A Structural Knowledge-Based Simulation Methodology for Distributed Systems

Chuchang Liu **Mehmet A. Orgun**

Department of Computing, Macquarie University, NSW 2109, Australia
E-mail: {cliu,mehmet}@krakatoa.mpce.mq.edu.au

Abstract

This paper presents a structural knowledge-based simulation methodology based on temporal logic that can be used for building a simulation environment for distributed computations. The main contributions of the paper are: (1) to propose a structural knowledge representation method suitable for describing simulation systems; (2) to provide several techniques for specifying distributed computations, including a knowledge acquisition technique and a constraint mechanism used for eliciting knowledge about the simulated system; and (3) to provide an approach to knowledge-based simulation management.

Introduction

Simulation requires models of a process to be simulated and it deals with processes evolving in time. The notion of time is implicitly built into temporal logic languages, which is essential in simulation. They can also provide support for rule-based and object-oriented paradigms and for powerful knowledge representation schemes. Therefore we claim temporal extensions of logic programming are very suitable for knowledge-based simulation applications. there are a number of programming languages based on temporal logic, such as Templog (Abadi & Manna 1989), Tempura (Moszkowski 1986) and Chronolog(MC) (Liu & Orgun 1995). The language Chronolog(MC) is based on the temporal logic TLC (Liu & Orgun 1996a), which is an extension of linear-time temporal logic with granularity of time. In this logic, all formulae can be clocked and are allowed to be defined on local clocks. TLC is therefore more flexible in describing the behavior of those systems where dynamic changes are essential (Liu & Orgun 1996b).

This paper presents a structural knowledge-based simulation methodology based on TLC and Chronolog(MC), which can be used for building a simulation environment for distributed computations. The first contribution of the paper is to provide a structural knowledge representation method for describing simulation systems. We develop a modeling formalism to represent the initial structural representation of a system as well as a method to use in determining the next representation of the structure. The second contribution is to provide several techniques for specifying distributed computations, including a knowledge acquisition technique and a constraint mechanism used for eliciting knowledge in simulating distributed systems. Constraint mechanism is also well-structured. The third contribution is to provide an approach to knowledge-based simulation management. This methodology provides an integration of modeling formalism, modeling environment, knowledge-base management and processing of the model in simulation.

The structure of the paper is as follows. After a brief introduction to the logic TLC and the language Chronolog(MC), we outline the framework of our knowledge-based simulation methodology. Then a structural knowledge representation method suitable for describing simulation systems is discussed. We also discuss specification techniques for constructing knowledge bases in simulating distributed computations and our approach to knowledge-based simulation management. Finally, we conclude the paper with a brief discussion.

Temporal Logic TLC

We first give a brief introduction to the temporal logic TLC, including the definition of local clocks, its axioms and inference rules (Liu & Orgun 1996a). We also introduce the language Chronolog(MC), which is an extension of logic programming based on TLC.

Clocks, syntax The global clock (denoted as gck) is the increasing sequence of natural numbers: $\langle 0, 1, 2, \ldots \rangle$, and a local clock is a subsequence of the global clock. In other words, a local clock is a strictly increasing sequence of natural numbers, either infinite or finite: $\langle t_0, t_1, t_2, \ldots \rangle$. Let $\mathcal{CK}$ be the set of all clocks and $\sqsubseteq$ be an ordering relation on the elements of $\mathcal{CK}$. For any

$ck_1, ck_2 \in \mathcal{CK}$, we define that

$$ck_1 \sqcap ck_2 \stackrel{\text{def}}{=} g.l.b.\{ck_1, ck_2\}$$
$$ck_1 \sqcup ck_2 \stackrel{\text{def}}{=} l.u.b.\{ck_1, ck_2\}$$

where $ck_1, ck_2 \in \mathcal{CK}$, *g.l.b.* stands for "the greatest lower bound" and *l.u.b.* for "the least upper bound" under the relation $\sqsubseteq$.

