VERIFYING MULTRAN PROGRAMS WITH TEMPORAL LOGIC

WANLI MA
Computer Sciences Laboratory
Research School of Information Science and Engineering
The Australian National University, Canberra, ACT 0200, Australia
E-mail: ma@cslab.anu.edu.au

and

MEHMET A. ORGUN
Department of Computing, Macquarie University
Sydney, NSW 2109, Australia
E-mail: mehmet@mpce.mq.edu.au

ABSTRACT
Multran is a high-level programming language based on a Linda-like tuple space to coordinate concurrent transactions. Transactions can be written in any language as long as they satisfy their pre-conditions and post-conditions. Multran has an intuitional presentation and enjoys a temporal logic semantics which can be used for verifying properties of programs. A Multran program can be executed in a parallel, sequential, or distributed manner based on available resources, and its correctness can be reasoned about by temporal logic. In particular, we show that temporal logic can be used to reason about the safety and liveness properties of Multran programs.

1. Introduction

Multran is a high-level parallel programming language based on multiset transformation and transaction programming paradigm. A multiset, or bag, is just like a set except that it may have multiple occurrences of its elements. A transaction is a piece of self-contained programming code, procedure or function, which has the properties of ACID\textsuperscript{10} (Atomicity, Consistency, Isolation, and Durability). Multran has an intuitional presentation and enjoys a temporal logic semantics which can be used for verifying properties of programs.

In Multran, the control and computation parts of a program are separated. A Multran program is composed by concurrent transactions as fundamental actions and a tuple space on which the actions take place. The elements of the tuple space are called tuples. An action may happen whenever its execution condition is satisfied. It will consume certain tuples from the tuple space, execute its operations, and generate new tuples and inject them back into the tuple space for future processing. The computational model resembles a succession of chemical reactions, as in the so-called chemical abstract machine\textsuperscript{5,4}, in which the elements of the tuple space are consumed and generated by actions. The
specification of the control part of a Multran program is called a declaration or skeleton, and the computation part is called a transaction program.

As a language for parallel programming, Multran has a formal background for program verification, for it is the only way to achieve robust programs. Temporal logics are very useful to reason about the notion of dynamic change through time, and are suitable for modelling parallel and distributed computations. They are widely used in program specification and verification, even as programming languages. A temporal logic deductive system is therefore used to provide a formal basis for verifying properties of Multran programs.

Multran is deeply rooted in Linda, GAMMA, and Unity. To summarize, Multran inherits (i) the idea of coordination and tuple space from Linda, (ii) the chemical reaction metaphor from GAMMA to control the process of computation, and (iii) the separation of formal verification from program presentation from Unity.

In addition, Strand (later PCN) and GLU are also coordination style languages. The former uses a simplified Prolog programming language as the coordinator; its computation parts, so-called foreign data and foreign code, are written in C and/or Fortran. The latter is based on an intensional programming language Lucid to coordinate a group of C and/or Fortran functions.

Compared to other approaches, Multran has an intuitional presentation, and yet it is firmly based on a formal background. A Multran program can be executed in a parallel, sequential, or distributed manner based on available resources, its correctness can be reasoned about by temporal logic, and its dynamic behavior can be displayed by a visualization toolset.

The rest of the paper is organized as follows. The syntax and semantics of Multran are explained in Section 2 with some example programs. A brief introduction to temporal logic is given in Section 3, and Multran’s temporal model is given in Section 4. In Section 5, we prove the correctness of the example programs. We conclude the paper with a brief summary and future work.

2. Multran Programming Language

Multran has two independent parts: (i) a programming part to write programs and (ii) a proof system to prove the properties of a given program. We mainly focus on the verification part in the paper; programming aspects are described elsewhere.

The programming part of a Multran program consists of six sections:

1. **Manifest** section is used to define macros and/or constant symbols. For

---

*We emphasize the “background” because the formal part can be separated from the program presentation.*
example, if we have the definition $M=5$, then $M$ will be written as 5 whenever it is encountered. This section is optional and defined symbols are valid globally.

2. **Tuples** section declares all possible tuple types which may appear in the tuple space. We use a type system and syntax similar to those of the C programming language (without pointer types) for tuple declarations. The declaration only specifies the possible tuple types. The number of declared tuples, when, and where they may enter the tuple space depend on the sequence of execution, and they cannot be predicted in advance. Declared tuple names are also valid globally.

