A parallel execution model for Chronolog

Chuchang Liu*1, Mehmet A Orgun† and Kang Zhang†

*Information Technology Division, Defence Science and Technology Organisation, PO Box 1500, Salisbury, SA 5108, Australia
Email: Chuchang.Liu@dsto.defence.gov.au
†Department of Computing, Macquarie University, Sydney, NSW 2109, Australia. Email: {mehmet, kang}@ics.mq.edu.au

Chronolog(2) is a logic programming language based on a linear-time temporal logic with unbounded past and future. By adding ‘choice predicates’ to Chronolog(2), it is possible to obtain exactly one answer to a given goal when we want to model dataflow style of stream-oriented computations. In this paper, we propose a parallel execution model for Chronolog(2) that supports AND- and OR-parallelism in the highly distributed dataflow environment. In order to exploit the inherent context-parallelism that exist in temporal logic programs, we introduce the concept of parallel context-processes into the model. We also introduce an intermediate virtual machine (CVM), which is granulated to exploit the argument parallelism through temporal unification. The details of the CVM instruction set are given. The use of a warehouse facility as an associate memory to store the results of previous computations is an important feature of this model. This paper discusses the structure of the warehouse, its role and use in parallel execution of Chronolog programs. We in particular outline an algorithm to manage the warehouse.

Keywords: Temporal logic programming, operational semantics, context-parallelism, warehouse, dataflow computation

1. INTRODUCTION

Temporal logic has been widely used as a formalism for program specification and verification [9, 12, 13], modelling temporal databases [1, 3, 14] and reasoning about time [8, 19, 20]. In temporal logic, the meanings of formulae depend on an implicit time parameter and elements from different moments in time can be combined through the use of temporal operators. Therefore, temporal logic can model time-dependent and dynamic properties of certain problems in the real-world in a natural way. Recently, several researchers have suggested that temporal logic can be used as a programming language; see [15] for a comprehensive survey. However, there have been a few attempts at implementing temporal logic languages. Some of the early suggested implementations, which are based on translation into standard logic programs [21] or by meta-interpretation [7], cannot exploit parallelism inherent in logic programs and context-parallelism which is offered by temporal logic programs.

Chronolog [16, 17] is a temporal extension of logic programming, based on a linear-time temporal logic [4] with an unbounded future in which the set of natural numbers models the collection of moments in time. For the temporal logic programming language Chronolog, we have outlined a parallel execution model, called CHEM (CHronolog Execution Model) [10]. The model is based on dataflow computation. Chronolog(2) is an extension of Chronolog based on a linear-time temporal logic with unbounded past and future. By adding ‘choice predicates’ to Chronolog(2), we can obtain exactly one answer to a given goal when we want to model dataflow style of stream-oriented computations. In order to exploit inherent parallelism in Chronolog(2), this paper,
based on the framework proposed in [10, 11], discusses and enhances the parallel execution model CHEM so that it is suitable for Chronolog(Z) programs with choice predicates.

In the rest of this section, we introduce some general concepts that are needed for the later discussion. We discuss the underlying temporal logic of Chronolog, its execution model, and choice predicates non-determinism.

1.1 Temporal logic operators

The temporal logic of Chronolog has two temporal operators, first and next, which refer to the initial and the next moment in time respectively. The initial moment in time is 0. Apart from two temporal operators first and next of Chronolog, the temporal logic of Chronolog(Z) introduces the third temporal operator prev which, informally, refers to the previous moment in time. Thus, in Chronolog(Z), the collection of moments in time is the set of integers \( Z \) with its usual ordering relation <.

The underlying logic of Chronolog(Z) refers to a natural extension of a standard first-order logic language. Let \( L \) be a standard first-order language and TL the temporal language corresponding to L. TL has all the formation rules of L plus the following new formation rule:

- If \( F \) is a formula of TL, so are first \( F \), prev \( F \) and next \( F \).

In TL, Temporal atomic formulae are defined inductively as follows:

- If \( p \) is an n-ary predicate symbol and \( e_1, ..., e_n \) are terms, then \( p(e_1, ..., e_n) \) is a temporal atomic formula. Such formulae are in particular called pure temporal atomic formulae or pure atoms.

- If \( A \) is a temporal atomic formula, so are first \( A \), next \( A \) and prev \( A \).

An atomic formula of TL is simply called a tl-atom (i.e. temporal atom). There are two sorts of tl-atoms defined as follows:

- 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.

Note that the operators first, next and prev are only applied to formulae, not to terms in this language. In the following, we write next \( (n) \) for \( n \) successive applications of next. When \( n = 0 \), next \( (0) \) is treated as the empty string. Similarly for prev.

1.2 Warehousing in the Chronolog Execution Model

Chronolog execution model CHEM is a data-driven execution model which has a flexible capability to support parallelism at various levels. For efficiency, it is desirable to be able to reuse the results of the previous computations and to avoid some unnecessary computations in the execution of Chronolog programs. Context-parallelism is implemented cost-effectively by using a warehouse (a blackboard mechanism) 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. The use of a warehouse as an associate memory to store the results in the previous computations is an important feature of CHEM. The strategy to employ a warehouse in the parallel executions of Chronolog programs can effectively avoid repeated and unnecessary evaluations of a temporal atom which needs to be reduced several times in different contexts.

As an example, let us consider the following Chronolog-program that specifies the predicate fib that is used to generate the sequence \( f = (0, 1, 1, 2, 3, 5, ...) \) of Fibonacci numbers, where \( f_0 = 0, f_1 = 1 \) and \( f_{i+2} = f_{i+1} + f_i \) for all \( i \geq 0 \):

\[
\begin{align*}
\text{first fib}(0). \\
\text{first next fib}(1). \\
\text{next next fib}(Z) & \leftarrow \text{next fib}(X), \text{fib}(Y), Z \equiv X+Y.
\end{align*}
\]

In the program, the first two program clauses define the first two Fibonacci numbers as 0 and 1; the last clause defines the current Fibonacci number as the sum of the previous two. Assume that we have the goal

\[
\leftarrow \text{first next next 5 fib}(0).
\]

Compare the following two cases:

- In the case with warehousing, if both of the atoms first next(3) fib(2) and first next(4) fib(3) are computed earlier and stored in the warehouse, then we only need to call the third program clause once to obtain the answer to the above goal.

- In the case without warehousing, the results of previous computations are not stored and cannot be reused. Therefore, to obtain the answer to the goal, we need to call third program clause at least four times, the second one three times, and the first one twice.

It is clear that without warehousing the complexity of the computation of each Fibonacci number is exponential, whereas with warehousing, significant speedups can be achieved. However, a warehouse facility which never ‘forgets’ is not necessarily a good one, as it will result in a large, ever-growing associative memory and increased warehouse searching. Therefore, a fine balance between the storage and removal of computed temporal atoms is required.

1.3 Choice predicates

Logic programming is non-deterministic in that there may be more than one possible solution to a given goal. However, in many cases, we are interested in only one solution to a given goal, but we do not want to make the choice unnecessarily specific in a given program. Perhaps the problem specification itself does not constrain the programmer to produce one particular stream. Therefore, it would be against the spirit of logic programming to force the programmer to make a
specific choice. Based on this idea, Orgun and Wadge proposed an extension to temporal logic programming called ‘choice predicates’ [16, 18].

Choice predicates are not defined by users in temporal logic programs, but they are supplied by the implementation for each predicate defined in a given program. They are associated with each predicate that appears in a given Chronolog program, and represent arbitrary but definite single-valued relations.