In TLC, there are two temporal operators, `first` and `next`, which refer to the initial and the next moment in time with respect to given clocks respectively. TLC formulas are constructed by the following rules: (1) Any formula of first-order logic is a formula of TLC; (2) if A is a formula, so are `first` A and `next` A.

We use a clock assignment to assign a local clock for each predicate symbol. A clock assignment ck is a map from the set of predicate symbols to the set of clocks. The clock associated with a predicate symbol p is denoted by $ck(p)$.

Let A be a formula and ck a clock assignment. The local clock associated with a formula A over ck, denoted as ck_A, is defined inductively as follows:

- If A is an atomic formula of the form $p(x_1, \ldots, x_n)$, then $ck_A = ck(p)$.
- If $A = \neg B$, `first` B or $(\forall x)B$ then $ck_A = ck_B$.
- If $A = B \wedge C$, then $ck_A = ck_B \sqcap ck_C$.
- If $A =$ `next` B, then (1) $ck_A = \langle t_0, t_1, \ldots, t_{n-1}\rangle$ when $ck_B = \langle t_0, t_1, \ldots, t_n\rangle$ is non-empty and finite; (2) $ck_A = ck_B$ when ck_B is infinite or empty.

When ck_A is finite, $ck_{\mathbf{next}A}$ is generated just by deleting last element in ck_A because the last element t_n does not have a next moment defined for it.

Given a local clock $ck_i = \langle t_0, t_1, t_2, \ldots\rangle$, we define the rank of t_n on ck_i to be n, written as $rank(t_n, ck_i) = n$. Inversely, we write $t_n = ck_i^{(n)}$, which means that t_n is the moment in time on ck_i whose rank is n. Obviously, for any given formula A, it $t \in ck_{\mathbf{next}A}$, then we have that $rank(t, ck_A) = rank(t, ck_{\mathbf{next}A})$.

In TLC, the semantics of formulas with logical connectives are defined in the usual way, but with respect to local clocks (Liu & Orgun 1996a). Here we only give the meaning of temporal operators:

- For any $t \in ck_A$, `first`A is true at t if and only if A is true at $ck_A^{(0)}$.
- For any $t \in ck_{\mathbf{next}A}$, `next` A is true at t if and only if A is true at $ck_A^{(i+1)}$, where $i = rank(t, ck_A)$.

Axioms and Inference Rules In TLC, all theorems of the form $\vdash A$ are assumed to hold on the local clock associated with the formula A. The axioms and rules of inference of TLC are summarized below:

A1. $\vdash$ `first first` $A \leftrightarrow$ `first` A.

A2. $\vdash$ `next first` $A \leftrightarrow$ `first` A.

A3. $\vdash$ `first` $(\neg A) \leftrightarrow \neg($`first` $A)$.

A4. $\vdash$ `next` $(\neg A) \leftrightarrow \neg($`next` $A)$.

A5. $\vdash$ `first` $(\forall x)(A) \leftrightarrow (\forall x)($`first` $A)$.

A6. $\vdash$ `next` $(\forall x)(A) \leftrightarrow (\forall x)($`next` $A)$.

A7. $\vdash$ `first`$(A \wedge B) \leftrightarrow($`first` $A) \wedge ($`first` $B)$, when $ck_A^{(0)} = ck_B^{(0)}$.

A8. $\vdash$ `next` $(A \wedge B) \leftrightarrow$ $($`next` $A) \wedge ($`next` $B)$, when $ck_A = ck_B$.

There are three inference rules:

R1. If $\vdash A \rightarrow B$ and $\vdash A$, then $\vdash B$, when $ck_A = ck_B$.

R2. If $\vdash A$, then $\vdash$ `first` A, when ck_A is non-empty.

R3. If $\vdash A$, then $\vdash$ `next` A, when $ck_{\mathbf{next}A}$ is non-empty.