Tuples in Multran resemble struct's in C and record's in Pascal. A tuple name is not a tuple, which is an instance of the tuple type. When there is no confusion or it can be distinguished by context, a tuple name is used for (i) the type of the tuple, (ii) a tuple instance of the type, and (iii) all the tuples of the type together.

3. **Initialization** section sets up the initial state of the tuple space. The initialization could be passive, assigning values to tuples, or active, calling one or more procedures.

4. **Reaction rules** section consists of a number of reaction rules, which operate on the tuple space and coordinate computational chunks. A reaction rule takes the form of

$$x_1 + x_2 + \cdots + x_n \text{ leads to } y_1 + y_2 + \cdots + y_m \text{ by } T \text{ when } f(x_1, x_2, \cdots, x_n)$$

where $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots$, and $y_m$ are tuples, whose types are declared in the **Tuples** section, $T$ is a transaction name, and $f(x_1, x_2, \cdots, x_n)$ is a boolean function. The rule means whenever the tuples $x_1, x_2, \cdots, x_n$ are all currently in the tuple space and the function $f(x_1, x_2, \cdots, x_n)$ evaluates to TRUE, (i) the tuples $x_1, x_2, \cdots, x_n$ are selected and consumed, (ii) the transaction $T$ is executed, and (iii) the new tuples $y_1, y_2, \cdots, y_m$ are generated and injected back into the tuple space by transaction $T$. As far as reaction rules are concerned, these three actions are indivisible.

Both by and when qualifiers can be omitted if the transaction used is null and/or the condition is trivially TRUE. If the number of operators in a transaction is very small (especially in fine-grain parallelism), the whole transaction can be encapsulated by a pair of curly braces and directly put into the reaction rule.

There may be some common tuples among $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_m$. As a result, more than one tuple of a certain type may be needed
for the reaction or some selected tuples are sent back to the tuple space without any change, for example, "x+x leadsto x+y." To distinguish different appearances of the same type of tuples, the "^" operator is used, for example, "x^1+x^2 leadsto x^3+y." Note that the numbers here never mean the order of the tuples. A curly-brace around a tuple name means all the tuples of this type are selected together.

A transaction may not necessarily consume all the tuples on the left-hand-side of its reaction rule. We use operator "!" to denote that the tuple is just read by the rule but not consumed, and it is still available in the tuple space. Note the difference between "x leadsto x+y by T" and "!x leadsto y by T." Before and after the execution of either rule, the tuple space is the same. The former rule consumes the tuple x, which means that x is not available during the execution of T and thus other rules are prevented from testing x. A new tuple of x, perhaps the same as the consumed one, is re-generated and placed back into the tuple space after the execution of T. The latter rule does not consume x so that it is still available for any other rule during the execution of T.

5. **Termination** section gives conditions such that whenever any of them is satisfied, the corresponding final action will be taken and the program will then terminate. In a reactive program, which does not terminate, there will not be a termination section.

6. **Transactions** specify pre-conditions, the language used, and post-conditions of the transactions referred to by reaction rules. For example, consider the following transaction:

   prod: |token|>0 // C // |token|' = |token|-1, |msg|' = |msg|+1

prod is the name of the transaction, "|token|>0" is its pre-condition and "|token|' = |token|-1, |msg|' = |msg|+1" its post-condition. C means that the transaction is written in the C programming language. The ' operator means the values after the execution of the transaction.

We do not provide the actual definitions of transactions in the paper. They can be considered as C functions, Pascal functions or procedures, and/or Fortran subroutines or subfunctions etc. with the enhancement that they have the ACID properties.

The execution of a Multran program proceeds as follows: before the termination condition is satisfied, all the reaction rules are fairly chosen and tested.

---

^If other rules which deal with tuple x change the tuple space, the result of the two rules could be different.
Whenever the *reaction condition* of a reaction rule holds, i.e., the tuples needed by the reaction rule are currently in the tuple space and the boolean function of its when qualifier, if exists, evaluates to **true**, then the corresponding transaction is invoked. By *fairness*, we mean that any reaction will eventually happen if its reaction condition is continuously satisfied. It is a weak fairness and it corresponds to Manna's fairness condition\(^{24}\). The test of a reaction condition is atomic, which means that it will lock all the tuples needed before an actual test takes place. If all the needed tuples cannot be locked, locking fails and the test is suspended and retried later.