Suppose we want to produce an ‘arbitrary’ stream of natural numbers starting from 0 in strictly increasing order. We may consider the following Chronolog program.

\[
\text{first stream}(0). \\
\text{next stream}(Y) \leftarrow \text{stream}(X), \ X < Y.
\]

What it really says that if \( X \) is the (say) 10th output, and \( X \) is less than \( Y \), then \( Y \) might be the 11th output. The stream predicate is not single valued. In fact, at time \( t > 0 \), it is true of all \( x \geq t \). It can be thought of as a ‘fuzzy’ stream which at each point in time has many possible values. The rule is that every value possible at time \( t + 1 \) is greater than some value possible at time \( t \). But there is no way in which we can regard the \text{stream} predicate as representing an increasing stream or even a family of increasing streams. There are of course many such streams.

The solution to the above problem is thus to provide an extra predicate \#\text{stream} which succeeds only for the ground term that was actually produced. Then the temporal logic program should be as follows:

\[
\text{first stream}(0). \\
\text{next stream}(Y) \leftarrow \#\text{stream}(X), \ X < Y.
\]

Here the \#\text{stream} predicate is called a ‘choice predicate’. Note that the symbol \# is not an operator, it is part of the choice predicate. Then the \#\text{stream} predicate represents a stream of output values chosen non-deterministically from those terms which the \text{stream} predicate represents over the collection of moments in time. Note that \#\text{stream} represents a single-valued relation at any moment in time.

Choice predicates are allowed in the bodies of program clauses and goals, but not in the heads. For a correct implementation of choice predicates, we need only keep a permanent record of choices made, as they are made. The use of the warehouse is therefore essential in an implementation of Chronolog(\( Z \)) with choice predicates.

1.4 Outline of the paper

In this paper, a detailed description of the CHEM model is presented. First, we propose the concept of parallel context-processes. A parallel context-process solves a goal by creating a child-computation to solve each context-sub-goal. Secondly, at the level of clause arguments, we propose an intermediate virtual machine, called Chronolog Virtual Machine (CVM), that is able to exploit the argument parallelism through temporal unification, and in particular, we give the details of the CVM instruction set which are important for implementing CHEM. In CHEM, a source Chronolog program is formally compiled into a CVM object program that is a set of dataflow graphs. Each graph is responsible for a clause execution, and all nodes whose operands are available can be executed in parallel.

We also discuss the structure of the warehouse, its use and role in the parallel execution of Chronolog programs. The warehouse maintains computed temporal atoms and provides temporal-matching to a given goal. To use the warehouse more effectively, realistic warehouse management strategies are necessary. We discuss the management of the warehouse, and propose an algorithm to manage the warehouse. We also show, through an example, that introducing warehouse to CHEM can deal with the context-parallel computations more efficiently.

2. OVERVIEW OF CHRONOLOG(\( Z \))

2.1 Temporal logic programs

The temporal logic language Chronolog(\( Z \)) is a subset of \( TL \).

It has the following simple syntax. Let \( A \) denote an atom of \( TL \). We have the rules:

\[
\begin{align*}
\text{Body:} & \quad B ::= \square | B_1, B_2 \\
\text{Clause:} & \quad C ::= A \leftarrow B \\
\text{Goal:} & \quad G ::= B
\end{align*}
\]

where \( \square \) denotes the empty body. Throughout this paper, we use letter \( A \) to denote a tl-atom, \( B \) for a body, \( C \) for a program clause, and \( G \) for a goal clause. The variables in program and goal clauses are implicitly universally quantified. In a clause, the head is the consequent of the implication. In a body, a comma stands for the conjunction operator, i.e. \( \land \). A program clause that has an empty body is a fact. An empty body corresponds to ‘true’. A goal clause can be treated as a clause with an empty head, and the empty head corresponds to ‘false’.

By the axioms of \( TL \) (see Section 2.2), all superfluous applications of temporal operators in a formula can be eliminated (for instance, we can eliminate \text{next} in the formula \text{next first} \( A \)). We can therefore simplify a tl-atom so that it is either an open-end tl-atom that may not contain any temporal operators or contains only a number of applications of next or a number of applications of prev, or a fixed-time tl-atom that has an application of first followed by a number of applications of next or a number of applications of prev. That is, any tl-atom can be simplified into one of the following forms: \( A, \text{next} (n) A, \text{prev} (n) A, \text{first} A, \text{first} (n) A, \text{and first} \text{prev} (n) A \), where we assume that \( A \) is a pure atom and \( n \) is a positive integer. We do not consider applications of temporal operators to whole program clauses, since all temporal operators can be pushed inside by the use of the axioms of \( TL \) until we reach atomic formulae. The details is omitted.

In CHEM, a source Chronolog(\( Z \)) program is a finite set of program clauses, that is, a conjunction of temporal logic program clauses. Also, for goals, we have the following formal definition:

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

The difference between standard logic programs and
Chronolog(\(Z\)) programs is, in fact, only that program clauses of Chronolog(\(Z\)) programs contain applications of temporal operators to atomic formulae. In other words, Chronolog(\(Z\)) program clauses are formulae built up from the standard logic program connectors, such that by dropping the temporal operators we may obtain a classical Horn clause.

Program clauses in Chronolog(\(Z\)) programs are interpreted as assertions true at all moments in time. As an example, let us consider the following Chronolog(\(Z\)) program which specifies the simulation of a traffic light modeled by the time-varying light predicate.

\[
\begin{align*}
\text{first light (green).} \\
\text{prev light (red) } &\leftarrow \text{light (green).} \\
\text{prev light (green) } &\leftarrow \text{light (amber).} \\
\text{prev light (amber) } &\leftarrow \text{light (red).} \\
\text{next light (amber) } &\leftarrow \text{light (green).} \\
\text{next light (red) } &\leftarrow \text{light (amber).} \\
\text{next light (green) } &\leftarrow \text{light (red).}
\end{align*}
\]

The program says that at time 0, the traffic is green and, at any given moment in time, if the light is green, it must rotate from red at the previous moment in time and rotate to amber at the next moment in time; if the light is red, it must rotate from amber at the previous moment in time and rotate to green at the next moment in time; if the light is amber, it must rotate from green at the previous moment in time and rotate to red at the next moment in time.

Two examples of fixed-time goals are given as follows:

\[
\begin{align*}
\text{first prev light (Color).} \\
\text{first next light (Color).}
\end{align*}
\]

The first one is a query about the history of the traffic light, while the second one about the future of the light. The goal

\[
\text{prev light (Color).}
\]

is an example of an open-end goal, which is not fixed to any particular moment in time. Such a goal stands for an infinite series of independent fixed-time goals, and the answers to the goal are those answers obtained from all the independent fixed-time goals. In fact, the above open-end goal stands for an infinite series of independent fixed-time goals of the form

\[
\begin{align*}
\text{first prev } n \text{ light (L) or first next } n \text{ light (L) for all } n \geq 0.
\end{align*}
\]

Therefore, the answers to the goal are those answers obtained from all the independent fixed-time goals listed above.

### 2.2  Axioms and rules of inference

\(TL\) has a number of axioms which formalize the interaction between logical and temporal operators. Read \(\leftrightarrow\) as ‘if and only if’. Let \(\forall\) stand for any of \(\text{first, next, and prev}\). Then we have the following axioms:

\[
\begin{align*}
\text{A1. } &\forall (\text{first } F) \leftrightarrow \text{first } F. \\
\text{A2. } &\text{next prev } F \leftrightarrow F. \\
\text{A3. } &\text{prev next } F \leftrightarrow F. \\
\text{A4. } &\forall (F_1 \land F_2) \leftrightarrow (\forall F_1) \land (\forall F_2). \\
\text{A5. } &\forall (\neg F) \leftrightarrow \neg (\forall F).
\end{align*}
\]

Here \(A1–A3\) are called temporal operator cancellation rules, \(A4\) and \(A5\) are temporal operator distribution rules, and \(A6\) the rule with rigidity of variables, which stipulates that the values of individual variables range over extensions (data values), not intensions (time varying value). It is an instance of the so-called Barcan formula combined with its converse [6].

