

# KNOWLEDGE-BASED SIMULATION WITH CHRONOLOG

CHUCHANG LIU and MEHMET A. ORGUN

*Department of Computing, Macquarie University  
Sydney, NSW 2109, Australia*

E-mail: {cliu, mehmet}@krakatoa.mpcce.mq.edu.au

## ABSTRACT

Simulation methods deal with processes evolving in time. Since temporal logic can model time-dependent dynamic properties of certain problems of real world, it is a natural idea to select a certain type of “executable” temporal logic as a language to describe simulation problems. Chronolog(Z) is a temporal extension of logic programming and it is therefore suitable for specifying time-dependent properties in a natural way. This paper shows that it can be effectively used for describing and implementing simulation. A simulated system consists of a rule-based component and an initial-condition component. In the Chronolog execution model, the rule clauses of a simulation program, which describe the dynamics of a modeled system are stored in the program store as individual dataflow graphs, and the initial conditions of the simulated system, i.e. the facts that are true at the time when simulation begins, are stored in the warehouse for priming the simulation. When a simulation is running, some results of previous computations, which can be also treated as part of the initial conditions, may be stored in the warehouse for reuse. This paper shows that there are several important advantages of describing and implementing simulation in the Chronolog execution model: an extensive modeling power, a well-defined semantics which can be easily understood, and computational efficiency.

## 1. Introduction

Since late 1970s, there has been a substantial interest in knowledge-based simulation methods. Ören<sup>5</sup> discusses possible future research directions for knowledge-based simulation systems. Many of these systems provide support for rule-based and object-oriented paradigms and for powerful knowledge representation schemes.

However, since simulation methods deal with processes evolving in time, knowledge-based simulation methods based on first-order logic must provide an explicit support for time. Since temporal logic can naturally model time-dependent dynamic properties of certain problems of real world, we can select a certain type of “executable” temporal logic as a language to describe simulation problems.

Chronolog<sup>7</sup> is a temporal extension of logic programming, based on a linear-time temporal logic<sup>2</sup> with an unbounded future in which the set of natural numbers models the collection of moments in time. Chronolog(Z) extends Chronolog with a linear-time temporal logic with unbounded past and future. We outline a parallel execution model for the temporal logic programming language Chronolog<sup>6,7</sup>, called CHEM (CHronolog Execution Model). CHEM is a data-driven execution model, which has a flexible capability to support parallelism at various levels<sup>3,4</sup>. It is suitable for specifying time-dependent properties in a natural way, and we claim that it can be effectively used for describing and implementing simulation.

A simulated system consists of a rule-based component and an initial-condition component. In the Chronolog execution model, the rule-based component which describe the dynamics of a modeled system are stored in the program store as individual dataflow graphs, and the initial-condition component of the simulated system, i.e. the facts that are true at the time when simulation begins, are stored in the warehouse for priming the simulation. When a simulation is running, some results of previous computations, which can be also treated as initial conditions, may be stored in the warehouse for reuse. The use of a warehouse facility to store the results of previous computations is an important feature of CHEM. The strategy to employ a warehouse can effectively avoid repeated and unnecessary computations.



The paper is organized as follows. In section 2, we give an overview of Chronolog and its execution model, called CHEM. Section 3 describes how Chronolog can be used as a simulation language using an example of a Flexible Manufacturing System. Section 4 outlines the way in which a simulation system can be implemented using the execution model of Chronolog.

## 2. Overview of Chronolog

In this section, we briefly introduce the language Chronolog, including its logical basis and an abstract form of its execution model which we call CHEM.

### 2.1. Chronolog Programs

Temporal logic<sup>8</sup> can be regarded as a special case of intensional logic where the set of possible worlds models a collection of moments in time, usually discrete, linearly ordered, without a last moment. Chronolog( $Z$ ) is an extension of logic programming based on temporal logic. Its temporal logic has three temporal operators, **first**, **prev** and **next**, which refer to the initial moment, the previous moment and the next moment in time respectively. The set  $Z$  of integers is the collection of moments in time. The underlying temporal logic of Chronolog, denoted by  $TL$ , refers to a natural extension of a standard first-order logic language.