In Multran programs, a transaction is an "operator" which is executed in one step. The number of operators in a transaction reflects its *granularity*. We can obtain different granularities, from *fine-grain* to *coarse-grain*, by adjusting the granularity of each transaction.

We give three examples below.

**Example 1 (producer-consumer)** The producer produces one message, a string of at most MAX characters, at a time, and the consumer consumes one message at a time. **Both producer and consumer are autonomous. The only constraint on them is the capacity of the repository (tuple space) where the messages are temporarily stored. We assume the capacity is N in our example. The producer will continue producing messages as long as the total number of messages is less than N, and the consumer will consume messages whenever they are available.** "The Multran program is given in figure 1." □

manifest

\[
MAX = 1024;\]

tuple

\[
\text{boolean } \text{token};
\text{char } \text{msg}[\text{MAX}];
\]

initialization

\[
N^* (\text{token}=1);
\]

reactionrules

\[
\text{token leadsto } \text{msg by prod}
\]

\[
\text{msg leadsto } \text{token by cons}
\]

transaction

\[
\text{prod: } |\text{token}| > 0 \text{ // } \text{C} \text{ // } |\text{token}'| = |\text{token}| - 1, |\text{msg}'| = |\text{msg}| + 1;
\]

\[
\text{cons: } |\text{msg}| > 0 \text{ // } \text{C} \text{ // } |\text{token}'| = |\text{token}| + 1, |\text{msg}'| = |\text{msg}| - 1;
\]

Fig. 1. The Multran Declaration of Producer-consumer
In this example, we are not concerned with the order in which messages are consumed (and in the proof in the Section 5). That any message will be eventually consumed is guaranteed by the fairness principle of Multran.

From now on, we drop manifest, tuples, and transactions sections whenever there is no confusion.

**Example 2 (Dutch flag)** We have an array of \( n \) elements, each of which is either Red, White, or Blue. A program is needed to re-arrange their positions so that all Red elements come before White ones, and all White elements before Blue ones. Each element is represented by a tuple in the tuple space. Every tuple has the form of \((x, y)\) where \( x \) is the sequence number of the element in the array and \( y \) is the color of the element. See Figure 2 for the program.

\[
\begin{align*}
\text{reaction rules} \\
(i, r) + (j, w) & \xrightarrow{} (i, w) + (j, r) \quad \text{when } (i > j) \\
(i, w) + (j, b) & \xrightarrow{} (i, b) + (j, w) \quad \text{when } (i > j) \\
(i, r) + (j, b) & \xrightarrow{} (i, b) + (j, r) \quad \text{when } (i > j) \\
\text{termination} \\
on(\forall (i, r), (j, w), (k, b) : i < j < k) & \text{ do } \text{stop;}
\end{align*}
\]

Fig. 2. The Program of Dutch Flag

The above two examples do not have transactions. The next example shows the use of transactions to solve the meeting scheduler problem.

**Example 3 (Meeting scheduler)** We are to find the earliest common meeting time for a group of people. For the sake of brevity, we suppose that there are only three people, \( F, G, \) and \( H \) in the group. Every member of the group suggests their acceptable meeting times. Finally, the earliest common meeting time is reached. Figure 3 shows a Multran program to solve the meeting scheduler problem. For a more detailed discussion of the problem, we refer the reader to Chandy and Misra\(^\text{b}\)(pp. 13–18).

In the program, the tuple \( \text{time} \) holds current suggested time for the meeting. It will be modified (changed) by \( F, G, \) and \( H \) according to their own agenda. The transaction \( \text{F.time} \) (resp. \( \text{G.time} \) and \( \text{H.time} \)) withdraws time and then matches it to his/her own agenda. If the \text{time} is acceptable, it is not changed and \( \text{F.changed} \) is set to \text{FALSE}; otherwise a new time is placed into the tuple space and \( \text{F.changed} \) is set to \text{TRUE}. The common meeting time will be reached when \( \text{F.changed}, \text{G.changed}, \) and \( \text{H.changed} \) are all set to \text{FALSE}.