Let the notation \(|- F|\) denote the fact that \(F\) is a theorem of \(TL\). In addition to substitution and Modus Ponens, we have the following temporal operator introduction rules which are used for temporal resolution.

\[
\begin{align*}
\text{R1. } &\text{If } |- F, \text{ then } |- \text{first } F. \\
\text{R2. } &\text{If } |- F, \text{ then } |- \text{next } F$. \\
\text{R3. } &\text{If } |- F, \text{ then } |- \text{prev } F
\end{align*}
\]

We read the rules of inference as ‘given \(F\) as a theorem, infer \(\text{first } F, \text{ prev } F\) and \(\text{next } F\) as theorems’. Also, we have a rule called the induction rule as follows:

\[
\begin{align*}
\text{R4. } &\text{If } |- \text{first } F, \text{ then } |- \text{next } F, \\
&\text{ and } | - F \rightarrow \text{prev } F, \text{ then } |- F.
\end{align*}
\]

Induction rule is a form of temporal operator elimination rule.

In this paper, we do not give the details about the semantics of \(TL\) and the soundness of the axioms and rules of inference. We refer the reader to [16, 17].

### 2.3  A revised TiSLD-resolution

Let \(P\) be a Chronolog(\(Z\)) program and \(G\) a goal \(\leftarrow B\). A correct answer of a computation, which is obtained by deriving a contradiction using TiSLD-resolution [16, 17] is an answer substitution \(\theta\) such that \(\forall (B\theta)\) is a logical consequence of \(P\). TiSLD-resolution is, in fact, a refutation procedure of Chronolog(\(Z\)), which is applied to a set of \{canonical\} (fixed-time) program clauses and goal clauses.

In Chronolog, the canonical instances of a formula \(F\) are obtained by applying the temporal operator \(\text{first}\) followed by a number of \(\text{next}\) ‘s and also \(\text{first}\) followed by a number of \(\text{prev}\) ‘s to the formula in all possible ways: \(\text{first } F, \text{ next } F, \text{ prev } F, \text{ first next } F, \text{ first next prev } F\), and so on. The correctness of TiSLD-resolution is based on the idea that the value of a given formula can be expressed in terms of the values of its canonical instances. Note that sub-goals obtained from an open-end goal are always canonical.

Formally, a TiSLD-derivation of \(P \cup \{G\}\) is a sequence of triples with the following form:

\[
\langle G_0, C_0, \theta_0 \rangle, \langle G_1, C_1, \theta_1 \rangle, ... \]

where \(G_0, G_1, ...\) are goal clauses and \(G_0 = G; C_0, C_1, ...\) the variants (up to renaming) of canonical instances of program clauses in \(P\); and \(\theta_0, \theta_1, ...\) the substitutions.

Suppose that at a step of a proof procedure, we have had the TiSLD-derivation

\[
E_0, E_1, ..., E_i
\]
where $E_i = \langle G_i, C_i, \theta_i \rangle$, $k = 0, ..., i$. Particularly, suppose we have

$$E_i = \langle \langle \langle A_0, ..., A_i, A_{i+1}, \rangle, (A \leftarrow B_0, ..., B_{m-1}), \theta_i \rangle \rangle$$

and at the next step, $A_i$ is the selected tl-atom in the goal $G_i$, via the computation rules and $A_i \theta_i = A \theta_i$ with the mgu (most general unifier) $\theta_i$, then

$$E_{i+1} = \langle \langle \langle A_0, ..., A_{i-1}, B_0, ..., B_{m-1}, A_{i+1}, ..., A_{i+1}, \theta_i, C_{i+1}, \theta_{i+1} \rangle \rangle \rangle$$

Now the TiSLD-derivation we obtain is

$$E_0, E_1, ..., E_i, E_{i+1}$$

For any sequence associated with a successful TiSLD-derivation, we have that for some $n \geq 0$, $G_n = \langle \langle \langle \rangle \rangle \rangle$, in which case the sequence has length $n$ with the last element $\langle \langle \langle \rangle \rangle \rangle$.

TiSLD-resolution given as above does not apply to Chronolog($Z$) with choice predicates. If the same choice predicate appears more than once at different stages of a derivation, it must be guaranteed that only one actual choice will be made at each moment in time, because choice predicates are supposed to represent single-valued relations. Otherwise, inconsistencies may arise and we can no longer trust the outcome of the derivation.

For instance, suppose that we are given a temporal logic program in which the predicate $p$ is true of the terms $a$ and $b$ at all moments in time. However, we cannot allow both of the tl-atoms first next $p(a)$ and first next $p(b)$ to appear in the same derivation. We must guarantee that the choice for $p$ is unique at any given moment in time. Therefore, we impose extra conditions on TiSLD-derivations to ensure that choices are made consistently. The resulting proof procedure will be referred to as ‘the revised TiSLD-resolution’.

Let $P$ be a Chronolog($Z$) program with choice predicates and $G$ is a goal. The revised TiSLD-derivation of $P \cup \{G\}$ now includes a non-decreasing sequence $S_0, S_1, ...$, of sets of rigid tl-atoms in which only choice predicates appear, as well as sequences of goals, program clauses, and substitutions as before. $S_0$ is always the empty set. $S_i$ at the $i$th step of the derivation contains all the tl-atoms for choice predicates that are selected up to step $i$. At every step of the revised TiSLD-derivation, some rigid tl-atom from the current goal is selected. Depending on whether the selected atom contains a choice predicate or not, there are two cases in which the derivation may proceed:

- When the selected tl-atom does not contain a choice predicate, the derivation proceeds much in the same way as given above. In addition, we set $S_{i+1}$ to be $S \theta_i$ where $\theta_i$ is the substitution from the unification process and $S \theta_i$ is the set obtained by applying $\theta_i$ to every member of $S_i$.
- When the selected tl-atom contains a choice predicate, there are two subcases to be considered. Let first next $(k)$ $\#p(e)$ be the selected tl-atom, then the two cases are:
  1. If there are no tl-atoms in $S_i$ of the form first next $(k)$ $\#p(s)$, the selected atom is unified with the head of some rigid instance of a program clause after renaming of the variables in the rigid instance. In other words, we do not differentiate between $p$ and $\#p$ and the definition of $p$ is used to prove the selected atom. A new goal is produced by replacing the selected atom in the goal by the body of the rigid instance, and then the substitution (mgu), say $\theta_i$ from the unification process is applied to the new goal. In addition, we set $S_{i+1}$ to be $S \theta_i \cup \{\{\text{first next} (k) \# p(e)\} \theta_i\}$.

The sets $S_i$’s serve as memory in a given derivation. It is essential to keep track of all the choices made in a given derivation to avoid inconsistencies. In short, the set $S_i$ reflects the status of the warehouse with regard to choice predicates.

3. PARALLEL EXECUTION MECHANISM

The parallel execution model CHEM is based on exploitation of parallelism inherent in Chronolog programs. Various forms of parallelism can be exploited from a Chronolog program. There are three levels of parallelism: argument parallelism, OR-parallelism and AND-parallelism, and context-parallelism. As in a standard logic program, in a Chronolog program, the low level parallelism utilizes a parallel processing capability among multiple arguments through the temporal-unification operation. At the high level, OR-parallelism is parallelism in creating multiple answers and thus related to the search space; AND-parallelism is the concurrent evaluation of more than one goal in a program clause body.

