J. Symbolic Computation (1996) **22**, 721–735

Executable Temporal Logic Systems

[A brief outline of a few important systems]

In this section, we present a brief introduction to five temporal logic-based programming languages: Chronolog, F-Limette, Concurrent MetateM, Tempura and Tokio. While a variety of verification systems based upon temporal logics have been produced, particularly involving model-checking techniques, the development of executable temporal logics is becoming increasingly important. It may turn out that the effective utilization of the full power of temporal logics in a wide range of areas will depend crucially upon the development of such languages. Since a number of researchers have tried to implement various programming languages based on temporal logic, the systems described here provide only limited coverage of the research area. However, they represent systems that are currently under active development and, as such, provide an indication of the breadth of ongoing research in this field.

Each of the sections that follow covers one of the five languages. Within each section, an introduction to the language and its applications is given, and the significant references and pointers to FTP and WWW sites are provided.

We can classify these languages in several ways. Chronolog and Concurrent MetateM use linear-time temporal logic, while Tempura and Tokio use interval temporal logic and F-Limette uses metric temporal logic. Concurrent MetateM and Tempura use deterministic execution schemes suitable for practical programming languages, while the others are extensions of Prolog (or at least SLD-resolution) and feature backtracking mechanisms. Concurrent MetateM is naturally applicable to concurrent object-based (and agent-based) systems, while the others are primarily intended for single object implementation. Thus, together, these languages cover much of the range of elements being actively explored throughout the field of executable temporal logics.

0747–7171/96/110721 + 15 $25.00/0

1. The Chronolog Family of Languages

Mehmet A. Orgun
Department of Computing,
Macquarie University, Australia

Anthony A. Faustini
Department of Computer Science & Engineering,
Arizona State University, USA

1.1. INTRODUCTION

Chronolog is a programming language based on an extension of logic programming with temporal logic (Wadge, 1988; Orgun and Wadge, 1992). Its declarative semantics is based on an extension of the standard least Herbrand models while its operational semantics is based on a temporal SLD-resolution, which is called TiSLD-resolution (Orgun, 1995). TiSLD-resolution naturally forms the basis for implementations of the language. Chronolog offers all forms of parallelism found in logic programming languages, as well as "context parallelism" by which queries at different moments in time can be executed simultaneously (Liu *et al.*, 1995).

The Chronolog family includes some extensions to the original language to improve its expressive power so that it is suitable for specifying time-dependent properties of certain problems in a natural way.

1.2. THE LANGUAGE FAMILY

The Chronolog family originated with the simple language proposed by Wadge (1988). It is based on a linear-time temporal logic in which the collection of moments in time is modeled by the set of natural numbers $\mathcal{N}$. The logic has two temporal operators, `first` and `next`. The intuitive meanings of these operators are as follows:

- `first` A: A is true at the initial moment in time,
- `next` A: A is true at the next moment in time.

Chronolog programs look like ordinary Prolog programs with the exception that they may contain (a sequence of) temporal operators applied to atomic formulae. The following simple Chronolog program defines the predicate `fib` which at each time t is true of $(t+1)^{th}$ Fibonacci number, and no other.

```
first fib(0).
first next fib(1).
next next fib(N) <- next fib(X), fib(Y), N is X+Y.
```

Read all program clauses as assertions true at all moments in time. The first two clauses define the first two Fibonacci numbers as `0` and `1`; the last clause defines the current Fibonacci number as the sum of the previous two.

In order to address certain applications such as temporal databases and knowledge-based simulation, Chronolog has been extended with an unbounded past as well as an unbounded future (Liu and Orgun, 1995; Orgun *et al.*, 1993; Orgun, 1995), in which the collection of moments in time is modeled by the set of integers $\mathcal{Z}$. The resulting language, called Chronolog($\mathcal{Z}$), has an additional operator, `prev`, to look into the past. Another extension of the language with choice predicates is suitable for modeling non-deterministic dataflow computations (Orgun and Wadge, 1992). Choice predicates in principle act like a dataflow node with multiple input lines which arbitrarily selects one of its inputs as output. Owing to non-determinism in the language, its declarative semantics is developed in terms of minimal Herbrand models (Orgun and Wadge, 1994).

The latest member of the family, Chronolog(MC), is based on a temporal logic with multiple granularity of time, in which each predicate symbol, and hence each formula, is associated with a local clock (Liu and Orgun, 1996). A local clock is a strictly increasing sequence of natural numbers, and a formula has values only at those moments in time on its clock. Local clocks are assigned to predicate symbols through programmable clock definitions and assignments.

Another language, called InTense, which includes the original Chronolog as a subset is described in Mitchell (1988) and Mitchell and Faustini (1989). InTense can, in theory, accommodate a possibly infinite number of temporal and spatial dimensions; thus each predicate varies in a time-space hyperfield, namely $\mathcal{Z}^{\omega}$. When restricted to a single time dimension the language is just like Chronolog($\mathcal{Z}$).