More formally transaction \( \text{F.time} \) executes the function \( f \):

\[
f : \text{int} \to \text{int}.
\]
initialization
    time=0;
    F_changed = G_changed = H_changed = true;

reaction rules
    time+F_changed+!G_changed+!H_changed leadsto time+F_changed
      by F.time when (G_changed==TRUE || H_changed==TRUE);
    time+G_changed+!F_changed+!H_changed leadsto time+G_changed
      by G.time when (F_changed==TRUE || H_changed==TRUE);
    time+H_changed+!G_changed+!F_changed leadsto time+H_changed
      by H.time when (F_changed==TRUE || G_changed==TRUE);

termination
    on(F_changed==false && G_changed==false && H_changed==false)
      do output;

transactions
    F_time: time=r//C//time=r & F_changed==FALSE ||
            time=f(r) & F_changed==TRUE
    G_time: time=r//C//time=r & G_changed==FALSE ||
            time=g(r) & G_changed==TRUE
    H_time: time=r//C//time=r & H_changed==FALSE ||
            time=h(r) & H_changed==TRUE

Fig. 3. The Program of Meeting Scheduler

The result of \( f(t) \) is a time acceptable to \( F \) to have the meeting according to
the current suggestion \( t \), that is, for any \( t \) and \( f(t) \) is acceptable to \( F \)
while any other time \( r \) \((t < r < f(t)) \) is not acceptable. Functions \( g \) and \( h \)
are defined accordingly.

In fact, we can define a unique transaction my_time instead of F_time, G_time, and H_time. The transaction executes the function \( \phi \):

\[
\phi : \{F, G, H\} \times int \rightarrow int, \quad \text{or} \quad \phi : \{F, G, H\} \rightarrow (int \rightarrow int).
\]

If we apply \( \phi \) to \( F, G \) and \( H \), then we obtain \( \phi(F) \equiv f \), \( \phi(G) \equiv g \), and \( \phi(H) \equiv h \).

We keep the different transactions, F_time, G_time, and H_time, in the paper
to simplify our proof in Section 5. The result of the program in figure 3 is the
realiest time \( u \) that satisfies the condition \( u = f(u) = g(u) = h(u) \).

3. Temporal Logic

Temporal logic comes from modal logic\(^{6,9}\) which studies the notions of necessity and possibility. Temporal logic studies time-dependent properties of certain
problems such as causality, historical necessity, and the notions of events and actions. In temporal logic, the value of a formula depends on an implicit time parameter.

Temporal logic is widely used in computer science to model the dynamic behavior of certain problems, and to express temporal properties such as safety, liveness, precedence and response in a natural and succinct way. There are many kinds of temporal logics, determined by the structure of their underlying collection of moments in time (e.g., linear or branching time, discrete, continuous or dense time and so on), and collection of temporal operators. Gotzhein\textsuperscript{15} presents a short introduction to temporal logic. In this paper, we adopt Manna-Pnueli temporal logic\textsuperscript{24} which is based on a linear, discrete, and non-ending timeline starting from 0. In other words, the collection of moments in time is modeled by the set of natural numbers with its usual ordering relation, $<$. The temporal logic is based on first-order (predicate) logic with some temporal operators. Formulas from predicate logic are called state formulas. A temporal formula is a state formula governed by temporal operators. We use the following temporal operators:

$$\Box, \Diamond, \mathcal{U}, \mathcal{W}.$$  

Their informal meanings are:

1. $\Box p$ says that $p$ is TRUE from now on. Read $\Box$ as always or henceforth.

2. $\Diamond q$ says that $q$ will be TRUE eventually.

3. $p \mathcal{U} q$ says that $p$ is TRUE and holds until $q$ eventually becomes TRUE.

4. $p \mathcal{W} q$ says that $p$ is TRUE and holds until $q$ eventually becomes TRUE or $p$ holds permanently if $q$ cannot become TRUE. $\mathcal{W}$ is called weak until.

In our temporal logic system, we do not use $\bigcirc$ (an operator to the next moment in time), because it is hard to define the exact meaning of the next state given that the reaction rules are all autonomous and the execution order of the reaction rules and the time spent for a reaction rule can be arbitrary. In addition, the next operator will also destroy the composibility of a logic system\textsuperscript{19}.