The temporal logic of Chronolog( $Z$ ) is defined as follows: We begin with a standard first-order language and extend it with three new formation rules: if  $F$  is a formula of  $TL$ , so are **first**  $F$ , **prev**  $F$  and **next**  $F$ . For more details on this temporal logic and its formal properties we refer the reader to Orgun and Wadge<sup>6</sup>. The informal semantics of temporal operators **first**, **prev** and **next** are as follows: Let  $t \in Z$ . A formula of the **first**  $F$  is true at the time  $t$  if and only if  $F$  is true at time 0, a formula of the form **prev**  $F$  is true at time  $t$  if and only if  $F$  is true at time  $t - 1$ . and a formula of the form **next**  $F$  is true at time  $t$  if and only if  $F$  is true at time  $t + 1$ .

A formula of  $TL$  is called a temporal formula. An atom of  $TL$  is called a temporal logic atom, or, for short, a tl-atom of  $TL$ .

Chronolog programs look like standard logic programs with the only difference being that program clauses contain applications of temporal operators to atomic formulas.

For example, consider the following temporal logic program that specifies the **rotate** predicate to generate all possible rotations of a given list.

```
first rotate(L) <- first (~ input(L)).
next rotate(L) <- rotate([H|T]), append(T,[H],L).

append([],L,L).
append([X|L1],L2,[X|L3]) <- append(L1,L2,L3).
```

Here **first rotate(L)**, **first (~input(L))**, **next rotate(L)** etc. are all tl-atoms. The first two clauses contain the use of temporal operators. The first clause says that the initial value of the list to be rotated is provided from the standard input. The symbol **~** is a directive which tells the implementation to store the input value in the warehouse permanently so that the user will not be asked to provide it again. It has no declarative meaning. The second clause is used to rotate the previous value of the list to obtain the new list, rotated one position to the right. The goal **<- rotate(L)** starts a non-terminating computation for producing all possible rotations of the input list, one at a time.

In temporal logic programs, non-terminating computations is modeled by time-varying predicates. Time-dependencies in the program are specified by the use of temporal operators. For instance, the **rotate** predicate is defined by an initial clause and a program clause with recursive past dependencies.

For temporal logic programs, we need the following definition:

- A tl-atom without the temporal operator **first** is called an open-end tl-atom;

- A tl-atom that contains the temporal operator **first** is called a fixed-time tl-atom.

According to the axioms of  $TL^6$ , all superfluous applications of temporal operators in a formula can be eliminated (e.g. **first** or **next** followed by **first**). So, we can simplify a tl-atom so that it is either an open-end tl-atom, which does not contain any temporal operator or it only contains a number of applications of **next**, or a fixed-time tl-atom which has an application of **first** followed by a number of applications of **next**. Following the above definition, we have:

- If all tl-atoms of a goal are fixed-time, then the goal is called a fixed-time goal; otherwise it is a open-end goal.

For the above example, we consider the fixed-time goal `<- first rotate(L)`. It will match the first clause, and the input predicate will ask for a ground term for the variable **L**, expecting it to be a list. Suppose the term `[0,1,2,3,4]` is supplied as an input value, in other words, the goal `first input([0,1,2,3,4])` succeeds. The answer to the original goal is then a substitution instance with **L** replaced by `[0,1,2,3,4]`.

The goal `<- rotate(L)` is an example of an open-end goal. The goal is not fixed to any particular moment in time and it in fact stands for an infinite series of independent fixed-time goals of the form `<- first next(n) rotate(L)` for  $n \geq 0$ . The answers to the goal are those answers obtained from independent fixed-time goals. For example, at time 0, **L** is replaced by `[0,1,2,3,4]` (input list), at time 1, **L** is replaced by `[1,2,3,4,0]` (input list rotated once); at time 2, **L** is replaced by `[2,3,4,0,1]` (input list rotated twice), and so on.

## 2.2. CHronolog Execution Model (CHEM)

CHEM is a parallel execution model for Chronolog<sup>4,6,10</sup>. CHEM is based on a data-driven execution model which has a flexible capability to support parallelism at various levels. The proposed execution model is in particular amenable to parallel implementations on multi-processor architectures.

Implementations of Chronolog rely on the underlying resolution-type proof procedure<sup>7</sup>. For efficiency, we combine features of logic programming implementations (unification, backtracking) with features of dataflow implementations (warehousing, tagging) such as those of Lucid implementations<sup>1</sup>. Otherwise it would waste resources recomputing results over and over again.