Chronolog program computations at different moments, i.e. in different contexts, can be executed in parallel. This parallelism which we call context-parallelism is an inherent 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. The answers obtained from independent computations are regarded as the answers to the original goal. By doing this, we can start a non-terminating computation to generate answers to a query and at the same time exploit context-parallelism.

3.1 Execution mechanism

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

**Step 1** The initial goal is assigned to a computation process $\text{comp}$ and the process independent child-computations for context-sub-goals of $G$ (one for each moment in time) are spawned:

$$\text{comp}_0, \text{comp}_1, \text{comp}_2, \text{comp}_3, ...$$
Note that, when \( G \) is a fixed-time goal, there is only one computation that is, \( \text{comp} \).

**Step 2** Simultaneously perform a number of child-computations (context-parallelism).

**Step 3** During each computation, there is a conjunction of tl-atoms to be proved. An AND/OR tree is produced.

**Step 4** Process the AND/OR tree and search for answers.

**Step 5** Go to Step 2, continue the computation.

When processing the AND/OR tree, some tl-atom \( A \) from the goal is selected and matched against program clauses by temporal-matching and unification. Temporal-matching involves the matching of temporal operators in the selected tl-atom \( A \) and a canonical instance of a program clause, starting from the top-most clause. Then \( A \) is unified with the head of the temporally-matching program clause. 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 are more than onematching clause, we adopt a standard backtracking mechanism in the operational model.

Implementations of Chronolog rely on the underlying resolution-type proof procedure for correctness [17]. For efficiency, we must combine features of logic programming implementations (unification, backtracking) with features of dataflow implementations (associative memo-ty, tagging) such as those of Lucid implementations [2]. The proposed execution model is in particular amenable to parallel implementations on multi-processor architectures.

### 3.2 Parallel context-processes

The execution model of the temporal language Chronolog is based on the proof procedure given in Section 2. Let \( P \) be a Chronolog program and \( G \) a goal. Then, as we said before, when \( G \) is a fixed-time goal, there is only one computation; however, if \( G \) is an open-end goal, there is an infinite series of independent child-computations: \( \ldots, \text{comp}_{-2}, \text{comp}_{-1}, \text{comp}_0, \text{comp}_1, \text{comp}_2, \ldots \) that are produced for the context-sub-goals:

\[
... \text{first prev}(2) G, \text{first prev} G, \text{first} G, \\
\text{first next} G, \text{first next}(2) G, ...
\]

respectively. Note that the computation \( \text{comp} \) for \( G \) is called the parent-computation. The goal \( G \) in fact stands for the infinite series of the independent fixed-time goals \( \ldots, \text{first prev}(2) G, \text{first prev} G, \text{first} G, \text{first next} G, \text{first next}(2) G, \ldots \), and the answers obtained from independent child-computations are regarded as the answers to the goal \( G \). Therefore, to obtain all the answers of the goal \( G \), all child-computations must be executed as long as resources permit. Thus, we have a context-process clause of the following infinite form

\[
\text{comp} \iff \ldots, \text{comp}_{-2}, \text{comp}_{-1}, \text{comp}_0, \text{comp}_1, \text{comp}_2, \ldots
\]

where \( \text{comp} \) is called the head and \( \ldots, \text{comp}_{-2}, \text{comp}_{-1}, \text{comp}_0, \text{comp}_1, \text{comp}_2, \ldots \) is the body of the process.

The process is invoked by the ‘goal’ clause

\[
\iff \text{comp}
\]

The informal semantics of the process clause is ‘if the computations \( \ldots, \text{comp}_{-2}, \text{comp}_{-1}, \text{comp}_0, \text{comp}_1, \text{comp}_2, \ldots \) are all successfully completed, the computation \( \text{comp} \) is successfully completed’. Being independent of each other, these child-computations can be executed in parallel. Replacing the conjunction operator ‘\&’ by a parallel operator ‘\( || \)’ in the above process clause, we have the clause:

\[
\text{comp} \iff \ldots || \text{comp}_{-2} || \text{comp}_{-1} || \text{comp}_0 || \text{comp}_1 || \text{comp}_2 || \ldots
\]

In principle, at any time, we can execute any number of these child-computations in parallel. Therefore, real parallel context-processes of the following form can be spawned at a given moment

\[
\text{comp} \iff \text{comp}_{k_1} || \text{comp}_{k_2} || \ldots || \text{comp}_{k_j}
\]

where \( k_1, k_2, \ldots, k_j \in Z \) (\( k_i \)’s are integers). Such a computation allows us to obtain a part of answers to the original open-end goal.

In CHEM, a parallel computation process solves an open-ended goal by creating a child-computation for each context-sub-goal. Context-parallelism in this model is achieved when more than one child-computation is active at any given time.

In general, because all child-computations are independent, the order in which the execution of the computations are performed is unimportant. Therefore in the above process, the low indices of those computations can be in any order. But, we may prefer the natural order so that the results can be presented to the user sequentially.

### 3.3 Parallel OR-Processes and AND-Processes

The execution of a Chronolog program amounts to constructing and searching the AND/OR tree of this program. In a Chronolog program, for a given fixed-time goal, there exists a unique AND/OR tree which represents the complete search space of the goal.

The initial goal is assigned to an AND process, which becomes the root of the AND/OR tree. For each group of clauses, if there exists a clause whose context matches the context of the goal of an AND process and whose head is also unifiable with the goal, one OR process is spawned, thus the corresponding dataflow graph is activated to carry out the unification and reduction of the matching clause. When head unification succeeds in an OR process, the reduction proceeds to the clause body by spawning one AND process for each body literal and then reducing them either sequentially or concurrently depending on the run-time data dependencies between the body literals. All the graphs (represented as groups of clauses) that include a matching context with a given goal’s context and are unifiable with the goal are activated and thus multiple OR processes are spawned simultaneously, therefore OR-parallelism is achieved. Variable bindings of an OR process are arranged to be entirely independent from its parent AND process so that OR-parallelism can be efficiently implemented on a highly distributed multiprocessor system.
When an AND process succeeds, it merges the argument variable bindings received from its child OR processes. The AND process then sends the variable bindings to its sibling AND processes which share the corresponding variables. A successful OR process merges the results received from its child AND processes, and sends them to the parent process.

4. THE CHRONOLOG VIRTUAL MACHINE

The execution model of Chronolog is built on top of an intermediate virtual machine, namely Chronolog Virtual Machine (CVM). The CVM extends a dataflow execution model for Prolog [24] to support Chronolog; it exploits the low level parallelism, the argument parallelism, through temporal unifications.

In our method, in executing a Chronolog(Z) program, we first compile the program into a CVM program, which is represented as a set of dataflow graphs. Each graph in the CVM program is responsible for a Chronolog(Z) procedure. The nodes in a graph are dataflow operators. All the nodes whose operands are available can be executed in parallel. Most of the nodes correspond to clause argument operations (particularly through unification). So, the parallelism exploited within a graph is called argument parallelism.

That is, when a Chronolog(Z) program and a query (treated as a goal) are given, a compilation procedure will first be performed. In the compilation procedure, the Chronolog(Z) program, as a source program, is compiled into a CVM program, i.e. a set of CVM graphs, and the query is compiled into a CVM graph. When the CVM program is running, the answers or part of answers to the query are generated.

4.1 Data structures

We define CVM data structures in terms of data areas and variable representation. The main data areas of CVM are:

- **CA** (the code area): it statically stores compiled Chronolog(Z) programs, which are represented as individual dataflow graphs.
- **SA** (the structure area): structured terms created during unification and constant terms are stored in SA.
- **BEA** (the binding environment area): binding environments (BEs) (see below for its definition), which record all the variable bindings produced during unification, are stored in BEA.
- **TA** (the target area): it contains choice-points and procedure call information.
- **WA** (the warehouse area): it stores intermediate results which may be reused during the entire computation.

