Knowledge-Based Modeling Methodology for Simulation of Distributed Computations Using Chronolog(MC)

(Extended Abstract)

Chuchang Liu  Mehmet A. Orgun

Department of Computing
Macquarie University
NSW 2109, Australia

E-mail: {cliu, mehmel} @ krakatoa. mpce. mq. edu. au

1 Introduction

Since late 1970s, there has been a substantial interest in knowledge-based simulation methods. Knowledge-based simulation involves several model-related concepts such as: modeling formalisms, modeling environments, model-base management and processing of models [?]. Providing good simulation languages and implementation of environments to make integration of the above concepts are highly desirable.

Simulation requires models of a process to be simulated and simulation problems deal with processes evolving in time. The notion of time is implicitly built into temporal logic languages, which is essential in simulation. Therefore, temporal extensions of logic programming are particularly suitable for knowledge-based simulation applications. There are a number of temporal languages which are suitable for modeling simulation tasks, such as Tempura [?], Templog [?], Chronolog [?] and so on.

Chronolog is a typical temporal language [?], in which the set of natural numbers \( \omega \) models the collection of moments in time with an unbounded future. All predicates appearing in the language are defined on the “global clock”, i.e., the sequence of natural numbers \( < 0, 1, 2, \ldots > \). To model the behavior of systems in which granularity of time is needed, we proposed an extension of Chronolog by introducing multiple clocks, that is, defining local clocks associated with predicate symbols [?, ?]. The extended language is called Chronolog(MC), for “Chronolog with Multiple Clocks”. It is based on a Clocked Temporal Logic (CTL) in which all formulas can be clocked and are allowed to be defined on local clocks which are subsequences of the global clock.

In a distributed computation, processes involved in the computation have their own “local clocks”. When modeling such a computation, we may need to consider multiple clocks. Because of the introduction of multiple clocks, Chronolog(MC) can be applied to the specification of a wide range of simulation systems and other relevant tasks. In particular, it can be used to model the functional and timing behavior of distributed computations based on Hoare’s CSP formalism. In [?], we have discussed the application of Chronolog(MC) in the simulation of distributed computations. In this paper, we furthermore propose a knowledge-based modeling methodology for the specifications of distributed computations with clocked temporal logic programming. The methodology is based on the idea that we may combine the trajectory and structural simulation [?] together to describe a distributed computation. It is intended to be used for building a simulation environment for distributed computations, which provides an integration of modeling formalism, modeling environment, knowledge-base management and processing of the model in simulation.

The structure of this paper is as follows. Section 2 introduces the clocked temporal logic Chronolog(MC) is based on, and a clocked temporal resolution method that can be used as the basis for executing programs. In section 3, we discuss modeling formalisms for describing distributed computations. In section 4, we discuss the use of Chronolog(MC) in the simulation of distributed computations based on
the formalisms, and propose two phase specifications for distributed computations. Section 5 discusses executing specifications and modification of the structural representation of simulation of a distributed computation. The last section concludes the paper with a brief summary.

2 Clocked Temporal Logic, CTL

2.1 Logical basis

Clocks We define that the global clock is the strictly increasing sequence of natural numbers: \(<0, 1, 2, \ldots, >\), and a local clock is a subsequence of the global clock. Let \(CK\) be the set of all clocks and \(\subseteq\) be a ordering relation on the elements of \(CK\) defined as follows: for any \(ck_1, ck_2 \in CK\), \(ck_1 \subseteq ck_2\) if for all \(t \in ck_1\), we have that \(t \in ck_2\). Here the expression \(t \in ck_i\) denotes the fact that \(t\) is a moment in time on the clock \(ck_i\). It is easy to show that \((CK, \subseteq)\) is a complete lattice. Therefore, we can define

\[
ck_1 \cap ck_2 \overset{\text{def}}{=} \text{g.l.b.}\{ck_1, ck_2\} \quad ck_1 \cup ck_2 \overset{\text{def}}{=} \text{l.u.b.}\{ck_1, ck_2\}
\]

where \(ck_1, ck_2 \in CK\), g.l.b. stands for “greatest lower bound” and l.u.b. for “least upper bound” under the relation \(\subseteq\).