For instance, suppose we have the following Chronolog program which specifies the predicate **s**, which is true of the running sums of factorials, i.e., it is true of  $1! = 1$  at time 0,  $1! + 2! = 3$  at time 1,  $1! + 2! + 3! = 9$  at time 2,  $1! + 2! + 3! + 4! = 33$  at time 3, and so on.

```
first s(1).
next s(X) <- s(Y), f(Z), i(C), X is Y+Z*(C+1).

first i(1).
next i(X) <- i(Y), X is Y+1.

first f(1).
next f(X) <- f(Z), i(C), X is Z*(C+1).
```

If we are given the following goal (call it  $G_0$ ):

```
<- first next(8) s(N).
```

where `next(k)` stands for *k* successive applications of **next**, and we have the results `first next(7) S(46233)`, `first next(7) f(40320)` and `first next(7) c(8)` which are computed in previous computations and



are stored for reuse, then we can avoid a lot of unnecessary computations. Based on this idea, we propose to employ an associative memory, called warehouse, which is used to store some results of previous computations<sup>4</sup>.

The execution model for Chronolog programs consists of the following four main components:

- **User Interface** It generates sub-goals for a given open-end goal and dispatches them to the interface engine, collects results of independent computations coming from the interface engine and presents the results for the user.
- **Inference Engine** It accepts independent sub-goals from the user interface. Given a sub-goal, it first checks the warehouse to see if any of the temporal atoms are computed before (subject to temporal-matching and unification). It also consults the program store to see if there are any matching program clauses. It reduces the goal and creates choice points in case there are multiply-matching computed tl-atoms and program clauses from the warehouse and from the program store respectively.
- **Program Store** It keeps program clauses, provides temporal-matching to given tl-atoms by inference engine.
- **Warehouse** It provides temporal-matching to a given temporal goal by inference engine. It is responsible for maintaining computed tl-atoms, and it does garbage collection.

An effective warehouse management scheme is required to reduce the amount of space used, without sacrificing much of the efficiency of an implementation. There is an algorithm called warehouse modification algorithm<sup>4</sup>, which can naturally deal with store-demands of computed results as well as the retirement plan based on contexts (moments in time) for Chronolog.

### 2.3. Execution of Chronolog Programs

Given a Chronolog program  $P$  and a goal  $G$ , the general computation steps are as follows:

1. The initial goal is assigned to a computation process  $\text{comp}$  and the process independent child-computations for context-sub-goals of  $G$  are spawned:  
 $\text{comp}_0, \text{comp}_1, \text{comp}_2, \dots$   
 where  $\text{comp}_0$  is the computation at time 0,  $\text{comp}_1$  is the computation at time 1, and so on. Note that, when  $G$  is a fixed-time goal, there is only one computation, that is,  $\text{comp}$ .
2. Simultaneously perform a number of child-computations (context-parallelism).
3. During each computation, there is a conjunction of tl-atoms to be proved. An AND/OR tree is produced.
4. Process the AND/OR tree and search for answers.
5. Go to step 2.

When processing the AND/OR tree, some tl-atom  $A$  from the goal is selected and matched against program clauses or an atom in the warehouse by temporal-matching and unification. Temporal-matching involves the matching of temporal operators in the selected tl-atom  $A$  and a canonical instance<sup>7</sup> of a program clause or a warehouse atom, starting from the top-most clause. Then  $A$  is unified with the head of the temporally-matching program clause. A temporally-matching program clause may be a clause in the *program store* or a computed tl-atom in the *warehouse*. A new goal is produced by replacing the selected temporal atom in the goal by the body of the matching canonical instance and then the substitution (i.e. the variable bindings)

obtained from unification is applied to the new goal. In case there is more than one matching clause, we adopt a standard backtracking mechanism in the operational model.

### 3. Chronolog(Z) as a Simulation Language

Tuzhilin<sup>9</sup> proposes a criteria of a good simulation language as follows: an extensive modeling power so that a wide range of applications can be described in concise terms, a well-defined declarative semantics which can be easily understood by users, and good performance.

Simulation problems deal with processes evolving in time. The notion of time is implicitly built into Chronolog, therefore it is suitable for specifying time-dependent properties of many applications such as simulation problems in a natural way. Chronolog programs have a well-defined declarative meaning and can be easily understood by users. For instance, the program clause

```
next can_eat(pie) <- ready(pie).
```

has the meaning: if a pie is ready currently, then at the next moment in time the pie can be eaten.