CVM handles three types of variables:

- **global variables**: a variable that has its first appearance in the clause head but not in a structured term is called a global variable;
- **dummy variables**: a variable that appears only once in the clause body and does not appear in the clause head is called a dummy variable;
- **local variables**: all variables other than global and dummy variables are called local variables.

In CHEM, a variable can be bound to any term. Let \( V \) be a variable and \( T \) a term (a structure term or another variable). Then we call \( V/T \) a variable-instance pair and say that the variable \( V \) is bound to the term \( T \).

- **BE (Binding Environment)**: a BE is a set of variable-instance pairs. For example, \( \{X/f(a, g(b)), Y/Z\} \) is a BE which contains two variable-instance pairs.
- **TBE (Temporary Binding Environment)**: a BE which is newly generated by a unification operation is called a temporary binding environment.
- **CBE (Current Binding Environment)**: an existing BE which is active throughout the evaluation of a process is called a current binding environment.

A binding environment BE is created only when one or more variables are bound during unification. Dereferencing a variable \( V \) is simply to find its corresponding pair \( V/T \), and then to read \( T \), due to the distributability of the CVM model [24]. When a dummy variable is bound to a term, it is simply replaced by the term, and there is no effect on the binding environment.

4.2 Instruction set

CVM has a finite set of primitive operators (instruction identifiers), denoted by \( \Omega \). According to requirements of temporal unifications, we have

\[
\Omega = \{\text{call, check, collect, construct, create, decompose, decrement, export, increment, instantiate, switch, unify}\}
\]

For any operator \( \lambda \in \Omega \), there is only one operation code \( \text{op}_\lambda \) representing the operator itself, and if \( \lambda \neq \mu \), then \( \text{op}_\lambda \neq \text{op}_\mu \). Therefore the set of all operation codes is also finite, denoted by \( \Gamma \). In our discussion, the operators themselves will be represented by their operation codes, i.e. \( \Gamma = \Omega \).

A virtual instruction is a tuple of the following form:

\[
(\text{op}_\lambda \cdot \ldots \cdot \Theta... \Theta)
\]

where \( \text{op}_\lambda \in \Gamma \), \( \cdot \) and \( \Theta \) denote input and output operands respectively.

That is, a virtual instruction, or a CVM instruction, consists of an operation code (or op-code), which represents a type of primitive operation, and several operands denoted by symbols \( \cdot \) and \( \Theta \) which are treated as input and output ports respectively. A CVM instruction contains at least one \( \cdot \) and at least one \( \Theta \). Each \( \cdot \) or \( \Theta \) represents an abstract position which will be occupied by an actual operand in a real CVM instruction.

Apart from the operands as input or output ports, each instruction has a special operand, known as an instruction trigger, for sequential control purposes.

In the following, we give a description of the instruction set. Note that, for simplicity, the trigger of each instruction is omitted in the description.
(1) \textit{decompose}: (\text{decompose} \cdot \Theta \Theta)

Depending on the input operand, the type of instructions are classified into two modes as follows:

\begin{align*}
&\text{(decompose } [\text{el}, \text{el}_2, ..., \text{en}] \text{ el } [\text{el}_2, ..., \text{en}] \text{ ()}) \\
&\text{(decompose } \text{X } \text{H } \text{T } (\text{X}/[\text{H}|\text{T}])\text{)}
\end{align*}

The three outputs of a decompose instructions are called head, tail and environment outputs respectively. It tries to decompose the input operand as the follows:

- If the term is a list or a structure, say \([\text{el}, \text{el}_2, ..., \text{en}]\), the instruction decomposes it into a head \text{el} and a tail \([\text{el}_2, ..., \text{en}]\) and sends them to the corresponding two outputs. It also sends an empty TBE (\{\}) to its environment output.
- If the term is a variable, say \(x\), the instruction generates two new variables \(\text{H}\) and \(\text{T}\) and a TBE \((\text{X}/[\text{H}|\text{T}])\).
- If the term is a dummy variable, the instruction sends a dummy variable to both the head and tail outputs, and an empty TBE to its environment output. If the input term is neither a list, a variable nor a dummy variable, the instruction generates a ‘fail’ signal to be send to the code area \text{CA}.

(2) \textit{unify}: (\text{unify} \cdot \Theta \Theta)

A unify instruction has two inputs and two outputs. In the execution of the instruction, a common instance of the two input operands, say \text{term1} and \text{term2}, is delivered into the instance output and an environment is delivered to the TBE output. The actual format of the unify instructions is as follows:

\begin{align*}
&\text{(unify } \text{term1} \text{term2 instance TBE)}
\end{align*}

A unify instruction, according to different input situations, can implement the standard unification algorithm, and generates an instance and an environment.

(3) \textit{construct, check, export} and \textit{collect}

The four types of instructions have the same form as follows:

\begin{align*}
&\text{(op}_k \cdot \Theta \Theta)
\end{align*}

Their actual formats are:

\begin{align*}
&(\text{construct } \text{F e } [\text{el}, ..., \text{en}] \text{ F(e, el, ..., en)}) \\
&(\text{construct } \cdot \text{H T } [\text{H}|\text{T}]) \\
&(\text{check TBE1 TBE2 TBE3 CBE}) \\
&(\text{export BE1 BE2 BE3 BE}) \\
&(\text{collect term1 term2 term3 term})
\end{align*}

Construct instructions are used for structures or lists. A construct instruction constructs a new structured term. In the second mode of the instruction, the functor of the list is identified as ‘\(\cdot\)’, which is normally omitted in the list representation.

In a check instruction, up to three inputs are received from the environment outputs TBEs of unify or decompose instructions. This instruction merges the TBEs and sends a new CBE as the output.

In an export instruction, the inputs are binding environmentments (either TBEs or CBEs). The instruction selects the variable-instance pairs from the input BEs and merges them into a new BE to be sent to the instruction output.

A collect instruction produces the first variable input as its output if the instruction trigger is a constant; it collects all the input into a stream if the instruction trigger is variable.

(4) \textit{instantiate}: (\text{instantiate} \cdot \Theta \Theta)

The actual format is:

\begin{align*}
&(\text{instantiate } f(\text{X}_1, ..., \text{X}_n) \text{ CBE } f(\text{T}_1, ..., \text{T}_n))
\end{align*}

An instantiate instruction receives a term and a CBE pointer which is normally connected either to a check instruction, or to the CBE output of a \textit{call} instruction (see below). It attempts to instantiate the variables appearing in the term with their binding instances in the CBE, and sends the instantiated term to the output.

(5) \textit{create}: (\text{create} \cdot \Theta)

The actual format is:

\begin{align*}
&(\text{create } \text{ACTIVATION-SIGNAL } \text{X})
\end{align*}

A create instruction receives an activation signal and create a unique variable symbol (i.e. a variable) to be sent to the output.

(6) \textit{call}: (\text{call} \cdot \Theta \Theta \Theta)

In a call instruction the last output operand is a CBE. Apart from the output CBE, the number of input operands is equal to the number of the output operands. The instruction represents a procedure call. Its actual format is

\begin{align*}
&(\text{call } \text{I}_1 ... \text{I}_n \text{ O}_1 ... \text{O}_n \text{ CBE})
\end{align*}

where each \(I_i /O_j\) corresponds to a process argument. A call instruction transfers the argument values \(I_1 \ldots I_n\) from the caller process to the callee process, and dynamically sets the argument returning pointers \(O_1 \ldots O_n\) in the callee process. These points will directly point to the callee’s sibling AND processes.