1.3. PROGRESS ON IMPLEMENTATION

Early implementations of Chronolog were based on meta-interpretation on top of Prolog or translation into Prolog. The current implementation is an implementation of InTense (Mitchell, 1988) supporting up to four time and four spatial dimensions. It is a full blown interpreter written in C and runs under UNIX^{TM}. The interpreter also accepts Chronolog($\mathcal{Z}$) programs, but it does not yet support extensions such as choice predicates and multiple granularity of time.

A parallel execution model for Chronolog has been described in Liu *et al.* (1995). The model is based on dataflow computation. It is supported by a virtual machine, which is granulated at clause argument level to exploit argument parallelism through temporal unification. Also, the use of a warehouse facility as an associative memory to store the results of previous computations is an important feature of this model. This feature is similar to caching (or "memoization"), and it is essential when context-parallelism is exploited.

1.4. DETAILS

The source code of the InTense interpreter can be found at

`http://www.csl.sri.com/lucid/intense`

or by emailing either `mehmet@mpce.mq.edu.au` or `tony.faustini@asu.edu`.

The speed of the interpreter is acceptable, and it is comparable to that of other advanced Prolog interpreters when running straight Prolog programs.

References

Liu, C., Orgun, M. A. (1995). Chronolog as a simulation language. In Fisher, M. (ed.), *Working Notes of IJCAI-95 Workshop on Executable Temporal Logics*, Montreal, Canada, pages 109–119.

Liu, C., Orgun, M. A. (1996). Dealing with multiple granularity of time in temporal logic programming. *J. Symbolic Computation* **22**(5/6):699–720.

Liu, C., Orgun, M. A., Zhang, K. (1995). A framework for exploiting parallelism in Chronolog. In *Proc. IEEE First International Conference on Algorithms and Architectures for Parallel Processing*, pages 163–172. Brisbane, Australia. IEEE Press.

Mitchell, W. H. (1988). Intensional Horn clause logic as a programming language – it's use and implementation. Master's thesis, Department of Computer Science and Engineering, Arizona State University, Tempe, Arizona, USA.

Mitchell, W. H., Faustini, A. A. (1989). The intensional logic language InTense. In *Proc. 1989 International Symposium on Lucid and Intensional Programming*, Arizona State University, Tempe, Arizona, USA.

Orgun, M. A. (1995). Foundations of linear-time logic programming. *International Journal of Computer Mathematics*, **58**(3–4):199–219.

Orgun, M. A., Wadge, W. W. (1992). Theory and practice of temporal logic programming. In Fariñas del Cerro, L., Penttonen, M., eds, *Intensional Logics for Programming*, pages 23–50. Oxford University Press .

Orgun, M. A., Wadge, W. W. (1994). Extending temporal logic programming with choice predicates non-determinism. *Journal of Logic and Computation*, **4**(6):877–903.

Orgun, M. A., Wadge, W., Du, W. (1993). Chronolog($\mathcal{Z}$): Linear-time logic programming. In *Proc. ICCI'93: The Fifth International Conference on Computing and Information*, pages 545–549. IEEE Computer Society Press.

Wadge, W. W. (1988). Tense logic programming: a respectable alternative. In *Proc. 1988 International Symposium on Lucid and Intensional Programming*, pages 26–32, University of Victoria, Victoria, B.C., Canada.

2. "F-Limette" Fuzzy Logic Programming Integrating Metric Temporal Extensions

Karl Schäfer
Institut für Algorithmen und Kognitive Systeme,
Universität Karlsruhe (TH), Germany

Christoph Brzoska
Institut für Logik, Komplexität und Deduktionssysteme,
Universität Karlsruhe (TH), Germany

2.1. INTRODUCTION

F-LIMETTE is a declarative programming language that combines fuzzy logic programming and point-based metric temporal logic programming over linear, discrete, and unbounded time. The semantics of the underlying logic FMTL are defined by interpreting ground atoms as fuzzy sets over the time domain. The FMTL is restricted to a Horn fragment that supports a variety of fuzzy and temporal operators while maintaining smallest Herbrand models, and feasible time sets are currently restricted to divisions (finite unions of intervals) with respect to the required implicit representation of time sets closed to arithmetics. The execution of queries is performed using a correct and complete state-based tableau calculus that transforms modular states consisting of substitutions, timesets, and truthvalues. Its methodology is in a similar fashion to the Hoare calculus known from program verification. Our F-LIMETTE system further supports constructive negation, basic temporal term narrowing, concurrent execution of operationally combined and alternative goals, heuristic search strategies up to incomplete pruning capabilities, and completeness-maintaining program updates during query execution. Thus it combines declarative and imperative methodologies by relating the assertional time structure referenced by formulae with the proof time structure defined by the linear sequence of inference steps.

