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.mpce.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 the 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 and succinct 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 execution model of Chronolog, 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 growing interest in knowledge-based simulation methods. Ören⁶ 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.

Simulation methods deal with processes evolving in time, and knowledge-based simulation methods based on classical first-order logic should therefore provide an explicit support for time. Since temporal logic can naturally model time-dependent dynamic properties of certain problems of the real world, we can select a certain type of "executable temporal logic" as a language to describe simulation problems.

Chronolog⁷ is a temporal extension of logic programming, based on a linear-time temporal logic² with an unbounded future in which the set of natural numbers models the collection of moments in time. Chronolog(Z)⁸ extends
Chronolog with a linear-time temporal logic with an unbounded past and future. Chronolog programs can be executed using a parallel execution model, called CHEM (CHronolog Execution Model)\textsuperscript{12,3,4}. CHEM is a data-driven execution model, which has a flexible capability to support parallelism at various levels. 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 tasks.

A simulated system consists of a rule-based component and an initial-condition component. In CHEM, 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, that is, 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 a blackboard facility called the warehouse for reuse. When the results are needed again, they can be retrieved from the warehouse. Therefore the use of 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. 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 Chronolog, including its logical basis and an abstract form of its execution model which we call CHEM.

2.1. Chronolog Programs

Temporal logic\textsuperscript{9} can be regarded as a special case of modal 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 \( \mathbb{Z} \) of integers models the collection of moments in time.

The underlying temporal logic of Chronolog, denoted as \( TL \), refers to a natural extension of a standard first-order logic language. 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.\(^7\)\(^8\).

The informal semantics of temporal operators first, prev and next are as follows: Let \( t \in \mathbb{Z} \). A formula of the form 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. A tl-atom is basically an atomic formula with a number of applications of temporal operators.

Chronolog programs look like standard logic programs with the only difference being that program clauses are made up of tl-atoms.

For example, consider the temporal logic program shown in figure 1. It specifies the rotate predicate to generate all possible rotations of a given list. In the program, first rotate(L), first (^input(L)), next rotate(L) etc. are all tl-atoms. Time-dependencies in the program are specified through the use of temporal operators. Program clauses are regarded as assertions true at all moments in time.

\[
\begin{align*}
\text{first rotate(L) } &<-
\quad \text{first } (^\text{input(L)}) , \\
\text{next rotate(L) } &<-
\quad \text{rotate([H|T])} , \\
&\quad \text{append(T, [H], L)} . \\
\text{append([], L, L)} . \\
\text{append([X|L1], L2, [X|L3]) } &<-
\quad \text{append(L1, L2, L3)} .
\end{align*}
\]

Fig. 1. List Rotation Program

In the program, the first clause says that the initial value of the list to be rotated is provided from the standard input. The symbol \(^\sim\) 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. The last two clauses define a standard append predicate.

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, 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 operators 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 example in figure 1, we consider the fixed-time goal \(-\text{first rotate}(L)\). 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 that 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 \([0, 1, 2, 3, 4]\).

In temporal logic programs, non-terminating computations are initiated by open-end goals. For instance, the goal \(-\text{rotate}(L)\) is 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 \(-\text{first next}(n) \text{ rotate}(L)\) for \(n \geq 0\). Here next\((n)\) stands for \(n\) successive applications of next. 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.

A non-terminating computation in the negative direction would not produce any results, because the rotate predicate is defined by an initial clause and a program clause with recursive past dependencies.

2.2. CHronolog Execution Model

CHronolog Execution Model (CHEM)\(^{12,4}\) is a parallel execution model for
the language. CHEM is based on a data-driven execution model which has a
flexible capability to support parallelism at various levels. It is in particular
amenable to parallel implementations on multi-processor architectures.