The above instructions implement the operations in the standard logic programming as well as in the temporal logic programming. There are also other instruction forms which are special for the implementation of temporal logic. Such instructions include \textit{decrement}, \textit{increment} and \textit{switch} stated below.

(7) \textit{decrement and increment}

The form of these instructions is:

\begin{align*}
&(\text{op}_k \cdot \Theta)
\end{align*}

Both the input and output are integers in these instructions. A decrement instruction always takes an integer input, typically a time tag, and produces an integer output which is the difference of the input integer subtracted from 1. An increment instruction receives an integer and produces an integer output that is the sum of the input integer and 1.

(8) \textit{switch}: (\text{switch} \cdot \Theta \Theta \Theta)

A switch instruction is used to index the clauses in a proce-
4.3 Constructing CVM dataflow graphs

In CVM, an instruction node (or simply a node) is the instruction itself. For each input operand of a node, there is an arc entering into the node; for each output operand of the node, there is an arc coming from the node. Let node1 and node2 be two different instruction nodes. We introduce the following denotations:

- \( \langle node1(k), node2(i) \rangle \) represents an arc from node1 to node2 where the \( k \)th operand (as an output) of node1 is as the \( i \)th operand (as an input) of node2.
- \( \langle node1(k),_i \rangle \) represents an arc coming from the \( k \)th operand (as an output) of node1.
- \( _, node2(i) \rangle \) represents an arc entering into the \( i \)th operand (as an input) of node2.

Some instruction operands may be pre-occupied by certain terms so that they become ever-ready operands. The compiler is responsible for producing these operands, based on the information of the source program. We adopt an approach to explicitly represent the term on the arc. For example, \( \langle node1(k), john, node2(i) \rangle \) means that the \( k \)th operand of node1 is pre-occupied by the name ‘john’.

We now give the definition of CVM dataflow graphs as follows:

- A CVM dataflow graph is a pair \( \langle I, A \rangle \) constructed based on the rules given below, where \( I \) is a set of instruction nodes and \( A \) is a set of arcs and, for any \( \langle a(k), b(i) \rangle \), \( a(k),_i \rangle \) or \( _, b(i) \rangle \in A, a, b \in I \).

In our model, a CVM dataflow graph is responsible for a procedure in a Chronolog(2) program. Here we do not intend to give a complete language specification for the compiler, but present the constructing rules that are useful in meeting the requirements in generating dataflow graphs from source programs. These rules are divided into four groups which are applied to the procedure, clause head, clause body and clause output respectively.

1. **Procedure**: the clauses in a procedure are compiled into subgraphs branched by a `switch` instruction. The `switch` instruction switches on the input time-tag to the clauses which have different time-tags. Its output arcs are connected to the time-tag arcs of corresponding instructions. If there are no side-effect built-in predicates, the subgraphs with the same time-tag will be executed in OR-parallel.

2. **Clause head**: for clause head, if an argument in the head is a ground term, structure or an empty list, it is unified with the corresponding incoming value from the caller literal using a `unify` instruction. A structure argument whose elements are required in the clause body should be decomposed after its unification in the head. If an argument is a list containing some elements that are constants or required in the body, its incoming value should be decomposed. When an argument is a stand-alone variable, its incoming value should be sent straight to the body literal where the variable is used, if the clause has a body. For each variable that appears more than once in the head, these multiple occurrences should be unified with each other. A `unify` instruction is also needed for the unification of each constant with its occurrences in the head. A `check` instruction always follows one, two or three `unify` instructions in a recursively defined clause.

3. **Clause body**: if there are no `check` instructions used, the clause arguments can be sent directly to the body, if there is a clause body. Otherwise, when `check` instructions are involved, each argument should be instantiated before being transferred to the non-recursive body literals. No `instantiate` instructions are needed between the head and a recursively called literal.

A `create` instruction is used for each local variable to generate a variable symbol at run-time upon each activation. It can be activated by any of the incoming values in a way that its input is connected to one of the clause inputs. For a non-recursive clause, each variable that appears only once in the clause body is represented by a dummy variable.

Each structured term appearing in the clause body should be constructed by `construct` instructions. If a body literal is not a built-in predicate, it is represented by a `call` instruction, labeled with the literal name. Built-in predicates are treated as the same as virtual instructions.

4. **Clause output**: if no `check` instructions are used, there should be a clause output corresponding to each head argument. When one or more `check` instructions are used, each clause argument should be instantiated before being sent to the output. The CBE from each body literal, and from the last `check` are joined together by one or more `export` instruction.

All the instruction nodes whose operands and instruction triggers are available can be executed simultaneously. Each node, after execution, may put a new result token on its output arcs, and thus fire the following nodes. If the execution fails, a ‘fail’ signal for the associated graph is immediately generated and reported to the code area where instructions are stored so that no more instruction nodes from the same graph are allowed for execution. Only when the graph is activated again may the ‘fail’ signal be disabled.

4.4 An example

We consider the following temporal logic program that specifies the `rotate` predicate to generate all possible rotations of a given list.

```prolog
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).
```

The first `rotate` clause says that the initial value of the list to be rotated is provided from the standard input. The second `rotate` clause is used to rotate the previous value of the list-
to obtain the new list, rotated one position to the right.

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 the term \([0,1,2,3,4]\) is supplied as an input value, in other words, the goal \text{first input}([0,1,2,3,4]) succeeds. The symbol \(^\wedge\) is a directive which tells the Chronolog execution system 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 answer to the original goal is then a substitution with \(L\) replaced by \([0,1,2,3,4]\).

Now suppose that we are given the following goal:

\(- \text{next rotate}(L)\).

Because the goal is open-end, we have the parallel context-process clause as follows:

\[
\text{comp} \Leftarrow \text{comp}_0 || \text{comp}_1 || ... || \text{comp}_i || ...
\]

where \(\text{comp}_0, \text{comp}_1, ...\) are the child-computations for the context-sub-goals

\(\text{first next rotate}(L), \text{first next next rotate}(L), ...\)

respectively. This implies that all the child-computations at different contexts can be simultaneously spawned.

Figure 1 illustrates an example of a dataflow graph which is compiled from the program. In the dataflow graph, \(t'\) represents as a variable and all the output ports of the \text{switch} instruction are open. Thus, the first two clauses of this program are triggered for execution. The first two clauses defined in \text{rotate} have exclusive relationships with regards to time-tags, in other words, no more than one clause can unify with the goal which has a time-tag. The whole procedure \text{rotate} is also known as determinate procedure due to this exclusiveness. The \text{switch} instruction switches on 0 and \(t\) according to \(t'\) from a given goal. If the goal is open-ended, \(t'\) is represented as a sequence of time-tags from 0 onwards.

When the goal is switched to the second clause, two new variables are generated, by two \text{create} instructions, and sent to the \text{construct} instruction, which produces a new list to call recursively the \text{rotate} procedure.

In general, a goal at time \(t\) can be evaluated by reducing simultaneously a number of goals in the same form as the original goal but with different time-tags \(t_i (< t)\). The original goal is solved when all the subsequent goals, which are reduced in parallel with the goal, are solved. These subsequent goals are also solved in the same fashion as the original goal. This style of execution is, in fact, demand-driven,
where a demand is sent, in the opposite direction of the data being sent, from the consumer of the data to the producer.

5. WAREHOUSE

CHEM uses a warehouse (a blackboard mechanism) as an associate memory to store the results of the previous computations, and as a means for communication between independent computations under different contexts. The early ideas about warehousing can be found in the context of dataflow computation for the language Lucid [2]. A similar concept is also referred to as memoing in the logic programming context [22].