2.2. LANGUAGE

The syntax of F-LIMETTE is an upward-compatible extension of PROLOG and is formalized by the following tree grammar for rules $\tilde{\forall} R$ (the elements of F-LIMETTE packages) and queries $\tilde{\exists} B$:

$$\begin{array}{lll} R & ::= & A \mid R \leftarrow_\nu B \mid \ \downarrow_\kappa R \ \mid \uparrow_\lambda R \mid \ \Box_S R, \\ B & ::= & A \mid B_1 \wedge_\nu B_2 \mid B_1 \vee_\nu B_2 \mid \downarrow_\kappa B \mid \uparrow_\lambda B \mid \Diamond_S B \mid \Box_F B \\ & & \qquad (S, F \subseteq \mathcal{T},\ |F| < \infty) \\ & & \qquad \kappa \in [0,1],\ \lambda \in (0,1] \end{array}$$

where A is an atom, S, F are feasible subsets of the time set $\mathcal{T}$, $\kappa, \lambda \in [0,1] \subset \mathcal{R}$ are fuzzy truth values, and $\nu \in \{w, m, s\}$ is a fuzzy operator semantic *(weak, medium, strong)*. This Horn fragment in conjunction with the operator semantics allows to state all program rules in a normal form

$$\uparrow_{\lambda_{i1}} \downarrow_{\kappa_{i1}} \Box_{S_{i1}} \Big(\ \ldots \ \uparrow_{\lambda_{in_i}} \downarrow_{\kappa_{in_i}} \Box_{S_{in_i}} (\uparrow_{\lambda_{i(n_i+1)}} \downarrow_{\kappa_{i(n_i+1)}} \Box_{S_{i(n_i+1)}} H_i \leftarrow_{\nu_{in_i}} B_{in_i}) \Big) \leftarrow_{\nu_{i(n_i-1)}} \ \ldots \ \leftarrow_{\nu_{i1}} B_{i1}$$

of nested rules, where $\Box_S$ denotes universal temporal qualification over the time set S, and $\downarrow_\kappa, \uparrow_\lambda$ denote fuzzy reduction and fuzzy intensification, respectively. The common temporal operators next $\circ^n$ and previous $\bullet^n$ are both equivalent to either $\Box_{\{\pm n\}}$ or $\Diamond_{\{\pm n\}}$.

The full set of F-Limette temporal operators includes universal $\Box_S$ and existential $\Diamond_S$ qualification, bounded since $\mathcal{S}_S$ and until $\mathcal{U}_S$, and fixation **init** to the initial time point $0 \in \mathcal{T}$. A temporal abundance $\Box^\theta_F$ operator ($\theta \in [0,1]$, true for at least $\lceil \theta \cdot |F| \rceil$ time points within subset $F \subset \mathcal{T}$) smoothly interpolates between universal $\Box_F \equiv \Box^1_F$ and existential $\Diamond_F \equiv \Box^\varepsilon_F$ ($\varepsilon \leq |F|^{-1}$) temporal qualification.

Predicate logic operators like negation as failure **not** and implicit set representation **findall** need special consideration when extended to the temporal and fuzzy temporal case. Thus negation as failure is constructive with respect to existentially quantified variables and points of time, i.e. **not** relies on the "closed world assumption", but does not requantify any unbound variable.

The primary fuzzy operators are reduction $\downarrow_\kappa$, $\mathcal{I}[\![\downarrow_\kappa \mathcal{F}]\!] = \min(1, \kappa^{-1} \cdot \mathcal{I}[\![\mathcal{F}]\!])$, and intensification $\uparrow_\lambda$. They allow us to assert or request κ-reduced truth of temporal knowledge, and to intensify truth of a consequent with respect to a precondition. F-Limette distinguishes exactly three kinds of subsumption $\leftarrow_\nu$, compiled into respective inference rules. It supports user-defined modifiers (*very, some, ...*), and defuzzifiers (*select maximum, compute weighted sum, ...*).

Concurrent execution in F-Limette is controlled by attributed operators that specify the temporal precedence or simultaneity of subordinate execution paths. For instance, the conjunction $\mathcal{F}_1 \wedge_S \mathcal{F}_2$ selects serialized processing *($\mathcal{F}_1$ and then $\mathcal{F}_2$)*, while $\bigwedge_{P,i} \mathcal{F}_i$ selects concurrent processing of the $\mathcal{F}_i$. Synchronization is formalized by their combination, e.g. $(\mathcal{F}_1 \wedge_P \mathcal{F}_2) \wedge_S \mathcal{F}_3$ ($\mathcal{F}_3$ *after both*, $\mathcal{F}_1$ *and* $\mathcal{F}_2$). The cut **!** operator terminates alternative paths upon its execution.