Syntax Any formula of first-order logic is a formula of CTL. In addition to the formation rules of formulas of first-order logic, we have two rules for temporal operators: if \(A\) is a formula, then \(\text{first } A\) and \(\text{next } A\) are also formulas.

In CTL, 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 all clocks. We denote the clock which is associated with a predicate symbol \(p\) by \(ck(p)\). For any formula \(A\), the clock \(ck(A)\) is determined by the clocks of predicate symbols appearing in \(A\). In other words, let \(\text{Pred}\) be the set of predicate symbols which appear in \(A\). Then we define that \(ck(A) = \bigcap_{p \in \text{Pred}} ck(p)\).

In the following, we use the notation \(ck_0, ck_1, \ldots, ck(A), ck(B), \ldots\) to represent local clocks, in particular, we denote the global clock as \(gck\).

Given a local clock \(ck_i =＜t_0, t_1, t_2, \ldots＞\). We define the rank of \(t_n\) on \(ck_i\) to be \(n\), written as \(\text{rank}(ck_i, t_n) = n\). 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 the global clock, we have \(\text{rank}(gck, t) = t\) and \(gck^{(t)} = t\).

Semantics The semantics of formulas with logical connectives are defined in the usual way, but with respect to local clocks [?]. Here we only give the informal meaning of temporal operators of CTL:

- For any \(t \in ck(A)\), \(\text{first } A\) is true at \(t\) iff \(A\) is true at \(ck(A)^{(t)}\).
- For any \(t \in ck(A)\), \(\text{next } A\) is true at \(t\) iff \(A\) is true at \(ck(A)^{(t)}\), where \(i = \text{rank}(ck(A), t)\).

Clocked atoms We say that an atom is pure if it does not contain any temporal operators. An atom is fixed-time if it has an application of \(\text{first}\) followed by a number of applications of \(\text{next}\). Any fixed-time atom is fixed to some moment in time on the local clock associated with the formula which the atom is currently involved in. Since different formulas may have different local clocks, the same predicate appearing in different formulas may have different fixed forms even when it is fixed to the same moment in time in the different formulas. Therefore, we need the following definition and lemma which form the basis of clocked temporal resolution (see below).

Definition 1[?] Let \(Q\) be a pure atom, \(ck_i\) a local clock, \(t \in ck_i\), \(ck_i \subseteq ck(Q)\), and \(n = \text{rank}(ck_i, t)\). Then \(\text{first } \text{next}(n)Q\) is a fixed-time atom on \(ck_i\), fixed to the moment \(t\) in time. Furthermore we call \(\text{first } \text{next}(n)Q|_{ck_i}\) a clocked fixed-time atom on \(ck_i\) and \(t\) the current time of the clocked fixed-time atom.

Lemma 1[?] Let \(Q\) be a pure atom appearing in a formula \(A\) and \(t \in ck(A)\). Let \(n = \text{rank}(ck(A), t)\) and \(m = \text{rank}(ck(Q), t)\). Then
first $\text{next}(n)Q_{\text{ck}}^{\text{cl}}(A)$ is true iff
first $\text{next}(m)Q_{\text{ck}}^{\text{el}}(Q)$ is true.

2.2 Chronolog(MC) Programming

A Chronolog(MC) program $P$ consists of three components:

$$P = P_c \ 1 \ P_a \ 1 \ P_b$$

where $P_c$, $P_a$ and $P_b$ are the clock definition, the clock assignment and the program body of the program, respectively, and $1$ means "jointing". That is, $P_c$, $P_a$ and $P_b$ jointly form the program $P$. The clock assignment $\text{ck}$ of $P$ is totally determined by $P_c$ and $P_a$.