Chronolog(MC) A Chronolog(MC) program consists of a clock definition, a clock assignment and a program body. The program body consists of rules and facts. The clock definition is an ordinary Chronolog program and it specifies all the local clocks involved in the program body. All the predicates specified in the clock definition are defined over the global clock. The clock assignment assigns clocks for all the predicate symbols appearing in the program body.

The following simple Chronolog(MC) program specifies a computational system consisting of two independent processes p and q running on their own local clocks. Note that program clauses are interpreted as assertions true at all moments in time on their local clocks.

```
% CLOCK DEFINITION (ck1, ck2) %
first ck1(0).
next ck1(N) <- ck1(M), N is M+2.
first ck2(1).
next ck2(N) <- ck2(M), N is M+3.

% CLOCK ASSIGNMENT (ck) %
is_ck(p,ck1).
is_ck(q,ck2).

% PROGRAM BODY %
first p(0).
next p(X) <- p(Y), X is Y+2.
first q(2).
next q(X) <- q(Y), X is Y*Y.
```

In the clock definition, the first clause for `ck1` states that the first value for `ck1` is 0. The second clause states that the subsequent values for `ck1` are determined by the previous value of `ck1` plus 2. Therefore, `ck1` models the sequence of even numbers over global time. The rest of the clauses in the clock definition can be explained in a similar manner. The clock assignment says that the clock of predicate p is `ck1` and the clock of predicate q is ck2. The predicate p represents the sequence of even numbers over the clock modeled by `ck1`; the predicate q represents the sequence $\langle 2^0, 2^2, 2^4, 2^8, \ldots \rangle$ over the clock modeled by ck2.

Now we may pose the following fixed-time goal (goals within the scope of the operator `first` are called fixed-time): `<- first (p(X), q(Y))`.

By the clocked temporal resolution method (Liu & Orgun 1996a), the answer is "X=4 and Y=4". The conclusion can be obtained by the following idea: Due to the fact that ck1=$\langle 0, 2, 4, 6, \ldots \rangle$ and ck2=$\langle 1, 4, 7, 10, \ldots \rangle$, the local clock of the above goal is the greatest lower bound of ck1 and ck2, that is, $\langle 4, 10, \ldots \rangle$. Obviously, both the predicates p(X) and q(y) are defined at each moment on the clock $\langle 4, 10, \ldots \rangle$. Therefore, to answer the above query, we need to obtain answers to the following two queries on their own clocks ck1 and ck2:

```
<- first next next p(X).
<- first next q(Y).
```

Then, the answer to the first query is "X=4", and the answer to the second query is "Y=4".

Simulation Framework

TLC provides a powerful mechanism to describe state and action changes with its temporal operators. Therefore, it can not only be used for representing declarative and procedural knowledge, but also can be used for representing intensional knowledge. In particular, it can be used for reasoning about the properties of a simulation system modelled by a Chronolog(MC) program.