Search strategies (depth search, breadth search, heuristic search) are selected by strategy operators that choose a predefined strategy within the execution of the subordinate goal. Different strategies may interact at the same time inside the entire system, each one assigned to a bunch of execution paths. Heuristic search is controlled by a parametric quality function of the state space that measures potential success of a single partial derivation, namely execution time progress (position in proof tree, prefers short proofs), rigidity of the time constraint (cardinality of an input time set, prefers large sets), generality of the partially instantiated goal (referenced part of substitution, prefers unbound variables), and required truth value (prefers small lower bounds).

2.3. implementation

The prototype implementation of the non-fuzzy kernel language generates Prolog source code which efficiently emulates the tableau calculus by SLD resolution and database manipulations. The full implementation is an interpreter written in C that directly implements the calculus. This migration step had been considered due to the lack of support for constructive negation (systems of term inequalities), runtime skolemization, breadth search strategies, and parallel execution in standard Prolog.

The system is available as a standalone program operating in either batch or interactive shell mode, or as an object code link library with procedural interface as a subordinate to existing application programs.

Concurrent execution is controlled by an integrated scheduler, operating at the granularity level of single inference steps. Derivation paths are represented equivalent to processes of an operating system. Shell commands and logic operators that come along with optional attributes for flow control may create, suspend, synchronize and terminate such paths.

F-LIMETTE has been successfully compiled for the SunOS 4.x and Solaris[†] operating systems. Solaris offers multi-threaded application processes, and concurrent inference is achieved really in parallel by a farm of inference threads synchronized by a single scheduler thread. The Solaris environment further supports the dynamic integration of downcoded packages (predicates implemented in C code).

2.4. APPLICATION

The system can be applied as a rule-based database manager for definite temporal knowledge. It integrates an extended answer generation scheme that explains the derivability of queries by analysing resulting proof trees. Modular databases with update facility, efficient lookup procedures and a compact mass storage format support large-scaled databases. Quick-hit incomplete query processing can be realized using pruned breadth search and the memorizing of recent proofs.

The combination of temporal and fuzzy logic formalism recommends F-LIMETTE for prototyping, specifying, and implementing applications in AI, where uncertain temporal information has to be represented and to be processed. We apply the system for the evaluation of process descriptions from natural image sequences by traversing situation graph trees, i.e. a hierarchy of transition diagrams, where states and actions are represented as fuzzy conceptual graphs, while the expansion of state and intermission intervals is constrained temporally. Those process descriptions, written in a language called SIT^{++}, and uncertain definite temporal assertions acquired from image evaluation are translated into F-LIMETTE source code, such that proof strategies correspond to traversal strategies directly.

References

Brzoska, C. (1993). Temporal-logisches Programmieren *(Temporal Logic Programming)*. Dissertation, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe/Germany.

Brzoska, C., Schäfer, K. (1993). LIMETTE—Logic Programming Integrating Metric Temporal Extensions. Report 9/93, Universität Karlsruhe, Institut für Logik, Komplexität und Deduktionssysteme.

Brzoska, C., Schäfer, K. (1994). Temporal Logic Programming applied to Image Sequence Evaluation. Logic Programming: Formal Methods and Practical Applications, C. Beierle and L. Plümer (ed.), Elsevier Science Publishers B.V., Amsterdam, Ch. **13**, 381–395.

Schäfer, K. Unscharfe zeitlogische Modellierung von Situationen und Handlungen in Bildfolgenauswertung und Robotik *(Fuzzy Temporal Representation of Situations and Actions for Image Evaluation and Robotics)*. Universität Karlsruhe, Fakultät für Informatik, Karlsruhe/Germany *(in preparation)*.

[†] *SunOS* and *Solaris* are registered trademarks of Sun Microsystems Ltd.

3. The Concurrent METATEM System

Michael Fisher & **Adam Kellett**[†]
Department of Computing,
Manchester Metropolitan University, United Kingdom

3.1. Introduction

Concurrent METATEM is a programming language based upon the direct execution of temporal logic formulae (Fisher, 1993). It consists of three distinct aspects: a basic execution mechanism for temporal formulae based upon the *Imperative Future* approach (Barringer *et al.*, 1996); an operational model that treats single executable temporal logic programs as asynchronously executing objects (agents) in a concurrent object-based system; and a flexible and powerful communication mechanism, based upon *broadcast* message-passing. Together, these features provide a coherent and consistent programming model. While the development of the language is ongoing, with both the syntax and implementation currently being refined, a variety of reactive systems have been represented and prototyped (Fisher, 1994). In particular, the language has significant potential applications in the area of multi-agent systems (Fisher, 1995).