The warehouse (WA) can be viewed as a temporal database that represents some results of previous computations at a time, which we want to keep for further computations while the program is running. For each fixed-time tl-atom goal, when an answer to the goal is found, we can store the answer into the warehouse in the form of its TiSLD derivation.

5.1 Reuse principle

Consider the following Chronolog program which specifies the predicate s, which is true of the running sums of factorials over moments in time, 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 f(1).
next f(X) <- f(Z), i(C), X is Z*(C+1).
first i(1).
next i(X) <- i(Y), X is Y+1.

we may have the warehouse as follows:

WA = {first next(4) s(153), first next(5) s(873), first next(6) s(5913), first next(7) s(46233), first next(4) f(120), first next(5) f(720), first next(6) f(5040), first next(7) f(40320), first next(4) i(5), first next(5) i(6), first next(6) i(7), first next(7) i(8)}

In fact, the warehouse can also be viewed as a set of program clauses (facts). At a given time, each clause of the set represents a fact which has been obtained from previous computations. Assume that we want to prove the goal

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

Since the tl-atoms first next(7) s(46233), first next(7) f(40320) and first next(7) i(8) are all computed earlier and have been stored in the warehouse, using these results and evaluating the second program clause, we can easily obtain the answer N = 409113 to the above goal (i.e. the sum \( \sum_{i=1}^{9} i! \)).

Thus, in CHEM equipped with a warehouse, our inference algorithm is as follows:

Give a fixed-time tl-atom goal <- G.

- Check if there is any clause (fact) in the warehouse WA which can be temporal-matched with G. If so, invoke the unification algorithm to obtain results.
- Sent the results to the parent process, and exit.
- Otherwise, check the program clauses in the given program to obtain results.

Thus, the concept of the revised TiSLD-derivation can be changed to the following form. A TiSLD-derivation of \( P \cap \{ G \} \) is a sequence of triples with the following form:

\[ \langle G_0, C_0, \theta_0 \rangle, \langle G_1, C_1, \theta_1 \rangle, ... \]

where \( G_i \) is a goal clause and \( G_0 = G \); \( C_j \) is the variant (up to renaming) of canonical instance of a program clause in \( P \) or a clause in the warehouse; and \( \theta_0, \theta_1, ... \) the substitutions.

5.2 Management strategies

The warehouse maintains computed tl-atoms and also provides temporal-matching to given temporal goals. Modifying the warehouse from time to time is important for an efficient use of the resources of a working system. An ideal warehouse management scheme should not only store all the distinct tl-atoms which have been computed, but also it should be able to dispatch the stored result in a time much shorter than the time they are recomputed. This obviously requires a large associative memory and also slows down the matching process. An efficient space-saving garbage collection scheme is, therefore, required. Faustini and Wadge [2] proposed a heuristic approach, called retirement plan, for the dataflow language Lucid.

According to the retirement plan, a computed tl-atom is stored in the warehouse with a global retirement age. The warehouse is swept periodically to discard those tl-atoms that have reached their retirement age. It is in general not an easy task to predict which stored atoms will be needed later and for how long. However, there is no harm in discarding a tl-atom that might be needed later, because it can be recomputed from the program at the expense of repeated computations, should a need arise.

Faustini and Wadge [2] adopted an retirement age scheme in which the age of a data item stored in the warehouse is the number of garbage-collecting sweeps it has survived since it was last used. The age of a data item is initially 0. Every time there is a demand for its value, its age is reset to 0. It is claimed that, with some improvements, the retirement age scheme has proved to be extremely successful for Lucid programs.

In CHEM, according to the feature of temporal logic programming, we propose an algorithm, called the ‘warehouse modification algorithm’ (or WM-algorithm), to manage the warehouse. WM-algorithm can naturally deal with store-demand of a computed result as well as the retirement plan based on the contexts for Chronolog.

For a given Chronolog program, its execution environment is given and defined. We make the following assumptions:
• In a real parallel context-process, each processor performs a child-computation for a context-sub-goal such as first next (k) s(M). We call k the context of the child-computation. Let \( m_\text{A} \) denote the maximum context of the tl-atoms with a pure atom A in the real parallel context-process.

• The number of the processors is given and denoted by \( n_p \) Each processor deals with a child-computation for a context-sub-goal in the parallel context-process and continues to fetch next child-computation to the one with the maximum context after it finishes the child-computation.

• For each pure atom A, there is a number, called the recurring-dependency number, associated with the atom. The definition of recurring-dependency numbers is as follows: Suppose that a program clause has A in its head as well as in its body. If, in the head, there are applications of \( \text{next}_k \) to A and in the body the minimum number of applications of \( \text{next}_k \) to A is \( j \), then the recurring-dependency number of A in this clause is \( k-j \). The maximum recurring-dependency number, denoted by \( r_A \), in all clauses with A in the head is called the recurring-dependency number of A in the given program.

Let \( N_A = m_A - n_p - r_A \). We use \( N_A \) as the retirement-control number for the pure atom A. We also employ the following list representation for the warehouse given in the previous section:

\[
\begin{align*}
\text{WA} = & \{(s \ 4 \ (153)) \ (s \ 5 \ (873)) \ (s \ 6 \ (5913)) \\
& (s \ 7 \ (64233)) \\
& (f \ 4 \ (120)) \ (f \ 5 \ (720)) \ (f \ 6 \ (5040)) \\
& (f \ 7 \ (40320)) \\
& (i \ 4 \ (5)) \ (i \ 5 \ (6)) \ (i \ 6 \ (7)) \ (i \ 7 \ (8))
\end{align*}
\]

List representation for the warehouse is not only simple, but also very convenient for performing temporal-matching operations. Using the above representation, a computed tl-atom is a list as \((A \ T \ L)\), where A is a pure atom, T is the context, i.e. the number of \( \text{next}_k \)s, and L is also a list which corresponds to the list of variables of the predicate A.

The abstract version of WM-algorithm is given as follows:

A computed tl-atom \((A \ T \ L)\) is stored in the warehouse WA if \( T \geq N_A \).

A stored tl-atom \((A \ T \ L)\) is discarded from the warehouse WA if \( T < N_A \).

We show the execution mechanism of WM-algorithm as below. Let \( w_i \) denote the content of the warehouse WA at the moment in time \( i \). Assume that the time 0 is the start time of the computation which we want to attempt. For any computation, the set of all pure atoms in the Chronolog program is given, say \( \{A_1, \ldots, A_n\} \), and we always let \( w_0 = \{\} \), i.e. when starting a computation we initiate the warehouse as an empty set. Suppose \( w_i \) has been obtained, we then use the following steps to find \( w_{i+1} \) at the next moment in time:

**Step 1** Let \( k = i \) and \( w_{i+1} = w_i \).

**Step 2** Find \( N_A \).

**Step 3** Check all elements with \( k \) in the warehouse, i.e. \( w_{i+1} \). If the context of such an element is less than \( N_A \), discard it from \( w_{i+1} \).

**Step 4** Check all results with \( k \) which is just computed at the moment in time. If the context of such an element is not less than \( N_A \), add it into \( w_{i+1} \).