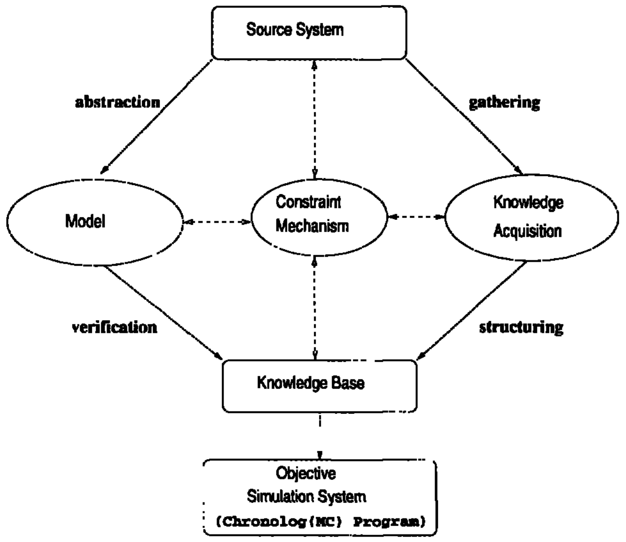

Figure 1: Knowledge-based simulation methodology

Figure 1 illustrates the main components of our methodology for simulating distributed systems. For a given source system, a corresponding model can be obtained through knowledge abstraction; knowledge acquisition is embarked upon through knowledge gathering based on the analysis of the source description of the system. To obtain a satisfactory knowledge base, which is essential in behavioral simulation, the following activities are usually involved: knowledge classification and structuring, verification and consistency checking, and updating the knowledge base. Constraint mechanism can be used in the process of knowledge acquisition to elicit knowledge for completing a knowledge specification through testing and verifying its consistency. Once a satisfactory knowledge base for the system is obtained, it is then translated into an executable program of Chronolog(MC).

Modeling Formalisms

In simulation activities (Ören 1987), a dynamic model is a model with time-indexable behavior, which is driven by investigating behavior change. There are two kinds of behavior: (1) *trajectory behavior* consists of a time-indexed sequence of values for some variables of the model, and (2) *structural behavior* is a time-indexed sequence of the representation of its structure. In both trajectory and structural simulation, one needs to specify the representation of the state and the state transition function. In trajectory simulation, we in particular have to specify the representation of state variables and state transition function; in structural simulation, we need to specify the representation of the model, and rules to generate the next representation of the model.

The development of modeling formalisms to represent initial representation of the structure of a system as well rules to be used in determining the next representation of the structure may facilitate structural simulation. We now present knowledge-based models based on both trajectory and structural simulation to simulate distributed computations with Chronolog(MC).

A distributed computation can be described by the execution of a distributed program which is performed by a set of processes. Therefore, from the point of view of structural simulation, it consists of a collection of processes and their local clocks.

To describe the trajectory behavior of the processes of the system, we need to define state predicates, control predicates as well as channel state predicates. We also need define a local clock for each predicate. Therefore, the behavior of a system is described through the specifications of state predicates, control predicates and channel state predicates, which are related with the computation. Thus, the behavior of the system is described by the rules which describe the state changes and the control of the system.

In modelling distributed computations, we first want to construct domain-specific knowledge bases. In our method, the initial knowledge base of a given simulation system is denoted by a 5-tuple:

$$\mathcal{KB} = < \mathcal{V}, \mathcal{P}, ck, \mathcal{R}, \mathcal{I} >,$$

where $\mathcal{V}$ is a set of variables, $\mathcal{P}$ is a set of predicates, ck is a clock assignment, a map from the set $\mathcal{P}$ of predicate symbols to the set $\mathcal{CK}$ of clocks, $\mathcal{R}$ is a set of rules which describe the behavior of each component and interactions between components, and $\mathcal{I}$ is a set of initial conditions represented as facts which are all true at the initial moment in time when the system starts.

The rules and initial conditions are expressed as Chronolog(MC) program clauses. Here are examples:

```
next get_service(X) <- queue(Q), head(X,Q).
first queue([tom]).
```

The first clause says that if at the current moment in time X is the first person in a queue waiting for service, then at the next moment X will get service; the second clause says that at the initial moment Tom is the first person in the queue.

Specifying Distributed Systems

Computation models Informally, a distributed computation describes the execution of a distributed program by a set of processes. The activity of each sequential process is modeled as executing a sequence of events (or instructions). An event may be either internal to a process and cause only a local state change, or it may involve communication with other processes.

There are two models of communication: the synchronous communication model and the asynchronous communication model. In both of the models, we need to address the following requirements: (1) The variables of each process are private, i.e., not accessible to other processes. (2) Message passing is used to provide interaction between processes. In the synchronous communication model, explicit send-receive primitives are used, the messages are not buffered, and communication is performed by a simultaneous execution of a send-receive pair. In the asynchronous communication model, the message exchange is not synchronised, and communication requires buffering for the messages that have been sent but not yet received.