The correctness of implementations of Chronolog relies on the underlying
resolution proof procedure8. For efficiency, we combine features of logic pro-
gramming implementations (unification, backtracking) with features of
dataflow implementations (warehousing, tagging) such as those of Lucid
implementations1. Otherwise it would waste resources recomputing results
over and over again.

\[
\begin{align*}
&\text{first } s(1). \\
&\text{next } s(X) \leftarrow s(Y), f(Z), i(C), X \text{ is } Y+Z*(C+1).
\end{align*}
\]

\[
\begin{align*}
&\text{first } i(1). \\
&\text{next } i(X) \leftarrow i(Y), X \text{ is } Y+1.
\end{align*}
\]

\[
\begin{align*}
&\text{first } f(1). \\
&\text{next } f(X) \leftarrow f(Z), i(C), X \text{ is } Z*(C+1).
\end{align*}
\]

\textbf{Fig. 2. Running Sums of Factorials}

For instance, suppose we have the Chronolog program shown in figure 2.
It specifies the predicate s. The predicate s is true of the running sums of
factorials, that is, 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.

CHEM uses a warehouse facility to store results of previous computations12.4.
Suppose that we are given the following goal (call it Go):

\[
\begin{align*}
\leftarrow & \text{first next}(8) \ s(N).
\end{align*}
\]

If we already have the results first next(7) \(s(46233)\), first next(7) \(f(40320)\) and first next(7) \(c(8)\) which are computed in previous com-
putations and are stored in the warehouse with their time-tags, then we can
avoid a lot of redundant computations.

In an abstract form, the execution model of Chronolog consists of the following
four main components:

- **User Interface** generates sub-goals for a given open-end goal and dis-
  patches them to the inference engine, collects results of independent com-
  putations coming from the inference engine, and presents the results for
  the user.
- **Inference Engine** accepts independent sub-goals from the user interface. Given a sub-goal, it first checks the warehouse to see if any of the tl-atoms in the sub-goal 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** keeps program clauses, provides temporal-matching to given tl-atoms by the inference engine.

- **Warehouse** provides temporal-matching to a given temporal goal by the 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\(^4\), which can naturally deal with store-demands of computed results as well as the retirement plan based on contexts (moments in time). There is no harm in throwing away a computed tl-atom from the warehouse, because it can be recomputed should a need arise.

### 2.3. Execution of Chronolog Programs

Chronolog program computations at different moments time, i.e. in different contexts, can be executed in parallel. This mode of parallelism which we call context-parallelism is an intrinsic property of temporal logic programming. Any open-end goal stands for an infinite series of independent fixed-time goals. Therefore, in principle, any number of fixed-time goals can be executed in parallel.

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:

   \[
   \text{comp}_0, \text{comp}_1, \text{comp}_2, \text{comp}_3, \ldots
   \]

   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. 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 a tl-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\(^8\) of a program clause starting from the top-most clause or a warehouse atom. 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.

3. Chronolog(Z) as a Simulation Language

Tuzhilin\(^9\) proposes the following criteria of a good simulation language: (1) an extensive modeling power so that a wide range of applications can be described in concise terms, (2) a well-defined declarative semantics which can be easily understood by users, and (3) 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

$$\text{next can\textunderscore eat(pie)} \leftarrow \text{ready(pie)}.$$  

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

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

$$\text{next light(amber)} \leftarrow \text{light(green)}.$$  
$$\text{next light(red)} \leftarrow \text{light(amber)}.$$
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:

**Rule 1** If light is green currently, then the light will turn amber at the next moment in time.

**Rule 2** If light is amber currently, then the light will turn red at the next moment in time.

**Rule 3** If light is red currently, then the light will turn green at the next moment in time.

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

\[
\text{first next next light(red).}
\]

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

\[
<- \text{first next(4) light(Colour).}
\]

The answer to this query is the answer substitution Colour = 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.

For the next example, we first introduce a new temporal operator "now\(_T\)" into Chronolog:

\[
\text{now}_T A = A \land (\text{prev} A) \land \ldots \land (\text{prev}(T-1) A) \land (\text{prev}(T) \neg A).
\]

The informal semantics of the new temporal operator now\(_T\) is:

- now\(_T\) A is true at time \(t \in \mathcal{E}\) 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)\(^{10}\). 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:

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

Next, we need to collect the rules, which describe the dynamics of the FMS system, 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 FMS system (we do not give all details about the system because of space limitations):