**Step 5**\( k = k+1 \).

**Step 6** If \( k \leq s \), go to Step 2; otherwise stop, input \( w_{i+1} \).

Informally, the WM-algorithm can also be represented as a meta-Chronolog(2) program shown below. \( n \) is the retirement-control number which can be obtained by the formula given above. Actually, different pure atoms may have different retirement-control numbers. For the sake of simplicity, we only consider the case in which there is only one pure atom and \( n \) is its retirement-control number. Here we also omit the details about producing the sets \( D \) and \( E \).

\[
\begin{align*}
\text{first}\ w() = \{\}. \\
\text{w}(S) & \leftarrow \text{prev}(Q), \ \text{N is control number}, \\
D & \text{is the set of all elements whose context} < N \text{in } Q, \\
E & \text{is the set of all results which are computed and whose contexts} \geq N, \\
S &= Q - D + E.
\end{align*}
\]

where predicate \( w(S) \) denotes that the set \( S \) is the warehouse, - and + are the set difference and union respectively.

Now we consider the \( \text{fib} \) program. For the predicate \( \text{fib}, \ r_{\text{fib}} = 2 \). Suppose we want to prove the goal

\[
\leftarrow \text{next}(8) \ \text{fib}(M).
\]

Assume that we have 16 processors, i.e.

\( n_p = 16 \)

and, in an ideal case, we assume that the time spent for each call to the procedure \( \text{fib} \) defined in the Section 1.2 is 1 unit time. We conduct an experiment as follows.

We assume that at the beginning, i.e. time 0, the contexts of the child-computations performed by processors are shown in Table 1 (the first context is 8 due to the use of \( \text{next}(8) \)):

<table>
<thead>
<tr>
<th>processor</th>
<th>context</th>
<th>processor</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>8</td>
<td>p9</td>
<td>16</td>
</tr>
<tr>
<td>p2</td>
<td>9</td>
<td>p10</td>
<td>17</td>
</tr>
<tr>
<td>p3</td>
<td>10</td>
<td>p11</td>
<td>18</td>
</tr>
<tr>
<td>p4</td>
<td>11</td>
<td>p12</td>
<td>19</td>
</tr>
<tr>
<td>p5</td>
<td>12</td>
<td>p13</td>
<td>20</td>
</tr>
<tr>
<td>p6</td>
<td>13</td>
<td>p14</td>
<td>21</td>
</tr>
<tr>
<td>p7</td>
<td>14</td>
<td>p15</td>
<td>22</td>
</tr>
<tr>
<td>p8</td>
<td>15</td>
<td>p16</td>
<td>23</td>
</tr>
</tbody>
</table>

At time 8, processor \( p_1 \) completed its first child-computation and obtained the answer \( \text{fib} \ (8) \). At time 7, the warehouse \( w_{8,7} = \{\} \). At time 8, since \( n_{\text{fib}} = 23 - 16 - 2 = 5 \) and \( 8 > 5 \), the result \( \text{fib} (8) \) is stored into the warehouse. Therefore, we have

\[
w_8 = \{(\text{fib} (8))\}.
\]
At time 9, processor \( p_1 \) has fetched the next child-computation with the context 24, the processor \( p_2 \) completed its first child-computation and obtained the answer \((\text{fib} \ 9 \ (34))\), and since \( N_{\text{fib}} = 24 - 16 - 2 = 6 \) and \( 9 > 6 \), the result \((\text{fib} \ 9 \ (34))\) is stored in the warehouse. Therefore, we have

\[
w_9 = \{(\text{fib} \ 8 \ (21)) \ (\text{fib} \ 9 \ (34))\}.
\]

At time 10, processor \( p_3 \) has fetched the next child-computation with the context 25, the processor \( p_3 \) completed its first child-computation and obtained the answer \((\text{fib} \ 10 \ (55))\), and since \( N_{\text{fib}} = 25 - 16 - 2 = 7 \) and \( 10 > 7 \), the result \((\text{fib} \ 9 \ (34))\) is stored in the warehouse. Also, using the results stored in the warehouse, the processor \( p_{12} \) completed its first child-computation and obtained the answer \((\text{fib} \ 19 \ (2584))\), and since \( 19 > 7 \), the result \((\text{fib} \ 19 \ (2584))\) is stored in the warehouse. Therefore, we have

\[
w_{10} = \{(\text{fib} \ 8 \ (21)) \ (\text{fib} \ 9 \ (34)) \\
(\text{fib} \ 10 \ (55)) \ (\text{fib} \ 19 \ (2584))\}.
\]

Continuing the computation, we obtain

\[
w_{12} = \{(\text{fib} \ 12 \ (144)) \ (\text{fib} \ 19 \ (2584)) \\
(\text{fib} \ 20 \ (4181)) \ (\text{fib} \ 21 \ (6765)) \\
(\text{fib} \ 22 \ (10946)) \ (\text{fib} \ 23 \ (17711))\}.
\]

At time 13, processors \( p_3, p_5, p_6 \) completed the child-computations with the contexts 24, 25, 13, respectively. At this time, \( N_{\text{fib}} = 33 - 16 - 2 = 15 \), so the result \((\text{fib} \ 13 \ (233))\) does not need to be stored, and \((\text{fib} \ 12 \ (144))\) is discarded from the warehouse. Therefore we obtain

\[
w_{13} = \{(\text{fib} \ 19 \ (2584)) \ (\text{fib} \ 20 \ (4181)) \\
(\text{fib} \ 21 \ (6765)) \ (\text{fib} \ 22 \ (10946)) \\
(\text{fib} \ 23 \ (17711)) \ (\text{fib} \ 24 \ (28658)) \\
(\text{fib} \ 25 \ (46368))\}.
\]

At any later moment in time, all processors can use the results, which are computed and stored in the warehouse, to obtain the answer to the child-computations executed by themselves. On average, for completing a child-computation, each processor only needs to call the third program clause 4 times.

In CHEM, an improved version of the MW-algorithm is also used. In the improved version, the results of computations from system-defined predicates and facts are never stored in the warehouse (as they are usually easy to compute), and the retirement-control number for a tl-atom is weighted inversely with the cost of its computation. It is in some cases cheaper to compute low-cost tl-atoms than to store them in the warehouse and cause extra overhead and increased memory size. However, it is still too early to report on the performance of this scheme and more benchmark programs need to be run under CHEM for extensive analysis.

We can employ one or more processors to deal with the warehouse. Note that tl-atoms for choice predicates are never discarded from the warehouse during a given computation.

6. CONCLUSION

CHEM is a process model for the parallel implementation of Chronolog programs based on dataflow computation. One or more clauses in a procedure are represented as an abstract dataflow graph, 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 form of parallelism offered by temporal logic programming is known as context-parallelism whereby computations at different contexts can be performed in parallel by creating parallel context-processes. The communication between contexts is controlled by time-dependencies which result from the use of temporal operators in dataflow graphs. Sequential execution can be performed to prevent the system overflow when the amount of parallelism exploited becomes overwhelming.

The modelsupports argument parallelism through parallel processing of multiple procedure arguments. A distributed variable binding scheme is used to allow OR-parallelism to be exploited in the dataflow environment. 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 warehouse is also essential for a correct implementation of choice predicates. 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.

The basic execution model supporting all forms of parallelism but context-parallelism has been implemented in Occam2 on a transputer system [23]. We are currently working on the tagging scheme to support context-parallelism. We will then simulate a tagged token dynamic dataflow machine, such as the Manchester Dataflow Machine [5], to run our model in order to evaluate how much more speedup the context-parallelism can offer. The support of Restricted AND-parallelism can also be incorporated in CHEM by taking advantage of dataflow graph annotation. Our long term plan is to implement the model on a real multiprocessor system with a large number of processing elements or on a network of workstations.

ACKNOWLEDGEMENTS

This work has been supported in part by an Australian Research Council (ARC) Grant and a Macquarie University Research Grant (MURG).

REFERENCES

17. M. A. Orgun and W. W. Wadge Chronolog admits a complete proof procedure. *Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 120–135, Université Laval, Québec City, Québec, Canada, April 1993