We give the semantics of the propositional subset of temporal logic as follows. The formal semantics of a temporal formula is defined on a model $M$, using a sequence of states $\sigma = s_0, s_1, s_2, \ldots$ along the time axis, where $s_i (i \geq 0)$ denotes the state at time instant $i$. State formulas can be evaluated at any individual state of the sequence, while a temporal formula should be evaluated on the sequence. Each state $s_i$ can be considered as a collection of propositions which
have the value TRUE at s₁, or equivalently, as a mapping between propositional variables and boolean values.

Formally, supposing σ = s₀s₁s₂s₃ ..., ∈ Μ, the meaning of temporal formulas is defined as follows (read iff as “if and only if”):

- \((σ, j) \models p\) iff \(p\) is TRUE in \(s_j\) if \(p\) is a state formula;
- \((σ, j) \models \neg p\) iff \((σ, j) \models \neg p\);
- \((σ, j) \models p \land q\) iff \((σ, j) \models p\) and \((σ, j) \models q\);
- \((σ, j) \models \Box p\) iff \((σ, k) \models p\) for every \(k \geq j\);
- \((σ, j) \models \Diamond p\) iff \((σ, k) \models p\) for some \(k \geq j\);
- \((σ, j) \models p \mathcal{U} q\) iff there is a \(k \geq j\), such that \((σ, k) \models q\), and for every \(i, j \leq i < k, (σ, i) \models p\);
- \((σ, j) \models p \mathcal{W} q\) iff \((σ, j) \models p \mathcal{U} q\) or \((σ, j) \models \Box p\).

The operators \(\Box, \Diamond, \mathcal{U},\) and \(\mathcal{W}\) are not independent of each other. In fact, we can use \(\mathcal{W}\) as a primitive operator and define the other operators as:

\[\Box p \equiv p \mathcal{W} \text{FALSE} \quad \Diamond p \equiv \neg \Box \neg p \quad p \mathcal{U} q \equiv p \mathcal{W} q \land \Diamond q\]

If \((σ, j) \models p\), we say that the model \(σ \in Μ\) satisfies \(p\) at position \(j\); if a formula \(p\) holds at position 0 of a model \(σ\), i.e. \((σ, 0) \models p\), we write \(σ \models p\) and say that the model \(σ\) satisfies the formula \(p\); and if a formula \(p\) is satisfied in every model \(σ \in Μ\), it is valid and we write \(Μ \models p\), or \(\models p\) for short.

A deductive system consists of a set of axioms, say \(A\), and a set of rewriting rules, called inference rules, which govern the deductive process. We do not deal with the issue of the completeness and soundness of a deductive system in this paper. For details, we refer the reader to the literature21,24,26,18. For any given temporal formula \(p\), if we can have a sequence of rewritings from the axioms by a sequence of applications of inference rules which leads to \(p\) (reasoning by syntax), then \(p\) is called a theorem. We can be sure that it is valid provided that our deductive system is sound (i.e., it is based on sound inference rules). We write \(A \vdash p\), or simply \(\vdash p\), to mean that \(p\) can be proved in our deductive system from given \(A\).

Here we give some of temporal logic axioms and inference rules. For more details, we refer readers to the corresponding papers24,15. Supposing \(p\) and \(q\) are temporal logic formulas, we write:

\[p \Leftrightarrow q\quad \text{for}\quad \Box[(p \to q) \land (q \to p)],\]

and
A0 axioms and tautologies of underlying logic
A1 □p ⇒ p
A2 p ⇒ ◦p
A3 □□p ⇔ □p
A4 ◦□p ⇔ ◦p
A5 ◦□□p ⇔ ◦□p
A6 □◦□p ⇔ ◦□p
A7 □p ⇒ p
A8 ◦¬p ⇔ ◦¬p
A9 □(p → q) ⇒ (□p → □q)
A10 □p → □(□p)
A11 (p ⇒ □p) ⇒ (p ⇒ □p)
A12 pWq ⇒ (q ∨ (p ∧ □(pWq)))
A13 □p ⇒ pWq

Fig. 4. General Axioms of Temporal Logic

p ⇒ q for □(p → q)
p ⇒ q is a stronger version of logic implication, which is known as entailment.
Some of the axioms* of temporal logic are given in figure 4.
Inference rules are:

1. Generalization (GEN): For a state formula p which is satisfied in every state:
   
P ⊨ □p

2. Specialization (SPEC): For a state formula p, we have that
   
□p ⊨ p

3. Modus Ponens (MP): For any given formulas p_1, ..., p_n and q:
   
(p_1 ∧ ... ∧ p_n) → q, p_1, ..., p_n ⊨ q

4. Entailment Modus Ponens (EMP): For any given formulas p_1, ..., p_n and q:
   
(p_1 ∧ ... ∧ p_n) ⇒ q, □p_1, ..., □p_n ⊨ □q

*Some of them are theorems, which can be derived from axioms. For brevity, we regard all of them as axioms in the paper. Similarly, we regard all of the derived inference rules as basic inference rules.
5. Entailment transitivity (ET): For any given formulas $p$, $q$, and $\gamma$:

$$ p \Rightarrow q, q \Rightarrow \gamma \vdash p \Rightarrow \gamma; $$

6. ◊T: For any given formulas $p$, $q$, and $\gamma$:

$$ p \Rightarrow q, q \Rightarrow \diamond \gamma \vdash p \Rightarrow \diamond \gamma $$

4. Temporal Model of Multran

To facilitate temporal logic reasoning and to assign temporal semantics, we recast Multran notation to a 4-tuple $M = (T, S, R, I)$ as follows.

1. $T$ is the set of all tuples (data) possibly appearing in the tuple space. It is specified in the tuples section of a Multran program;

2. $S$ is the state of the tuple space. Its purpose is twofold. On the one hand, it designates the tuples currently in the tuple space. In short, for any tuple $t \in T$, the characteristic function $C(t) = \text{TRUE}$ if $t$ is currently present in the tuple space, i.e. $t \in TS$; otherwise $C(t) = \text{FALSE}$. For simplicity, we write $t$ for $C(t)$ whenever there is no ambiguity. On the other hand, it assigns values to tuples or the variables contained in the tuples. In other words, it maps variables to their domains;

3. $R$ is a set of reaction rules. The elements of $R$ come from reaction rules section;

4. $I (I \subseteq T)$ is the initial state of the tuple space, which is specified by initialization section of a Multran declaration.

The logic model of a Multran program is a sequence of tuple space states,

$$ \sigma = (s_0s_1s_2s_3\cdots). $$

A Multran deductive system consists of a set of axioms and a set of inference rules. Axioms come from the general axioms of temporal logic and a given Multran program. Suppose that we are given a reaction rule, say,

$$ x_1 + x_2 + \cdots + x_n \leftrightarrow y_1 + y_2 + \cdots + y_m \text{ by } T \text{ when } f(x_1, x_2, \cdots, x_n). \quad (1) $$

Also suppose that the pre-condition and post-condition of $T$ are $p$ and $q$, i.e., $\{p\}T\{q\}$, and there are no repetitive elements among $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_m$. Then we have:

$$ \left[\bigwedge_{i=1}^{n} (|x_i| \geq 1)\right] \wedge f(x_1, x_2, \cdots, x_n) \wedge p \Rightarrow \diamond\left[\left(\bigwedge_{i=1}^{n} |x_i|^{-1}\right) \wedge \left(\bigwedge_{j=1}^{m} |y_j|^{+1}\right)\right] \wedge q, \quad (2) $$
where \(|x|\) denotes the number of \(x\) tuples currently in the tuple space, \(|x|^{-t}\) denotes the number of \(x\) tuples minus \(t\), and \(|x|^{+t}\) denotes the number of \(x\) tuples plus \(t\). If there is no multiplicity of occurrences of a same type tuple, formula 2 can be written as:

\[
\left[ \bigwedge_{i=1}^{n} x_i \land f(x_1, x_2, \ldots, x_n) \land p \right] \Rightarrow \bigvee_{j=1}^{m} y_j \land q,
\]

(3)

where \(x_i\) or \(y_j\) means the tuple is in the tuple space, while \(\neg x_i\) or \(\neg y_j\) means it is not in the tuple space.

To deal with repetitive elements, we define \(x\)-type tuples and \(y\)-type tuples.