We limit our discussion to the weakest possible model for an asynchronous distributed system characterized by the properties. There exist no bounds on the capacity of buffers as well as the capacity of channels, and there exist no bounds on message delays. The form of distributed programs we consider is as follows:

parallel:	$P : [\overline{Y}_1; P_1 \parallel \ldots \parallel \overline{Y}_m; P_m]$
sequential:	$P_i : \{l^i_0; \ldots ; l^i_{k_i}\},\ i = 1, \ldots, m$

where $P_1, \ldots,$ and P_m are parallel processes of the program P, which can be executed concurrently and the individual processes are sequential programs; $\overline{Y}_1 \cup \ldots \cup \overline{Y}_m = \overline{Y}$ is a set of data variables, and $\overline{Y}_1, \ldots, \overline{Y}_m$ are disjoint subsets of data variables private to the processes $P_1, \ldots,$ and P_m, respectively; $l^i_0, \ldots, l^i_{k_i}$ are simple instructions.

The types of instructions include *skip*, *assignment* and *send* and *receive*. The *send* and *receive* instructions are used to pass messages between processes.

skip:	*skip*
assignment:	$X := e$
receive:	$c; \alpha \uparrow X$
send:	$c; \alpha \downarrow e$

Here c is a boolean expression, e is an expression and X is a variable, and α is a channel name. The first two types of instructions place no restrictions on execution; a process whose next command is *skip* or an assignment instruction is ready and can be executed. The *receive* instruction can be performed only when c is true and it then attempts to receive a value from channel α for variable X. The *send* instruction can be performed only when c is true and it then attempts to send the value of e along channel α.

Processes communicate by sending and receiving messages along *channels*. Each channel connects a sending process and a receiving process. We use the notation $ch_{P_j-P_i}$ to denote the channel that connects the sending process P_i and the receiving process P_j.

Specification Techniques We propose the following techniques in the specification of distributed computations: (1) a knowledge acquisition technique based on the structural representation method; and (2) a constraint mechanism used for eliciting knowledge about the simulated distributed system.

Given a distributed computation as above, its initial structural representation is a knowledge base of the form $\mathcal{KB} = < \mathcal{V}, \mathcal{P}, ck, \mathcal{R}, \mathcal{I} >$, that can be obtained through the analysis of the system itself and its implementation environment.

The set $\mathcal{V}$ consists of all private variables of all processes and their control variables and the variables representing the status of the channels. For each process P_i ($i = 1, 2, \ldots, m$), we may need to define several predicates which are local to the process and several predicates to describe the status of the channels related to the process. Thus, the set $\mathcal{P}$ contains process state predicates, such as $\mathtt{state\_P}_i\mathtt{(U_1, \ldots, U_{i_k})}$, process control predicates, such as $\mathtt{cl\_P}_i\mathtt{(S)}$, assignment predicates, such as $\mathtt{assign\_p}_i\mathtt{(U,E)}$, channel state predicates, such as $\mathtt{ch}_{P_j-P_i}\mathtt{(L)}$, and channel value change predicates, such as $\mathtt{in\_change\_ch}_{P_j-P_i}\mathtt{(L,E)}$ and $\mathtt{out\_change\_ch}_{P_j-P_i}\mathtt{(L,E)}$.

States of process P_i are assignments of values from the appropriate domains to variables $\mathtt{U}_1, \ldots, \mathtt{U}_{i_k}$. The control predicate $\mathtt{cl\_P}_i\mathtt{(L)}$ means that the control of the statement execution in process P_i has been moved to the instruction whose label is S, in other words, the process is executing the instruction S. $\mathtt{ch}_{P_j-P_i}\mathtt{(L)}$ means that the value of channel $ch_{P_j-P_i}$ is the list L. $\mathtt{in\_change\_ch}_{P_j-P_i}\mathtt{(L,E)}$ means that the value of channel $ch_{P_j-P_i}$ is changing by incoming data E and $\mathtt{out\_change\_ch}_{P_j-P_i}\mathtt{(L,E)}$ means that the value of channel $ch_{P_j-P_i}$ is changing by outgoing data E.

