a future-time formula [1]. As we wish to treat eventualities separately, we define a subset, $R_{a_{ev}}$, specifying rules for which the future-time formula is of the form $\Diamond q(\bar{x})$. The set of automaton states created represents interpretations for the remaining rule set, i.e. $R_a - R_{a_{ev}}$, and therefore contains only the predicates L_a. To determine the potential initiation of an eventuality, a state represents any rule belonging to $R_{a_{ev}}$ for which the past-time formula is satisfied by the automaton state valuation. An eventuality is initiated in a state specifying $r \subseteq R_{a_{ev}}$ if the past-time formula of r can be satisfied during execution.

A transition relation models the conditions required for a destination state to be a valid successor state. This is determined during execution from the input of the environment and eventuality automata, and from the evaluation of the set of rules, R_s, associated with a state.

4.4 Environment Interaction

At any moment in time, messages may be received by an object and must be incorporated in future evaluation. For any environment predicate, p, corresponding to an external message received at time i, the temporal state formula for $i+1$ must consist of $\phi \wedge p$, where ϕ is the result of the internal evaluation of i. An I/O automaton [17], capable of interaction with the environment from any state, is used to provide communication between Concurrent METATEM objects. This approach divides the actions performed by an automaton into three sets: input, internal and output. Each state may receive input from some external source, and produce output to be distributed by the communication mechanism.

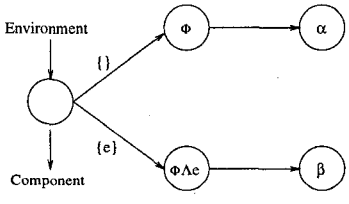

Figure 1 — Integrated I/O Automaton

Figure 1 shows the structure for a state directly combining internal and environment predicates. Transitions link successor states representing the combination of local and external actions. Labels associated with transitions specify the environment predicates required for a transition to be selected during execution. The destination state of any transition represents the result of internal rule evaluation together with the environment predicates received.

As, in general, behaviour involving interaction with the environment requires a significant number of states to represent, we define a separate structure, performing this function. Our approach uses the set of environment predicates, L_e, in order to create a separate *environment automaton*, not dependent on program rules. The environment automaton consists of an initial state, linked by transitions to states representing every subset of the environment set:

$$\forall e \subseteq L_e.\, V(s_e) = e \wedge T(s_0, s_e)$$

At each moment in time, the environment automaton performs a transition from the initial (empty) state, to some final state recording the combination of environment predicates received in this cycle. The timing of this transition is synchronized with that of the internal automaton. Instead of a single automaton state representing internal and environment predicates, a full state formula consists of the combination of environment and internal automata states. From the previous example in Figure 1, the result of internal evaluation, ϕ, is the held by the internal automaton, while the potential environment predicates received $(\{p\}, \{\})$, are represented as final states in the environment automaton (see Figure 2). The final state of the environment automaton acts as a link to the internal automaton and is useful when choosing a subsequent transition.

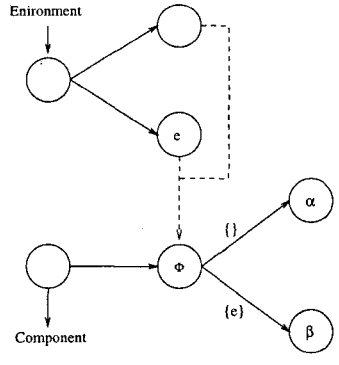

Figure 2 — Environment and Internal Automata

4.5 Eventualities

The use of Separated Normal Form (SNF) removes all future-time temporal operators except $\Diamond$ from program rules. Whenever an eventuality is initiated, there must exist a future state at which this can be satisfied. Satisfaction is possible in any temporal state in which the negation of the specified predicate is not forced. The most beneficial exe-

cution sequences provide the earliest possible satisfaction of eventualities.