Chronolog can be considered a good simulation language. In the rest of this paper, we show that temporal logic language Chronolog(Z), as a simulation language, not only has a strong expressive power for a range of applications and a clear semantics, but also has a good computational efficiency. Now we explain various points of a simulation through several examples.

Consider the following Chronolog(Z) program clauses

```
next light(amber) <- light(green).
next light(red) <- light(amber).
next light(green) <- light(red).
```

which specify a simple traffic light simulation modeled by the time-varying light predicate. These program clauses have a very clear meaning. They correspond to the following rules respectively:

**Rule1** If light is green currently, then the light is amber at the next moment in time.

**Rule2** If light is amber currently, then the light is red at the next moment in time.

**Rule3** If light is red currently, then the light is green at the next moment in time.

Suppose we are given the initial condition: at time 2, the light is red. This fact is represented as the clause

```
first next next light(red).
```

We now have a Chronolog program which consists of three rule clauses and one initial condition clause shown above. For example, suppose that we are given the query

```
<- first next(4) light(Colour).
```

The answer to this query is the answer substitution  $\text{Colour} = \text{amber}$ .

As shown in the above example, a simulated system consists of two components: one is the rule-based component, the other is the initial condition component. In Chronolog, a rule of a simulated system can be represented as a procedure clause and its initial conditions can be represented as facts which are true at the time when the simulation begins. The facts are defined by program clauses with empty bodies.

To describe a simulation system in Chronolog, we introduce a new temporal operator "now<sub>T</sub>":

$$\text{now}_T A = A \wedge (\text{prev } A) \wedge \dots \wedge (\text{prev}(T-1) A) \wedge (\text{prev}(T) \neg A).$$

The informal semantics of the new temporal operator now<sub>T</sub> is:

- now<sub>T</sub> A is true at time  $t \in \mathcal{Z}$  if and only if A is false at the time  $t - T$  and is true from time  $t - T + 1$  to  $t$  inclusively.



Now we consider a Flexible Manufacturing System (FMS)<sup>5</sup>. We assume that an FMS assembles certain products. The initial part of an assembly is brought into the system by the load-unload station. Then it is carried among various manufacturing units, called cells, where assembly processes take place. A special vehicle (V) carries incomplete assemblies among various cells. When the assembly process is completed, the finished units are brought back by vehicles to the load-unload station where they are removed from the FMS system.

To describe the system, we first define the following predicates modelling behavior of the system:

<code>dock(V, C):</code>	a vehicle V is docked at a cell C.
<code>loaded(ASM, V):</code>	an assembly ASM is loaded on a vehicle V.
<code>ready(ASM, C):</code>	an assembly ASM is ready in a cell C.
<code>empty(V):</code>	a vehicle V is empty, i.e. does not carry any assembly.
<code>cell_next(C1, C2):</code>	the next assembly operation is done in cell C2 after the previous assembly operation is done in cell C1.
<code>moving(V, C1, C2):</code>	a vehicle is moving from cell C1 to C2.
<code>travel(C1, C2, T):</code>	it takes T units of time for a vehicle to travel from cell C1 to C2.
<code>process-time(C, T):</code>	it take T units of time to perform an operation in cell C.

Next, we need to collect the rules, which describe the dynamics of the system FMS, and represent them as program clauses. We assume that a cell never processes the same assembly twice. Below are examples of some rules for the system FMS:

**Rule:** If a vehicle V is docked at a cell C with an assembly ASM which has been ready by cell C1 currently and C2 is the next cell which the vehicle moves to, then at the next moment in time move the vehicle from cell C1 to C2.

```
next moving(V, C1, C2) <- dock(V, C1), loaded(ASM, V), ready(ASM, C1), cell_next(C1, C2).
```

**Rule:** If it takes T units of time for a vehicle to travel from cell C1 to C2, and the vehicle V is traveling from cell C1 to C2 currently, then at the next moment in time the vehicle continues to move if the time which has been spent after the vehicle starts the moving from C1 is less than T, or the vehicle arrives at C2 if time is equals to T.

```
next moving(V, C1, C2) <- moving(V, C1, C2), travel(C1, C2, T),  $\neg$  nowT moving(V, C1, C2).
next dock(V, C2) <- travel(C1, C2, T), nowT moving(V, C1, C2).
```

We do not give all details about the system because of space limitations. However, from the above discussion, we can find that the meaning of describing simulations in the language Chronolog is not only clear, but also easy to understand by users. In addition, we are allowed to use negations in the body of a rule clause. Original declarative semantics of Chronolog does not allow to use " $\neg$ ". We extend Chronolog with negation as failure (NAF), so that it becomes more suitable for simulation applications. That is, negated fixed tl-atoms are evaluated using the negation as failure proof rule.