3.2. language

The syntax for Concurrent METATEM programs is straightforward. For example, the following represents a partial definition of an agent, called 'phil' that will recognize 'give' and 'hungry' messages and can broadcast 'give', 'hungry', and 'eat' messages.

phil(give,hungry)[give,hungry,eat]:

$$
\begin{array}{rcl}
\textbf{start} & \Rightarrow & \mathsf{has(left)} \wedge \neg\mathsf{has(right)}; \\
\odot(\neg\mathsf{has(left)} \ \vee \ \neg\mathsf{has(right)}) & \Rightarrow & \neg\mathsf{eat(phil)}; \\
\textbf{start} & \Rightarrow & \Box\Diamond\mathsf{eat(phil)}; \\
\odot\mathsf{hungry(Other)} & \Rightarrow & \Diamond\mathsf{give(phil,Other)}.
\end{array}
$$

The above agent consists of a several temporal rules, which together provide its behaviour. For example, the last rule states that if a message of the form 'hungry(phil2)' was received in the last moment in time ('$\odot$' is the *last-time* operator), then the agent will attempt to satisfy '$\Diamond$give(phil,phil2)', where '$\Diamond$' is the "sometime in the future" operator.

The logic used as a basis for Concurrent METATEM is a discrete, linear temporal logic. This presents a simple view of time and, in doing so, provides the systems designer with a direct analogy between the models for the logic and the discrete, linear execution sequences with which he or she is familiar. Thus, the temporal operators that can be used in Concurrent METATEM programs relate to the logical operators used in this model of time, e.g. '$\Diamond$' (sometime in the future), '$\blacksquare$' (always in the past), '$\mathcal{U}$' (until), '$\Box$' (always in the future) and '**start**' (at the beginning of time).

Examples of applications include transport simulation (Finger *et al.*, 1993), concurrent theorem-proving (Fisher, 1996), Distributed AI (Fisher and Wooldridge, 1993), and the modelling of simple artificial societies (Fisher and Wooldridge, 1995).

[†] E-mail: `{M.Fisher,A.Kellett}@doc.mmu.ac.uk`

3.3. IMPLEMENTATION

The current implementation is an interpreter written in C++. It simply reads the program text (consisting of agent definitions of the above form) from `stdin` and writes output (i.e. messages broadcast by agents) to `stdout`. The interpreter is implemented on stand-alone workstations (so far both Sun SPARCs and PCs). The concurrent activity is simulated through the scheduling policy involved in the execution of each agent. Command line arguments currently allow the user to specify the number of time steps to be executed, the debugging level, an output file for debugging information, and whether the execution is to be synchronous or asynchronous.

3.4. DETAILS

Details of the implementation (current version 2.1) can be found by emailing the authors, or via `http://www.doc.mmu.ac.uk/STAFF/michael/cmet.html`. The current version implements the basic Concurrent MetateM system, incorporating asynchronous and synchronous execution, first-order temporal rules, and broadcast message-passing. The features that it does not yet implement are: creation and manipulation of groups; dynamic agent creation; user-defined primitives; synchronisation mechanisms. These features, together with an improved user interface, are being developed for version 3. While the current interpreter has acceptable performance on small examples, producing about 20 temporal states per second on a SPARCStation-2, it is inappropriate for large-scale experiments. Consequently, we are currently developing a compiler for the language that will not only improve performance, but also will allow the utilization of parallel architectures.

References

Barringer, H., Fisher, M., Gabbay, D., Owens, R., Reynolds, M., editors (1996). *The Imperative Future: Principles of Executable Temporal Logics.* Research Studies Press, U.K.

Finger, M., Fisher, M., Owens, R. (1993). 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. Gordon and Breach Publishers.

Fisher, M., Wooldridge, M. (1993). Executable Temporal Logic for Distributed A.I. In *Twelfth International Workshop on DAI*, Hidden Valley Resort, Pennsylvania.

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

Fisher, M. (1993). Concurrent MetateM — A Language for Modeling Reactive Systems. In *Parallel Architectures and Languages, Europe (PARLE)*, Munich, Germany. (Published in *Lecture Notes in Computer Science*, volume 694, Springer-Verlag).

Fisher, M. (1994). A Survey of Concurrent MetateM — The Language and its Applications. In *First International Conference on Temporal Logic (ICTL)*, Bonn, Germany. (Published in *Lecture Notes in Computer Science*, volume 827, Springer-Verlag).

Fisher, M. (1995). Representing and Executing Agent-Based Systems. In Wooldridge, M., Jennings, N. R., editors, *Intelligent Agents.* Springer-Verlag.

Fisher, M. (1996). An Open Approach to Concurrent Theorem-Proving. In *Parallel Processing for Artificial Intelligence (volume 3)*, Elsevier B.V. In Press.