**Definition 1 (\(x\)-type tuples, \(x\)-type multiset, \(y\)-type tuples, and \(y\)-type multiset)** In a reaction rule such as

\[x_1 + x_2 + \cdots + x_n \text{ leads to } y_1 + y_2 + \cdots + y_m \text{ by } T \text{ when } f(x_1, x_2, \ldots, x_n),\]

the tuples which appear on the left hand side of leads to are called \(x\)-type tuples. They may repetitively appear on the right hand side of leads to. The multiset \(\{x_1, x_2, \ldots, x_n\}\) is called \(x\)-type multiset, because it consists of \(x\)-type tuples. The tuples on the right hand side of a reaction rule which do not appear on the left hand side are called \(y\)-type tuples. The multiset \(\{y_1, y_2, \ldots, y_m\} - \{x_1, x_2, \ldots, x_n\}\) is called a \(y\)-type multiset.

**Definition 2 (Representative set)** A representative set of a multiset \(M\) is the set obtained from \(M\) by eliminating repetitive elements. We write \(\hat{M}\) as the representative set of \(M\).

Suppose that \(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\) are the elements of the representative set of \(x\)-type multiset; \(\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m\) are of \(y\)-type multiset; and the repetition number of \(\bar{x}_i\) in the \(x\)-type multiset is \(r_i\) and \(\bar{y}_j\) is \(s_i\). Then the more general form of the formula is given as follows:

\[
\left[ \bigwedge_{i=1}^{n} (|\bar{x}_i| \geq r_i) \right] \land f(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n) \land p \Rightarrow \bigvee_{j=1}^{m} (|\bar{y}_j|^{s_j}) \land q,
\]

(4)

We also assume that a transition from one state into another state of the tuple space caused by a reaction rule only affects the tuples involved in the reaction rule. In other words, if there is a reaction rule \(r\) by which we go from state \(s_i\) to the state \(s_{i+1}\), i.e.,

\[s_i \xrightarrow{r} s_{i+1},\]

only the values and populations of the \(x\)-type tuples and the \(y\)-type tuples of the reaction rule are changed from \(s_i\) to \(s_{i+1}\). All the other tuples are unaffected.

\[\text{The number of } x\text{-type tuples which appear on the right hand side of leads to has been counted in } r_i, \text{ e.g., if } x+x+x \text{ leads to } x+y \text{ then } r_x = 2\]
by the transition. The transition is closed under stuttering\(^1^9\). In addition we assume that, once a terminal state (say \(s_n\)) is reached, e.g., in a program whose termination condition is met, we stay in state \(s_n\) indefinitely, i.e.,

\[
\sigma = (s_0s_1s_2 \cdots s_{n-1}s_n s_n s_n \cdots).
\]

We do not consider the race of reaction rules here. If a reaction rule deprives the execution of other rule(s), a race happens. In case of a race, we bunch the reaction rules into several groups according to their race conditions, and then translate the groups of reaction rules to groups of temporal logic formulas. We will discuss the racing situation in a latter paper.

5. Multran Program Verification

Safety and liveness are two fundamental temporal properties\(^1^9\) of programs. Safety ensures that something bad never happens (\(\neg\diamond\neg P\) or \(\Box P\) in temporal logic formula), while liveness guarantees that something good will eventually happen (\(\diamond Q\) or \(P \rightarrow \diamond Q\)). A specification (or program) disciplines the behavior of a computation, and the behavior is actually a safety property of a computation. In the paper, safety properties are expressed by Multran programs, which will be translated into a set of temporal logic formulas. Liveness is not always given out by a specification. Temporal logic is very useful to prove liveness properties from their corresponding safety properties\(^2^0\).

To prove the correctness of producer-consumer program, we first translate its Multran declarations in figure 1 into temporal logic formulas. Formulas will be used as additional (non-logical) axioms of the “producer-consumer” deductive system. The axioms, referred to as \(A_{pc}\), are:

\[
(\sigma, 0) \vdash (|\text{token}| = n) \land (|\text{msg}| = 0) \tag{0}
\]

\[
|\text{token}| > 0 \Rightarrow \diamond(|\text{token}|^{-1} \land |\text{msg}|^+1) \tag{1}
\]

\[
|\text{msg}| > 0 \Rightarrow \diamond(|\text{msg}|^{-1} \land |\text{token}|^+1) \tag{2}
\]