The initial conditions of a simulation system are also defined by clauses but with empty bodies (i.e., facts). For instance, the facts

```
first dock(v1, c2).
first dock(v3, c1).
first prev travel(c1, c2, 15).
first next(3) ready(asm1, c3).
```

are examples of some initial conditions for the FMS system.

#### 4. Implementation

Implementing a simulation system, in general, includes the following steps:

1. Analyse the modeled system and define predicates.
2. Collect rules for the system. For a knowledge-based system, it is important to acquire correct rules which describe the dynamics of the modeled system.
3. Transform the rules collected in step 2 into Chronolog program clauses.
4. Give initial conditions and check if they are consistent with the rules.
5. Run the program, obtain results.

Now consider a more complicated traffic light simulation problem. Suppose there is a traffic control system at the junction of a north-south road and a west-east road. The system consists of several north-south lights and west-east lights, and the time lengths allocated for these lights at different directions and different colours are shown in Table 1. We assume that those lights at the same direction have the same colour at the same time, and synchronously change from one colour to another.

Table 1. Traffic Light

light	green	amber	red
north-south light	3	1	2
west-east light	1	1	4

We define two basic predicates:

`ns_light(X)`: north-south light has X colour;  
`we_light(X)`: west-east light has X colour.

The rules describing the behavior of the system are presented as the following programs clauses:

```
next ns_light(amber) <- now3 ns_light(green).
next ns_light(green) <- ns_light(green), ¬ now3 ns_light(green).
next ns_light(red) <- ns_light(amber).
next ns_light(green) <- now2 ns_light(red).
next ns_light(red) <- ns_light(red), ¬ now2 ns_light(red).

next we_light(amber) <- we_light(green).
next we_light(red) <- we_light(amber).
next we_light(green) <- now4 we_light(red).
next we_light(red) <- we_light(red), ¬ now4 we_light(red).

we_light(red) <- ns_light(green).
we_light(red) <- ns_light(amber).

ns_light(red) <- we_light(green).
ns_light(red) <- we_light(amber).
```



Above rules form a Chronolog program. There is little difference between this program and a standard Chronolog program. That is, this program contains the new temporal operator `nowT` and  $\neg$  in the bodies of some program clauses. But, due to the definition of the new temporal operator, the program can be easily transformed to a program which is like a standard one but with negation.

The rules describe the dynamics of the simulated system. In order to prime a simulation, it is necessary to specify the initial conditions of the simulation system, i.e. the facts that are true at the time when the simulation begins.

In the traffic control simulation, we may have the following initial conditions:

```
first we_light(red).
first prev we_light(amber).
first ns_light(green).
first prev ns_light(red).
```

Under these conditions, if we are given the goal:

```
<- first next(3) we_light(Colour).
```

the answer to this query is `Colour = red`.

Initial conditions may be different, but they should be consistent with the program with respect to certain domain-specific constraints. For instance, the conditions

```
first we_light(green).
first prev we_light(red).
first ns_light(green).
first prev ns_light(green).
```

are not consistent, because they are in conflict with the program clause

```
we_light(red) <- ns_light(green).
```

As shown above, the rule-based component, or the set of the rule clauses of a simulated system, describes the dynamics of the modeled system. The rule clauses make up a Chronolog program, the basic part of the simulation system. In CHEM, a simulation program is represented as individual dataflow graphs, and is stored in the *program store* (see Section 2).

In CHEM, the initial-condition component of the simulated system, i.e. the facts that are true at the time when simulation begins, are stored in the warehouse for priming the simulation.

When a simulation is running, some results of previous computations, which can be also treated as the initial conditions, may be stored in the warehouse for reuse. Therefore all elements in the warehouse can be treated as initial conditions. We have two kind of condition clauses: prime condition and non-prime condition clauses. We call the conditions directly put into the warehouse by users the prime-conditions. We never discard any prime condition clauses from the warehouse. Other conditions are called non-prime conditions. We manage non-prime condition clauses according to the warehouse modification algorithm.

Apart from exploiting various AND-, OR-, and context-parallelism inherent in temporal logic programs, the use of a warehouse facility as an associate memory to store the results of previous computations is an important feature of Chronolog execution model CHEM. The strategy to employ a warehouse can effectively avoid repeated and unnecessary computations.