4. The Programming Language Tempura

Ben Moszkowski[†]
Department of Electrical and Electronic Engineering,
University of Newcastle upon Tyne, United Kingdom

4.1. Introduction

Tempura (Moszkowski, 1986) is an experimental programming language based on executable subsets of Interval Temporal Logic (ITL) (Moszkowski, 1983; Halpern *et al.*, 1983; Moszkowski, 1985). We developed it in order to narrow the gap between temporal logics and the imperative programs they typically describe. Tempura contains destructive assignment constructs, ";" (sequential composition) and while-loops. It consequently looks much more like a conventional imperative programming language than most other executable temporal logic systems. Unlike most logic-based languages, it is very deterministic and offers no facilities for backtracking. This approach has been adapted primarily for execution efficiency. A number of temporal constructs have been included in Tempura and can be used for hardware simulation and other applications. These include $\Box$ (*always*), $\bigcirc$ (*next*) and a form of projection between different granularities of time. Almost all of Tempura's constructs can be formally defined using conventional first-order logic and the ITL operators *skip*, ";" (*chop*) and "*" (*chop-star*). These three operators respectively provide ways to measure interval length, sequentially combine formulae and iterate a formula. The basic semantics of a Tempura program is readily obtained by considering it to be an ITL formula. Consequently, logical implication can be used to relate an ITL specification with a Tempura program that implements it.

4.2. Language

Tempura programs typically consist of a logical conjunction of various ITL formulae and are executed in a simulated time consisting of a sequence of one or more discrete states. Here is an example:

$$M = 4 \wedge N = 1 \wedge \mathit{halt}\,(M = 0) \wedge M \;\mathit{gets}\; M + 1 \wedge N \;\mathit{gets}\; 2N. \tag{4.1}$$

This executes over 4 units of discrete time (i.e., 5 states). In the initial state, M equals 4 and N equals 1. Over adjacent states, M increases by 1 and N doubles until the computation terminates when M equals 0. The behavior of M and N can also be expressed by the following logically equivalent ITL formula:

$$\begin{gathered} M = 4 \wedge N = 1 \wedge \bigcirc(M = 3 \wedge N = 2) \wedge \bigcirc\bigcirc(M = 2 \wedge N = 4) \\ \wedge \bigcirc\bigcirc\bigcirc(M = 1 \wedge N = 8) \wedge \bigcirc\bigcirc\bigcirc\bigcirc(M = 0 \wedge N = 16). \end{gathered} \tag{4.2}$$

A slight variation of this can be directly executed by the Tempura interpreter.

The following ITL fragment, which operates over 1 unit of time (2 states), decreases M by 1 and doubles N:

$$\mathit{skip} \wedge M \leftarrow M - 1 \wedge N \leftarrow 2N. \tag{4.3}$$

The *skip* construct specifies that the interval has length 1. The two *temporal assignments* modify in parallel the values of M and N. Unlike conventional imperative assignment,

[†] E-mail: `Ben.Moszkowski@ncl.ac.uk`

temporal assignment lacks *framing* and does not ensure the all nonassigned variables remain unchanged. Instead, this must be explicitly stated by using multiple temporal assignments in parallel or by using the *stable* construct (e.g., *stable* A). The fragment (4.3) can be included in a while-loop and combined with initialization to obtain the program now shown:

$$M = 4 \wedge N = 1 \wedge \textit{while}\, M \neq 0 \;\textit{do}\; (\textit{skip} \wedge M \leftarrow M - 1 \wedge N \leftarrow 2N). \tag{4.4}$$

This is logically equivalent in ITL to formulae (4.1) and (4.2). Note that the conjunction of two temporal assignments performs them in parallel. Therefore, the following exchanges the values of the variables A and B in one unit of time:

$$\textit{skip} \wedge A \leftarrow B \wedge B \leftarrow A.$$

Universal quantification ($\forall$) provides a way to perform a large number of activities in parallel. Local scoping of variables is typically achieved by enclosing them in existential quantifiers ($\exists$). There are also some constructs for input and output as well as projection between different time granularities.

Timing constraints can be specified. The *len* construct illustrated in the next program offers one way to do this:

$$M = 4 \wedge N = 1 \wedge \textit{len}(M) \wedge M \;\textit{gets}\; M + 1 \wedge N \;\textit{gets}\; 2N. \tag{4.5}$$

This is logically equivalent in ITL to formulae (4.1), (4.2) and (4.4). The *len* construct can also be used in nested constructs. For example, here is a program which describes a digital signal X ranging over 0 and 1 with stable periods of equal widths over an overall interval of 18 units of time (19 states):

$$X = 0 \wedge \textit{for}\; 3\,\textit{times}\;\textit{do}\;\big((\textit{len}(5) \wedge \textit{stable}\; X); (\textit{skip} \wedge X \leftarrow \neg X)\big).$$

This can be combined with other Tempura fragments to model a circuit containing gates which respond to X's changes.