A predicate related to communications may be involved in more than one process, and those processes may have different "local times". Therefore, such predicates should be investigated from a global point of view, and, therefore, they may be defined on the global clock. The other predicates may be defined on local clocks based on the environment provided. For the above predicates, $\texttt{state\_p}_i$, $\texttt{cl\_P}_i$ and $\texttt{assign\_p}_i$ may be defined on the local clock of process P_i. The rest of the predicates are defined on the global clock *gck*.

The allocation of time for each process determines the clocks associated with the predicates. Therefore, the component *ck* of the specification depends on the environment of the distributed computation.

The rules describing the behavior of the computation can be automatically produced based the instructions of each process. The initial conditions represent the initial states of the system, including initial values of variables, and the initial states of control and channels.

To improve the quality and reliability of the knowledge base as a specification of a distributed system in simulating it, we propose a constraint mechanism used for eliciting knowledge in the knowledge acquisition process (Liu & Orgun 1997). Due to the well-structural representation of knowledge bases, the constraint mechanism is also well-structured. Constraints include: variable constraints, predicate generation constraints, clock constraints, and time-dependent relation constraints. Liu and Orgun (1997) have given a general discussion on these types of constraints. We now discuss how to formalize and use time-dependent relation constraints to obtain the set of rules $\mathcal{R}$ and the initial condition set $\mathcal{I}$ in a knowledge base.

The rules come from the instructions of the system that we want to specify. There are two kinds of instructions: one has a simple action, such as *skip* and *assignment* instructions; the other has an action with a condition, which are involved in message communication, such as the instructions *send* and *receive*.

We have the following time-dependent relation constraints on the execution of instructions to obtain rules describing the behavior of a distributed system.

- If the process p is currently executing L which is a single action, then it will execute the next instruction at the next moment in time.
- If the process p is currently executing L which is an action with a condition c, then it will execute the next instruction at the next moment in time when c is true, otherwise it waits until c becomes true.

These constraints can be formalized by the following rules which describe the changes of control:

```
cl_p(l) → next cl_p(l').
cl_p(l) ∧ c is true → next cl_p(l').
cl_p(l) ∧ ¬(c is true) → next cl_p(l).
```

Here l' is the next instruction which will be executed.

The rules that describe the changes of process states and channel states can be obtained based on individual instructions. For each kind of an instruction, we have a standard rule pattern. For instance, the rule pattern

$$\begin{aligned}&\texttt{cl\_P}_i(l)\wedge\texttt{state\_p}_i(\texttt{U}_1,\ldots,\texttt{U}_v,\ \ldots,\texttt{U}_{i_k})\rightarrow\\&\qquad\texttt{next state\_P}_i(\texttt{U}_1,\ldots,\texttt{E},\ldots,\texttt{U}_{i_k}).\end{aligned}$$

is for assignment instructions of the form l: $\texttt{U}_v$ = E.

For message communication, we have the following rule patterns:

(1) *Send message*: l: $c; ch_{P_i-P_j} \downarrow E$

$$\begin{aligned}&\texttt{cl\_P}_i(l)\wedge\texttt{true(c)}\wedge\texttt{ch}_{P_j-P_i}\texttt{(L)}\rightarrow\\&\qquad\texttt{in\_change\_ch}_{P_j-P_i}\texttt{(L,E)}.\\&\texttt{ch}_{P_j-P_i}\texttt{(L1)}\wedge\texttt{in\_change\_ch}_{P_j-P_i}\texttt{(L,E)}\wedge\texttt{L1=L*E}\rightarrow\\&\qquad\texttt{next ch}_{P_j-P_i}\texttt{(L1)}.\end{aligned}$$