We impose a syntactic restriction on clock definitions to ensure that each clock definition specifies a local clock. Program clauses allowed to appear in clock definitions for any given local clock $\text{cki}$ can only be of the following form:

$$\text{first} \ cki(n).$$
$$\text{next} \ cki(N) \leftarrow \ cki(M), \ N \text{ is } E(M), \ N \geq M.$$  

where $n \in \omega$ is the initial value of the clock, and $E(X)$ is a single-valued function from $\omega$ to $\omega$. The second clause specifies the "next" value of the clock using its current value.

$P_a$ assigns local clocks for all the predicate symbols in the program body. It consists of several facts of the form

$$\text{is-ck}(p, cki).$$

The clause says that $cki$ is the local clock associated with the predicate symbol $p$, where $cki$ is defined in $P_c$ and $p$ appears in $P_b$.

The program body $P_b$ consists of rules and facts. Its meaning depends on a given clock, i.e., the clock definition $P_c$ and clock assignment $P_a$. Two programs with different clock definitions and/or different clock assignments are totally different even when they have the same program body.

By lemma 1, if two clocked fixed-time atoms have the same pure atom and the same current time, then they have the same truth value. Therefore, we have the following definition:

**Definition 2** [?]: Let $A^t_{\text{cl}}$ and $B^t_{\text{cl}}$ be two clocked fixed-time temporal atoms. If $t = s$ and the pure atoms contained in $A$ and $B$ can be unified, then we say that $A^t_{\text{cl}}$ and $B^t_{\text{cl}}$ are unified, and the substitution $\theta$, which unifies the pure atoms, is a substitution unifying $A^t_{\text{cl}}$ and $B^t_{\text{cl}}$.

The definition extends the concepts of temporal-matching and temporal unification in the original Chronolog.

Clocked temporal matching includes two aspects: (1) matching the pure atoms, (2) matching the current time. The matching of pure atoms is straightforward; while "time-matching" is involved in the execution of clock definitions and clock assignments [?].

3 Modeling Formalisms for Distributed Computations

3.1 Distributed 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 process.
The specification for distributed computations will be based on two models: the synchronous communication model and the asynchronous communication model. In a distributed computation, the variables of each process are private, i.e., not accessible to other processes, and message passing is used to provide interaction between processes. In the synchronous communication model, we use explicit send-receive primitives. The messages are not buffered, and communication is performed by a simultaneous execution of a send-receive pair. Asynchronous communication is message exchange without acknowledgment. The message exchange is not synchronised, so communication requires buffering for the messages that have been sent but not received.

The form of distributed programs is as follows:

parallel: \( P : [Y_1; P_1 || \ldots || Y_m; P_m] \)
sequential: \( P_i : \{d_1^i; \ldots ; d_k^i\}, i = 1, \ldots , m \)

where \( P_1, \ldots , P_m \) are parallel processes of the program \( P \), which can be executed concurrently and the individual processes are sequential programs; \( Y_1 \cup \ldots \cup Y_m = Y \) is the set of data variables, and \( Y_1, \ldots , Y_m \) are disjoint subsets of data variables private to the processes \( P_1, \ldots , \) and \( P_m \), respectively; \( d_1^i, \ldots , d_k^i \) are simple instructions.

The types of instructions include:

skip: \( \text{skip} \)
assignment: \( X := e \)
receive: \( c; \alpha \uparrow X \)
send: \( c; \alpha \downarrow e \)

The first two types of instructions place no restrictions on execution. The instructions send and receive serve to pass messages between processes through channels. Here \( c \) is a boolean expression, \( e \) is an expression and \( X \) is a variable, and \( \alpha \) is a channel name. 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 then it attempts to receive a value from the channel \( \alpha \) for the variable \( X \). The send instruction can be performed only when \( c \) is true and then it attempts to send the value of \( e \) along the channel \( \alpha \).