For example, suppose we are given a goal

```
<- first next(3) we_light(Colour1), first next(4) ns_light(Colour2).
```

At this computation, two child-computations:

```
<- first next(3) we_light(Colour1).
```



and

```
<- first next(4) ns_light(Colour2).
```

are spawned off, both of which can be executed in parallel. At each child-computation, when a sub-goal is given, we first check to see if any of the tl-atoms we need for the computation are in the warehouse. Thus, using the features of CHEM, we can effectively run a simulation system.

## 5. Concluding Remarks

Chronolog is particularly simple and elegant, and Chronolog execution model has an important advantage on the parallel computational efficiency, so our approach to describing and implementing simulation in the model not only has a clear meaning on the expressive power, but also has an effective computational power, making it suitable for simulation applications.

CHEM is a process model for the parallel implementation of Chronolog programs based on dataflow computation. A simulation system consists of a rule-based component and one initial-condition component. Rule clauses make up the basic part of the simulation system. They, as procedure clauses, are stored in the *program store*. These clauses are represented, as abstract dataflow graphs, whose execution is supported by the Chronolog virtual machine. The graph nodes, corresponding to virtual machine instructions, are responsible for clause argument operations, and can be executed once their operands are available so that argument parallelism is exploitable within individual graphs. The initial condition clauses are stored in the *warehouse*. Given a sub-goal, we first check if any of tl-atoms we need are in the warehouse.

CHEM supports argument parallelism through parallel processing of multiple procedure arguments. A new variable binding scheme is used to allow OR-parallelism to be exploited in the distributed dataflow environment. The support of Restricted AND-parallelism can also be incorporated in CHEM by taking advantage of dataflow graph annotation. Context-parallelism can be exploited when a goal is evaluated by executing multiple copies of the goal at different moments in time. This form of parallelism is implemented cost-effectively by using a warehouse to store previous computations for reuse. The dynamic tagging scheme used in the conventional tagged token dataflow machines is naturally suited to the warehouse implementation. It is, therefore, found that dataflow computation lends the CHEM model a flexible capability for support of parallelism at various levels. Therefore, Chronolog execution model is very suitable for knowledge-based simulation applications.

We are currently working on developing general methods to collect rules of a simulated system and intend to work out a formal method to directly transfer the rules of a simulation system into a Chronolog program, so that Chronolog and its execution model become a more useful tool in simulation applications.

## 6. References

1. A. A. Faustini and W. W. Wadge. Intensional programming. Technical Report DCS-55-IR, Department of Computer Science, University of Victoria, Victoria, B.C., Canada, June 1986.
2. R. Goldblatt. *Logics of Time and Computation*. CSLI – Center for the Study of Language and Information, Stanford University, 1987. Lecture Notes no:7.
3. C. Liu, M. A. Orgun, and K. Zhang. A framework for exploiting parallelism in Chronolog. In *Proceedings of The 1st IEEE International Conference on Algorithms and Architectures for Parallel Processing*, Brisbane, Australia, 19–21 April 1995. IEEE Press.
4. C. Liu, M. A. Orgun, and K. Zhang. A parallel execution model for Chronolog. Technical Report 95-166C, Department of Computing, Macquarie University, Sydney, NSW 2109, Australia, February 1995.
5. T. I. Oren. Bases for advanced simulation: Paradigms for the future. In T. I. Oren M.S. Elas and B.P. Zeigler, editors, *Modelling and Simulation Methodology*, pages 29–43, Gustav Stresemann Institut, Bonn, Germany, 1989. North-Holland.

6. M. A. Orgun and W. W. Wadge. Theory and practice of temporal logic programming. In L. Fariñas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 23–50. Oxford University Press, 1992.
7. M. A. Orgun and W. W. Wadge. Chronolog admits a complete proof procedure. In *Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 120–135, Université Laval, Québec City, Québec, Canada, April 26–27 1993.
8. N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, 1971.
9. Alexander Tuzhilin. Simtl: A simulation language based on temporal logic. *TRANSACTIONS*, 9(2):86–99, 1992.
10. W. W. Wadge. Tense logic programming: a respectable alternative. In *Proceedings of the 1988 International Symposium on Lucid and Intensional Programming*, pages 26–32, Sidney, B.C., Canada, April 7–8 1988.