As with any realistic logic-based programming language, there are major restrictions on permitted Tempura constructs. The language's lack of nondeterminism limits the logical operator $\vee$ (*or*) to being used only in Boolean tests (as in most programming languages). The temporal operator $\Diamond$ (*sometimes*) is completely absent from Tempura.

4.3. IMPLEMENTATION

Moszkowski (1986) presents the design of a Tempura interpreter which executes a program by successively examining one state at a time. The interpreter determines the program's effect on variables in the current state and also extracts from the program those parts which should be postponed until later on. During the processing of a program in a given state, an immediate assignment of the form $v = e$ sets the variable v's current value to be that of the expression e. Sometimes this requires multiple passes over a formula as the following illustrates:

$$J = I + 1 \wedge I = 2.$$

Of course, circular assignments such are $I = I + 1$ are not valid. Some assignments are postponed until the next state. For example, the following equality increases $\bigcirc I$ (I's next value) by 1:

$$(\bigcirc I) = I + 1.$$

As each program piece is analysed, those parts which do not affect the current state are pushed off until later. For example, the fragment $\Box(K = 1)$, which sets K to 1 in all states, is reduced to the equivalent formula shown below:

$$K = 1 \wedge ⓦ\Box(K = 1).$$

Here ⓦ (*weak-next*) is a weak form of the $\bigcirc$ operator and is trivially true on any interval with only one state. The net effect of the reduction is to assign K the value 1 in the current state and to save the formula $\Box(K = 1)$ for evaluation in the successor state if there is one. A sequential formula of the form $S; S'$ is evaluated by first interpreting the left subformula S until its associated subinterval terminates and then interpreting the right subformula S' from then on until the completion of the overall interval. While-loops and other interactive constructs are processed in an analogous way.

We originally implemented an interpreter for Tempura in Lisp. A port to C done by R. Hale and sample programs are available from the author by email.

4.4. DISCUSSION

Tempura provides a framework for investigating the relationship between conventional imperative programming and temporal logics. Although Tempura has never been widely used, it has helped to generate interest in executable temporal logic systems. For example, the Tokio (Fujita *et al.*, 1986) and METATEM (Barringer *et al.*, 1989) systems have both been influenced by it. In addition, various researchers have applied Tempura to hardware simulation and other areas where timing is important (Dowsing and Elliot, 1991; Dowsing *et al.*, 1994; Hale, 1987; Kilis *et al.*, 1989; Lichota, 1988). Ruddle (1992) favorably rated ITL and Tempura as the basis of a formal method for fast prototyping of real-time C programs.

Some deficiencies in Tempura such as the lack of any kind of framed assignment have stimulated research on extending Interval Temporal Logic (Hale, 1988; Moszkowski, 1995). The continued vitality of conventional programming languages suggests that further development on executable imperative subsets of temporal logics is a worthwhile endeavor.

References

Barringer, H., Fisher, M., Gabbay, D. *et al.* (1989). MetateM: A framework for programming in temporal logic. In: *Proc. REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, LNCS 430, Springer-Verlag, Berlin.

Dowsing, R., Elliot, R. (1991). A higher level of behaviorial specification: An example in Interval Temporal Logic. In: Proc. EUROMICRO '91, 517–524.

Dowsing, R., Elliot, E., Marshall, I. (1994). Automated technique for high-level circuit synthesis from temporal logic specifications. *IEE Proc.-Comput. Digit. Tech.*, **141**, (3) 145–152.

Fujita, M., Kono, S. *et al.* (1986). Tokio: Logic programming language based on temporal logic and its compilation to Prolog. In: *Proc. 3rd Int'l. Conf. on Logic Programming*, LNCS 225, Springer-Verlag, Berlin, 695–709.

Hale, R. (1987). Temporal logic programming. In: *Temporal Logics and Their Applications* (A. Galton, ed.), London: Academic Press, 91–119.

Hale, R. (1988). Programming in Temporal Logic. PhD Thesis, University of Cambridge, Cambridge, England.

Halpern, J., Manna, Z., Moszkowski, B. (1983). *A hardware semantics based on temporal intervals.* In: ICALP83, LNCS 154, Springer-Verlag, Berlin, 278-291.

Kilis, D., Esterline, A. C., Slagle, J. R. (1989). Specification and verification of network protocols using executable temporal logic. In: *Proc. IFIP Congress '89*, Amsterdam, North Holland Publishing Co, 845–850.

Lichota, R. W. (1988). Evaluating Hardware Architectures for Real-Time Parallel Algorithms using Temporal Specifications. PhD Thesis, University of California at Los Angeles.

Moszkowski, B. (1983). Reasoning about Digital Circuits, PhD thesis, Stanford Univ. (Available as Tech. Rep. STAN–CS–83–970).