The processes communicate by sending and receiving messages along channels. Each channel connects two processes: one is the sending process, the other the receiving process. We use the notation \( ch_{B|TR} \) to denote the channel which connects the sending process \( P_i \) and the receiving process \( P_j \).

As the first step for describing distributed computations with Chronolog(MC), in this paper, we consider the model, which can be viewed as the weakest possible model for a distributed system called an asynchronous system and characterized by the following 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.

### 3.2 Trajectory and structural simulation

In any type of simulation [?], a dynamic model is a model with time-indexable behavior, which is driven by investigating behavior change. Trajectory behavior of a model is the most common type of model and consists of a time-indexed sequence of values for some variables of the model. Structural behavior of a model 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. Particularly, in trajectory simulation, we 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 its next structural representation.

Development of modeling formalisms to represent initial representation of the structure of a system as well rules to use in determining the next representation of the structure may facilitate structural simulation. The structural change may also use rule-based decision making.
3.3 Structural representation of distributed computations

As we use formalisms where knowledge of the domain is explicit, we call these models knowledge-based. A requirement of the knowledge-based models for simulation purposes is to provide the possibility of reasoning on the conceptual schema (i.e. model) of the application. In the following, we present knowledge-based models, based both trajectory and structural simulation, to simulate distributed computations with Chronolog(MC). Through building models we show how to provide a structural representation for simulation of a distributed computation and how to manage it. We also show how a knowledge base, modeling a given distributed computation, can be directly transformed into an executable specification.

A distributed computation can be described by the execution of a distributed program which is performed by a set of processes, therefore, in view of structural simulation, it consists of a collection of processes and the local clocks associated with these processes.

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

That is, to model distributed computations, we first want to construct domain-specific knowledge bases. In general, we have the following steps:

- defining predicates which model the behavior of the system,
- defining local clocks for predicates,
- collecting the rules which describe the dynamics of the system,
- specifying the initial conditions of the system which are represented as facts that are true at the initial time when the simulation begins.

Thus, a knowledge base for specifying a distributed computation consists of four components: a clock definition, denoted by $C_d$, a clock assignment, denoted by $C_a$ which assigns a local clock for each predicate symbol, a set of rules, denoted by $R$, and a set of initial conditions, denoted by $I$. We denote the knowledge base as $<C_d, C_a, R, I>$, which is a structural representation of the distributed computation we are specifying.

Usually, the change of structural representations occur at the management level. When a new process is currently produced in the computation, several new predicates may be needed and therefore some new local clocks as well some new rules may need to be put into the knowledge base. Thus, at the next moment in time, a new structural representation is generated. More discussion about the change of structural representations is given in the section 5.

4 Specifications for Distributed Computations

In order to illustrate the use of the formalisms proposed in the previous section, we elaborate the idea through applying them to specifications of distributed computations.

4.1 First phase: Specification in Hoare’s CSP

We use Chronolog(MC) to model distributed computations based on Hoare’s CSP formalism [7]. We now consider the first phase: specification in Hoare’s CSP.

Given a distributed computation as shown in section 3, for each process $P_i$ ($i = 1, 2, \ldots, m$), we may need to define several "local" predicates (local to the process) and several predicates to describe the status of the channels related to the process. These predicates are:
• the process state predicate: \( \text{state}_p(U_1, \ldots, U_k) \)
• the process control predicate: \( \text{cl}_p(S) \)
• the channel value predicate: \( \text{chP}_R(L) \)
• the assignment predicate: \( \text{assign}_p(U, E) \)
• the channel value change predicates: \( \text{in}_p \text{change}_R(L, E), \text{out}_p \text{change}_R(L, E) \)

States of process \( P \) are assignments of values from the appropriate domains to variables \( U_1, \ldots, U_k \). The control predicate \( \text{cl}_p(L) \) means that the control of the statement execution in process \( P \) has been moved to the instruction whose label is \( S \), in other words, the process is executing the instruction \( S \).