While using an ω-automaton to model METATEM programs can potentially describe the full program formula, it introduces a number of problems when used as a compiled representation. As the eventualities outstanding cannot be fully determined during compilation, every state must effectively provide successor states in which the satisfaction or non-satisfaction of any eventuality is represented. As with environment predicates, this means that successor states must exist for every state, to represent the satisfaction of any subset of the set of eventualities.

For our approach, only the initiation of eventualities is specified by the internal automaton. We define a separate automaton to represent the effect of the satisfying any $ev \subseteq L_{ev}$. This is in a similar form to the environment automaton, consisting of an initial state, linked by transitions to states representing every subset of L_{ev}:

$$\forall ev \subseteq L_{ev}.\, V(s_{ev}) = ev \ \wedge \ T(s_0, s_{ev})$$

At each moment in time, a transition from the initial to some final state represents the set of eventualities satisfied in the current temporal state. This consists of some subset of the outstanding eventualities initiated by previous internal automaton states. Eventualities are satisfied in any temporal state which does not contain formulae prohibiting their satisfaction.

5 Comparison of Results

This approach to the compilation of Concurrent METATEM has been applied to a number of existing programs. In this section we examine the effect on execution time of the compiled model, and the effectiveness of the space reduction techniques applied. We demonstrate both the evaluation of an object from the well-known concurrency problem, the dining philosophers [7], and illustrate the more general case with some baseline examples. In the tests performed, we assume an equivalent subset of the set of environment propositions, received at each moment in time with each approach. We compare the interpreted language, performing direct execution of formulae, with our automata model. A single automaton implementation of the dining philosopher problem is used to evaluate of the effect of our divided model.

The dining philosophers is a concurrent problem implemented using propositional Concurrent METATEM. While the SNF representation of this is too complex to be included here, the program contains 16 propositions, with 3 eventualities and an environment set of size 4. A comparison of the execution time for this application using both automata models and the existing interpreter is given in Figure 3. The states per second ratio is unaffected by changes

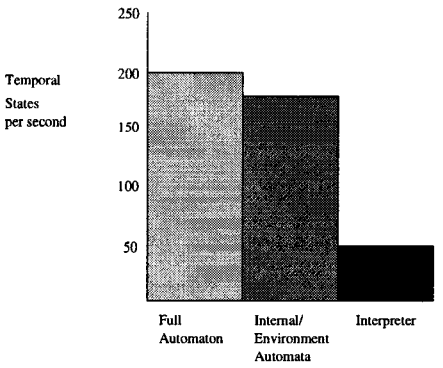

Figure 3 — Execution time for a Dining Philosopher

in the period of execution. The reduction in the execution time of our model compared with the full automaton, indicates the effect of combining the separate automata. Because of the significantly smaller size of our approach, shown in Figure 4, we consider to be an adequate tradeoff.

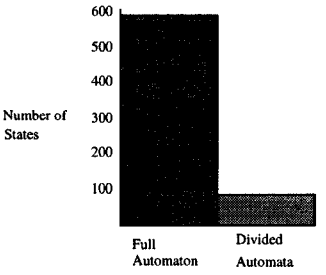

Figure 4 — No. States for a Dining Philosopher

For both automata-based approaches, the reduction in execution time over the interpreted language is large. In general, the increase in performance becomes greater as the complexity of the program increases. To illustrate this, the execution of a set of baseline examples, producing a fixed number of successor state interpretations at each moment in time, is shown in Figure 4.

As the number of interpretations increases, there is a rapid increase in the execution time for the interpreted language. This reflects the fact that all possible interpretations must be generated for each state. As a compiled model determines these choices in advance, this cost is removed from execution. The formulae producing the most interpretations for a successor state are those containing many eventualities or disjuncts of literals in future-time formula. The benefits of the compiled language over direct execution is reflected most significantly in programs of this type.