**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.

```prolog
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 then T, or the vehicle arrives at C2 if time is equals to T.

\[
\begin{align*}
\text{next moving}(V, C1, C2) & \leftarrow \\
\text{moving}(V, C1, C2), \\
\text{travel}(C1, C2, T), \\
\text{now} \_T \text{ moving}(V, C1, C2).
\end{align*}
\]

\[
\begin{align*}
\text{next dock}(V, C2) & \leftarrow \\
\text{travel}(C1, C2, T), \\
\text{now} \_T \text{ moving}(V, C1, C2).
\end{align*}
\]

We are allowed to use negations in the body of a rule clause. The original declarative semantics of Chronolog does not allow the use of “\_” in the bodies of porogram clauses and goals. We extend Chronolog with negation as failure (NAF\(^5\)), 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

\[
\begin{align*}
\text{first dock}(v1, c2). \\
\text{first dock}(v3, c1). \\
\text{first prev travel}(c1, c2, 15). \\
\text{first next}(3) \text{ ready}(as1, c3).
\end{align*}
\]

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 any 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 traffic light simulation problem. 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
Table 1. Traffic Light

<table>
<thead>
<tr>
<th>light</th>
<th>green</th>
<th>amber</th>
<th>red</th>
</tr>
</thead>
<tbody>
<tr>
<td>north-south light</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>west-east light</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

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.

We define two basic predicates:

\[
\text{ns.light}(X): \text{ north-south light has X colour;}
\]
\[
\text{we.light}(X): \text{ west-east light has X colour.}
\]

The rules describing the behavior of the system are shown in figure 3.

There is little difference between the program in figure 3 and a standard Chronolog program. The program contains the new temporal operator now 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 rule-based component, or the set of the rule clauses of a simulated system, describes the dynamics of the modeled system. In order to prime a simulation, it is necessary to specify the initial conditions of the simulation system, that is, the facts that are true at the time when the simulation begins. The rule clauses together with the initial conditions make up a Chronolog program.

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

\[
\text{first we.light(red).}
\]
\[
\text{first prev we.light(amber).}
\]
\[
\text{first ns.light(green).}
\]
\[
\text{first prev ns.light(red).}
\]

Under these conditions, if we are given the goal:

\[
<- \text{first next(3) we.light(Colour).}
\]
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).

---

Fig. 3. Traffic Light Simulation

the answer to this query is Colour = red.
Initial conditions should be consistent with the program with respect to certain domain-specific constraints. For instance, the conditions

\[
\begin{align*}
\text{first we.light(green).} \\
\text{first prev we.light(red).} \\
\text{first ns.light(green).} \\
\text{first prev ns.light(green).}
\end{align*}
\]

are not consistent with the program. The we.light predicate should be single-valued at any given moment in time (we cannot have that it is both green and red). So, the conditions are in conflict with the clause

\[
\begin{align*}
\text{we.light(red) <- ns.light(green).}
\end{align*}
\]

In CHEM, a simulation program is represented as individual dataflow graphs and stored in the program store (see Section 2). The initial-condition
component of the simulated system 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.

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.

Using CHEM, we can exploit AND-, OR-, and context-parallelism inherent in temporal logic programs in simulation tasks. Therefore the Chronolog execution model also offers an important advantage on the parallel computational efficiency. For example, suppose we are given a goal

\[
\begin{align*}
\text{<- first next(3) we.light(Color1),} \\
\text{first next(4) ns.light(Color2).}
\end{align*}
\]

At this computation, two child-computations:

\[
\begin{align*}
\text{<- first next(3) we.light(Color1).}
\end{align*}
\]

and

\[
\begin{align*}
\text{<- first next(4) ns.light(Color2).}
\end{align*}
\]

are spawned off, both of which can be executed in parallel (AND-parallelism). 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. Therefore the use of the warehouse offers extra potential for OR-parallelism.

5. Concluding Remarks

In Chronolog, 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 are stored in the program store represented as abstract dataflow graphs, whose execution is supported by the Chronolog virtual machine\textsuperscript{12,4}. 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. Initial condition clauses are stored 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. 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.

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 and to check the consistency of initial conditions automatically, so that Chronolog and its execution model become a more useful tool in simulation applications. In short, we believe that our approach to describing and implementing knowledge-based simulation applications in Chronolog not only has a clear meaning on the expressive power, but also has an effective computational power.

6. Acknowledgements

The work presented in this paper has been supported in part by an Australian Research Council (ARC) small grant. Chu chang Liu has been supported by an Australian Postgraduate Award (APA).

7. References