A predicate related to communications may be involved in more than one process, and these processes may have different "local times". Therefore, such predicates should be investigated with the global view, and, therefore, they may be defined on a global clock. The others may be defined on local clocks based on the environment provided. For the above predicates, \( \text{state}_p, \text{cl}_p \) and \( \text{assign}_p \) may be defined on the local clock of process \( P \). 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 clock definition and assignment depend on the computational environment of a distributed computation. The rules describing the behavior of the computation can be automatically produced based on 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.

The first phase of the specification for the distributed computation is its initial structural representation based on Hoare's CSP formalism, denoted by \( < C_0^I, C_0^O, R_0, \Phi > \). The clock definition \( C_0^I \) and clock assignment \( C_0^O \) are directly given based on the computational environment, for example, we may have:

\[
C_0^I = \{ c k_1 = \langle 0, 3, 6, 9, \ldots >, \ c k_2 = \langle 1, 2, 4, 5, \ldots >, \ c k_3 = \langle 1, 3, 5, 7, \ldots >, \ \ldots \}.
\]
\[
C_0^O = \{ \ c k(\text{state}_P_1) = c k_1, \ c k(\text{cl}_P_1) = c k_1, \ c k(\text{state}_P_2) = c k_2, \ c k(\text{cl}_P_2) = c k_2, \ \ldots \}
\]

The rules come from the instructions of the system that we want to specify. There are two kinds of instructions: one has only a simple action, the other has an action with a condition. Let \( L : \text{action} \) be an instruction of a process \( p \), then we have the rule:

• If the process \( p \) is currently executing \( L \), then it will execute the next instruction at the next moment in time.

More formally, we have:

\[
\text{executes}(p, \text{action}) \rightarrow \text{next executes}(p, L').
\]

where \( L' \) is the next instruction. Similarly, let \( L : c; \text{action} \) be an instruction with a condition, then have the rule:

\[
\text{executes}(p, \text{action}) \& c \text{ is true } \rightarrow \text{next executes}(p, S).
\]
\[
\text{executes}(p, \text{action}) \& \neg (c \text{ is true}) \rightarrow \text{next executes}(p, S).
\]

All rules which describe the dynamic of a computation, including the changes of process states and channel states involved in the computation, forms the set \( R^0 \) under the structural representation.

The set \( R^0 \) of initial conditions includes the facts that are true at the initial moment on their clocks. 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
• At the begin, the control of the process \( p \) at the instruction \( L \).

Formally, we write

\[
\text{control}(p, L).
\]

Details about the structural representation will be presented in the full paper.

### 4.2 Second phase: Executable specifications

The specification we make in the first phase can not be directly executed. However, the specification in Hoare’s CSP for a distributed computation can be automatically transformed into an executable specification, which is represented as a Chronolog(MC) program. The translation procedure is totally mechanical.

Let a knowledge base \( \langle C^0_0, C^0_1, R^0_0, \theta^0 \rangle \) be the first phase specification for a distributed computation. Then we have the following translation form:

\[
\begin{align*}
C^0_0 & \to P_C \\
C^0_1 & \to P_a \\
R^0_0 \cup \theta^0 & \to P_b
\end{align*}
\]

where \( P_C, P_a, P_b \) will form a real Chronolog(MC) program. This is, any structural representation, or a knowledge base for a distributed computation can be transformed into an executable specification written in Chronolog(MC). Therefore, it can be directly executed.