(2) *Receive message*: l: $c; ch_{P_i-P_j} \uparrow U$

$$\begin{aligned}&\texttt{cl\_P}_i(l)\wedge\texttt{true(c)}\wedge\texttt{ch}_{P_i-P_j}\texttt{(L)}\wedge\texttt{E=head(L)}\rightarrow\\&\qquad\texttt{out\_change\_ch}_{P_i-P_j}\texttt{(L,E)}.\\&\texttt{out\_change\_ch}_{P_i-P_j}\texttt{(L,E)}\wedge\texttt{L1=tail(L)}\rightarrow\\&\qquad\texttt{next ch}_{P_i-P_j}\texttt{(L1)}.\\&\texttt{cl\_P}_i(l)\wedge\texttt{out\_change\_ch}_{P_i-P_j}\texttt{(L,E)}\wedge\\&\texttt{state\_P}_i(\texttt{U}_1,\ldots,\texttt{U},\ldots,\texttt{U}_{i_k})\rightarrow\\&\qquad\texttt{next state\_P}_i(\texttt{U}_1,\ldots,\texttt{E},\ldots,\texttt{U}_{i_k}).\end{aligned}$$

Here we assume that the time for executing each instruction is 1 unit of time. Note that there are several new predicates such as `head(L)`, `tail(L)` and `true`, whose meanings are obvious. Because of space limitations, we omit their explanation.

All rules describe the dynamics of the computation, including the changes of control and the changes of process states and channel states. These rules form the set $\mathcal{R}$ under the structural representation.

The set $\mathcal{I}$ of initial conditions includes the facts that are true at the initial moment. They include the initial state of each process which is represented by its state predicate and the initial control state. For example, we may have the initial conditions as follows: At the beginning, the control of the process p_1 has been moved to the instruction l_1, the initial value of the state variable X of the process p_1 is 0, and the empty list is the initial value of `ch1`. They can be formally represented as $\texttt{first cl\_p}_1(l_1)$, $\texttt{first state\_p}_1\texttt{(0)}$, and `first ch1({})`.

Simulation System Management

Suppose that we have obtained the initial structural representation of a given distributed computation and, at some moment in time, the knowledge base for the computation is $< \mathcal{V}, \mathcal{P}, ck, \mathcal{R}, \mathcal{I} >$ and there are no changes with the computation, then, at the next moment in time, the knowledge base is still the same. However, if there is a change at some stage of the computation, when, for instance, a new process is produced, then several new predicates may be needed and therefore some new local clocks as well as some new rules may need to be placed into the knowledge base. Thus, at the next moment in time, a new structural representation is generated. Since the knowledge base has a straightforward structure, it can easily be modified to obtain a new knowledge base from it.

To describe structural changes of the knowledge base, we define the following meta-predicates which will never appear in the specification of the simulation system:

`kb(S)`:	S is a knowledge base for the system.
`add_pr(X)`:	Process X is added to the system.
`assert_cod(X)`:	A initial condition X is asserted.
`ini_set(I)`:	I is the set of initial conditions.

At the management level, we have rules to generate the next structural representation. These rules can be represented as meta-Chronolog(MC) program clauses. For example, we assume that, when a new process p is currently produced, a new state variable X and control variable m private to the process are added, new predicates $Q1$ and $Q2$ are defined, a new local clock ck_i may therefore be defined with $ck(Q1) = ck(Q2) = ck_i$. We also assume that a set $\{r_1, \ldots, r_k\}$ of new rules may therefore need to be added into the set of rules. To do this, we may use the following rule:

```
next kb($\mathcal{V}'$,$\mathcal{P}'$,$ck'$,$\mathcal{R}'$,$\mathcal{I}$) <-
        kb($\mathcal{V}$,$\mathcal{P}$,$ck$,$\mathcal{R}$,$\mathcal{I}$), add_pr(p),
        $\mathcal{V}' = \mathcal{V} \cup \{X, m\}$, $\mathcal{P}' = \mathcal{P} \cup \{Q1, Q2\}$,
        $ck' = ck \cup \{ck(Q1) = ck_i$,
        $ck(Q2) = ck_i\}$, $\mathcal{R}' = \mathcal{R} \cup \{r1, \ldots, r_k\}$.
```