Moszkowski, B. (1985). A temporal logic for multilevel reasoning about hardware, *Computer* **18**(2), 10–19.

Moszkowski, B. (1986). Executing Temporal Logic Programs, Cambridge: Cambridge Univ. Press.

Moszkowski, B. (1995). Embedding imperative constructs in Interval Temporal Logic. Internal memorandum EE/0895/M1, Dept. of Elec. and Elec. Eng., University of Newcastle, UK. Available from the author (`Ben.Moszkowski@ncl.ac.uk`).

Ruddle, A. R. (1992). Formal methods in the specification of real-time, safety-critical control systems. In: *Proc. Z User Workshop* (J. P. Bowen and J. E. Nicholls, eds.), London, Springer-Verlag, pp. 131–146.

5. The Tokio System

Shinji Kono
Information Engineering, University of the Ryukyus, Japan

M. Fujita
Fujitsu Labratory, USA

5.1. INTRODUCTION

Tokio is a logic programming language based on Interval Temporal Logic. It was designed in 1984 intended for use as a Hardware Description Language. Since then several tools for Tokio have been developed. Tokio system consists of three parts;

Prolog like programming language: Tokio
RTL level hardware description language: RTL-Tokio
Automatic verifier on Propositional Interval Temporal Logic: Lite

Programming language Tokio is an extension of Prolog and can be considered as another implementation of Tempura. RTL-Tokio is a restricted version of Tokio for Register Transfer Level description of a computer architecture. Using RTL-Tokio, automatic verification of pipelined CPU is demonstrated. Lite is an automatic verifier for ITL and can be used as a logic synthesis tool or model checking tool as well. It is also possible to combine the output of Lite with Tokio.

5.2. LANGUAGE

The syntax of Tokio is an extension of Prolog. It includes all the Prolog Language and temporal logic operators, such as `@` (next), `&`(chop), `<>` ($\Diamond$, sometime) or `#` ($\Box$, always) can be used in the body part of a clause. Like Prolog, negation is not allowed in the body part. For example,

```
receive_data(Call,Hear,Data) :-
     halt(Hear=1)
     &       write(accept(Data)),halt(Hear=0)
     &       @receive_data(Call,Hear,Data).
receive(Call,Hear,Data) :-
     #receive_sync(Call,Hear),receive_data(Call,Hear,Data).
```

This is a part of a send/receive protocol specification. RTL-Tokio allows only one chop operator in a body and the former part of the chop operator contains only a fixed length specification. Because of backtracking mechanisms, it is possible to execute a slightly wider range of programs in Tokio than in Tempura.

In the case of the automatic verifier, Lite, we have no restrictions on ITL formulae. It accepts a local ITL formula and outputs a finite state machine which accepts all satisfiable states of the formula. For example,

```
          exists(Q,(Q,
            '[]'((Q ->
                  (((((a,skip) & (b,skip)), @ keep(~Q)) & Q);empty))
            )))
     <->  *(((a,skip) & (b,skip) ; empty))
```

will generate a state machine accepting everything. That is, the left- and right-hand side formulae are equivalent.

The executable range of the Tokio programming language is limited because of clausal form and restriction of negation. However, with an appropriate use of Lite (which can handle arbitrary ITL formulae), one can use Tokio/Lite for almost all aspects of hardware designs.

5.3. IMPLEMENTATION

Prolog is used to implement all three systems. Tokio is implemented as a translator from Tokio programs to Prolog. It is 7 times slower than original Prolog if we execute the same program.

The Lite verifier is also implemented in Prolog. It has a finite state machine format to interface with logic synthesis tools. It also has a GUI based tool.

5.4. DETAILS

You can find Tokio system in

```
ftp://ftp/csl.sony.co.jp/CSL/soft
```

This includes incomplete manuals and several large examples such as mc6502 micro processor. It is possible to simulate practical and large hardware designs. However, in the case of the automatic verification tool, Lite, the size of specification that can be processed is limited because it requires huge computational resources.

References

Kono, S. (1992). Automatic Verification of Interval Temporal Logic. In *Proc. 8th British Colloquium for theoretical computer science.*

Kono, S., Aoyagi, T., Fujita, M., Tanaka, H. (1985). Implementation of Temporal Logic Programming Language Tokio. In *Proc. Logic Programming Conference, LNCS-221*, Springer-Verlag.

Moszkowski, B. (1983). Reasoning about Digital Circuits. Technical Report STAN-CS-83-970, Dept. of Computer Science, Stanford University.

Kono, S. (1994). A Combination of Clausal and Non Clausal Temporal Logic Program. In *Executable Modal and Temporal Logics*, volume LNAI-897. Springer-Verlag. Lecture Notes in Artificial Intelligence.