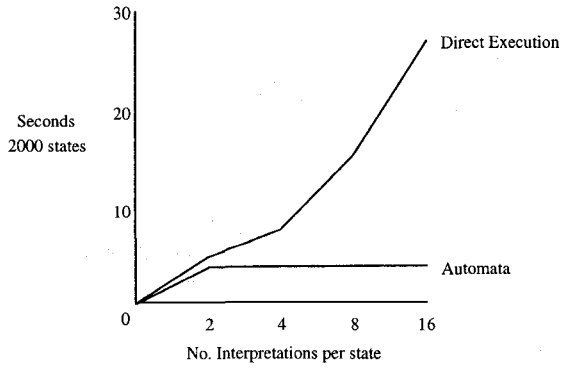

Figure 5 — Increasing Interpretations per State

6 Conclusions and Future Work

We have presented an automata-based approach to the compilation of Concurrent METATEM. Rather than utilising a standard ω-automaton representing the full program, we utilise a combination of internal, environment and eventuality automata. While this split approach produces marginally slower (but acceptably so) execution, it dramatically improves the space requirements for the compiled representation.

Future work comprises further enhancement of the automata representations, and their incorporation in a full compiler for Concurrent METATEM.

References

[1] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: An Introduction. *Formal Aspects of Computing*, 7(5):533–549, 1995.

[2] H. Barringer, M. Fisher, D. Gabbay, R. Owens, and M. Reynolds, editors. *The Imperative Future: Principles of Executable Temporal Logics*. Research Studies Press, Chichester, United Kingdom, 1996.

[3] K. Birman. The Process Group Approach to Reliable Distributed Computing. Techanical Report TR91-1216, Department of Computer Science, Cornell University, July 1991.

[4] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. Technical Report CMU-CS-92-160, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1992.

[5] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier, 1990.

[6] M. Finger, M. Fisher, and R. Owens. METATEM at Work: Modelling Reactive Systems Using Executable Temporal Logic. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE)*, Edinburgh, U.K., June 1993. Gordon and Breach Publishers.

[7] M. Fisher. A Survey of Concurrent METATEM — The Language and its Applications. In *First International Conference on Temporal Logic (ICTL)*, Bonn, Germany, July 1994.

[8] M. Fisher. Representing and Executing Agent-Based Systems. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents — Proceedings of the 1994 Workshop on Agent Theories, Architectures, and Languages*. Springer-Verlag, 1995.

[9] M. Fisher. An Introduction to Executable Temporal Logics. *Knowledge Engineering Review*, 11(1):43–56, March 1996.

[10] M. Fisher and R. Owens. From the Past to the Future: Executing Temporal Logic Programs. In *Proceedings of Logic Programming and Automated Reasoning (LPAR)*, St. Petersberg, Russia, 1992.

[11] M. Fisher and M. Wooldridge. Executable Temporal Logic for Distributed A.I. In *Twelfth International Workshop on Distributed A.I.*, Hidden Valley Resort, Pennsylvania, May 1993.

[12] M. Fisher and M. Wooldridge. A Logical Approach to the Representation of Societies of Agents. In N. Gilbert and R. Conte, editors, *Artificial Societies*. UCL Press, 1995.

[13] M. Fisher. A Normal Form for First-Order Temporal Formulae. In *Proceedings of Eleventh International Conference on Automated Deduction (CADE)*, Saratoga Springs, New York, June 1992.

[14] M. Fisher. Concurrent METATEM — A Language for Modeling Reactive Systems. In *Parallel Architectures and Languages, Europe (PARLE)*, Munich, Germany, June 1993.

[15] M. Fisher. An Open Approach to Concurrent Theorem-Proving. In *Parallel Processing for Artificial Intelligence*. North-Holland, 1996. (In press.).

[16] L. H. Landweber. Decision problems for ω-automata. *Mathematical Systems Theory*, 3:376–384, December 1969.

[17] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, September 1989. Centre for Mathematics and Computer Science, Amsterdam.

[18] A. P. Sistla, M. Vardi, and P. Wolper. The complementation problem for büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[19] P. Wolper, M. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the Twentyfourth Symposium on the Foundations of Computer Science*. IEEE, 1983.