When updating a knowledge base, we also have to check whether the change is consistent with the existing knowledge. In order to ensure that the knowledge in the knowledge base of a simulation system is both accurate and consistent, we use knowledge representation techniques, including the constraints mechanism (Liu & Orgun 1997) and the feedback from the experts to perform consistency checking. At any moment in time, if the knowledge base for a given distributed computation is consistent and satisfactory, we can first translate it into a Chronolog(MC) program and then run it to obtain simulation results.

Discussion

There are also other knowledge-based modeling and simulation tools, such as Simulation Craft (Sathi *et al.* 1986), MASCOT(Mackulak & Cochran 1989), and SimKit (Silverman & Stelzner 1989). Most of these systems and tools are based on first-order logic which does not support time directly. Knowledge-based simulation methods based on first-order logic must therefore provide an explicit support for timing requirements. Our methodology is based on temporal logic which can naturally handle timing requirements. Compared with other simulation languages, Chronolog(MC) is more flexible in describing those systems where multiple granularity of time is essential.

Model development and application of expert knowledge are fundamental problems common to simulation methodology (Mackulak & Cochran 1989). In our methodology, the knowledge base of a simulation system is represented in a well-classified structural form and can be automatically transformed into a Chronolog(MC) program, which can be read declaratively and can be executed. Therefore, the user of the methodology can easily handle the above problems.

Acknowledgements

The work presented in this article has been supported in part by an Australian Research Council (ARC) Grant. C. Liu has been supported by an Australian Postgraduate Award (APA) and an MPCE Postgraduate Research Scholarship at Macquarie University.

References

Abadi, M., and Manna, Z. 1989. Temporal logic programming. *Journal of Symbolic Computation* 8:277–295.

Liu, C., and Orgun, M. A. 1995. Chronolog as a simulation language. In Fisher, M., ed., *Proceedings of IJCAI-95 Workshop on Executable Temporal Logics*, 109–119.

Liu, C., and Orgun, M. A. 1996a. Dealing with multiple granularity of time in temporal logic programming. *Journal of Symbolic Computation* 22:699–720.

Liu, C., and Orgun, M. A. 1996b. Executing specifications of distributed computations with Chronolog(MC). In *Proceedings of ACM Applied Computing 1996*, 393–400.

Liu, C., and Orgun, M. A. 1997. A constraint mechanism for knowledge specification of simulation systems based on temporal logic. In Sattar, A., ed., *Advanced Topics in Artificial Intelligence*, 485–495. Springer.

Mackulak, G. T., and Cochran, J. K. 1989. Mascot: A Prolog-based simulation modelling and training environment. In M.S. Elas, T. O., and Zeigler, B., eds., *Modelling and Simulation Methodology*, 145–159. North-Holland.

Moszkowski, B. 1986. *Executing Temporal Logic Programs.* Cambridge University Press.

Ören, T. I. 1987. Simulation: Taxonomy. In Singh, M. G., ed., *Systems and Control Encyclopedia*, 4411–4414. Oxford, England: Pergamon Press.

Sathi, N.; Fox, M.; Baskaran, V.; and Bouer, J. 1986. Simulation Craft: An artificial intelligence approach to the simulation life cycle. In *Proceedings of the SCS Summer Simulation Conference.*

Silverman, D., and Stelzner, M. 1989. Simkit^{TM}: Knowledge-based simulation tools. In M.S. Elas, T. O., and Zeigler, B., eds., *Modelling and Simulation Methodology*, 189–197. North-Holland.