We can use the code producer to produce Chronolog(MC) programs in a specific method. In the method, any element in \( C^0_0, C^0_1 \) and \( R^0_0 \cup \theta^0 \) can be automatically transformed into one or more Chronolog(MC) program clauses. For example, the clock \( \text{ck}.1 = \langle 0, 3, 6, 9, \ldots \rangle \) is transformed into

\[
\begin{align*}
\text{first } & \text{ck1}(0). \\
\text{next } & \text{ck1}(N) \leftarrow \text{ck1}(N), \text{N is } M+1.
\end{align*}
\]

In this paper, we are assuming that all clocks admit such a transformation. The clock assignment \( \text{ck}(\text{cl.p})=\text{ck1} \) is transformed into

\[
\text{is.ck}(p, \text{ck1}).
\]

Any rule related to a process can be also easily transformed into program clauses. For example, assume that we have a process called \( P_1 \) as follows:

\[
P_1 = \{ h_1 : I = I + 1; \\
\text{h2 : I > 0; ch1 \downarrow I; } \\
\text{h3 : skip h1} \}.
\]

Then we have the following rules related to the process \( P_1 \).

\[
\begin{align*}
\text{next.state.(P1(U1))} & \leftarrow \\
& \text{cl.P1(h1)}, \text{state.(P1(U1))}, \text{U1 is } U+1. \\
\text{next cl.P1(h2)} & \leftarrow \text{cl.P1(h1)}. \\
\text{in.change.ch1(L,U)} & \leftarrow \\
& \text{cl.P1(h2)}, \text{U > 0, ch1(L)}. \\
\text{next ch1(L1)} & \leftarrow \\
& \text{in.change.ch1(L,U)}, \text{L1 = L\ast U}. \\
\text{next.state.(P1(U))} & \leftarrow \\
& \text{cl.P1(h2)}, \text{state.(P1(U))}. \\
\text{next cl.P1(h3)} & \leftarrow \text{cl.P1(h2)}, \text{U > 0}.
\end{align*}
\]
next cl\_P\_1(h_2) <-
c1\_P\_1(h_2), \neq (\exists u > 0).
next state\_p\_1(u) <-
c1\_P\_1(h_3), state\_p\_1(u).
next c1\_P\_1(h_1) <- c1\_P\_1(h_3).

The translation of the initial conditions is similar.

5 Management of Knowledge Bases

Let \( < C_d^0, C_a^0, R^0, r^0 > \) be the initial structural representation for a given distributed computation. If at some moment in time the structural representation, or knowledge base for the computation is \( < C_d, C_a, R, I > \) and there is no change with the computation, then at the next moment in time the knowledge base is still \( < C_d, C_a, R, I > \). However, there may be some changes for the computation. The changes mainly come from the following reasons:

- new processes are produced, and
- new initial conditions are required.

The changes will bring about the change of the knowledge base for the computation. That is, at this case, the next structural representation will be different from the current one.

Formally, we define another predicate:

\( kb(S): S \) is the structural representation for the distributed computation

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, when a new process is currently produced, a new predicate \( Q \) is defined, and a new local clock \( ck_i \) may therefore be defined and a set \( \{ r_1, \ldots, r_k \} \) of new rules may need to be added into the set of rules. To do this, we may use the following rule:

\[
\text{next } kb(C_d^0, C_a^0, R^0, I) \leftarrow kb(C_d, C_a, R, I), \text{ process } p \text{ is added,}\\
C_d^0 \text{ is } C_d \cup \{ck_i\}, C_a^0 \text{ is } C_a \cup \{ck(Q) = ck_i\},\\
R^0 \text{ is } R \cup \{r_1, \ldots, r_k\}.
\]

The full description of the rules to generate the next structural representation will be presented in the full paper.

6 Conclusions

In Chronolog(MC), the presentation of granularity of time is explicitly provided by a clock definition and assignment. Therefore, the language is flexible and it is suitable for modeling and specifying a range of simulation systems. Granularity of time is also considered in applications such as ecological modeling [?], and temporal databases [?].

We have presented a knowledge-based modeling methodology for simulation of distributed computations using Chronolog(MC). The methodology can be used to build a simulation environment for distributed computations. Such an environment provides an integration of modeling formalism, modeling knowledge-base management and processing of the model itself.

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).
References