Two main properties of the producer-consumer problem are reactivity (there are always prod or cons actions, \(\Box(\Diamond\text{prod} \lor \Diamond\text{cons})\)), and progress (every produced message will be eventually consumed, that is, \(\text{prod} \Rightarrow \Diamond\text{cons}\)). These two properties can be represented by assertions on tuples token and msg, but it is natural to reason on the actions of prod and cons. We write (2) as

\[
|\text{token}| > 0 \Rightarrow \Diamond\text{prod} \quad \text{and} \quad \text{prod} \Rightarrow \Diamond(|\text{token}|^{-1} \land |\text{msg}|^+1).
\]

Besides, from \(A_{pc}\), we can also have the assertion “the total number of token’s and msg’s are \(n\),” that is, \(A_{pc} \vdash \Box(|\text{Token}| + |\text{Msg}| = n)\), which is also called an axiom in the following proof for brevity.
Thus, we have the following new set of axioms:

\[ \Box(\text{Token} + |\text{Msg}| = n) \]  
\[ (\text{Token} + |\text{Msg}| = n) \Rightarrow \Diamond \text{prod} \lor \Diamond \text{cons} \]  
\[ \text{prod} \Rightarrow \Diamond (|\text{Msg}| > 0) \]  
\[ \text{cons} \Rightarrow \Diamond (|\text{Token}| > 0) \]  
\[ (|\text{Msg}| > 0) \Rightarrow \Diamond \text{cons} \]  
\[ (|\text{Token}| > 0) \Rightarrow \Diamond \text{prod} \]

where (pc.3) comes as the consequence of formula “\( \text{prod} \Rightarrow \Diamond (|\text{token}|^{-1} \land |\text{msg}|^{+1}) \)” (recall the tautology of \( p \land q \Rightarrow p \)). Similarly, we can obtain (pc.4).

**Proposition 1 (reactivity of producer-consumer)** There will always be \text{prod} or \text{cons} actions, i.e., \( \Box (\Diamond \text{prod} \lor \Diamond \text{cons}) \).

**Proof:**

1. \( \Box (\text{Token} + |\text{Msg}| = n) \)  
2. \( (\text{Token} + |\text{Msg}| = n) \Rightarrow \Diamond \text{prod} \lor \Diamond \text{cons} \)  
3. \( \Box (\Diamond \text{prod} \lor \Diamond \text{cons}) \)  

EMP, 1, 2

**Proposition 2 (liveness of producer-consumer)** Every produced message will be eventually consumed: \( \text{prod} \Rightarrow \Diamond \text{cons} \).

**Proof:**

1. \( \text{prod} \Rightarrow \Diamond (|\text{Msg}| > 0) \)  
2. \( (|\text{Msg}| > 0) \Rightarrow \Diamond \text{cons} \)  
3. \( \text{prod} \Rightarrow \Diamond \text{cons} \)  

\( \Diamond T, 1, 2 \)

Dutch flag problem shows the properties of a terminating program and the use of universal quantifier (\( \forall \)). Recall the rules of Multran declaration in figure 2; the temporal logic formula we get has the form:

\[ (\forall i, j : 1 \leq i, j \leq n) \ (i, r) \land (j, w) \land i > j \Rightarrow \Diamond [(i, r) \land (j, w) \land i < j], \]

Note the \( i \) and \( j \) in both sides of the formula may have different values, because in the underlying temporal logic, values of terms are not required to be rigid. The new values, on the right hand side, are obtained by swapping the old values of \( i \) and \( j \). So we have “\( i > j \)” on the left hand side of the formula while “\( i < j \)” on the right. Following the convention and also for brevity, we drop the universal quantifier \( \forall \) whenever there is no confusion. Besides, as the \( i \) and \( j \) are the position indexes of the color blocks to be sorted, obviously, never will two blocks occupy the same position, i.e., \( \Box (i \neq j) \). Thus the above formula becomes:

\[ (i, r) \land (j, w) \land i > j \Rightarrow \Diamond [(i, r) \land (j, w) \land i < j] \]

(df.1)
Proposition 3 (Dutch flag) When the program terminates (no rules are applicable any further), all Red elements come first, then White ones, and then Blue ones. In other words, we have the following temporal logic formulas:

1. \( \Box \Diamond [(i, r) \land (j, w) \rightarrow i < j] \)
2. \( \Box \Diamond [(i, w) \land (j, b) \rightarrow i < j] \)
3. \( \Box \Diamond [(i, r) \land (j, b) \rightarrow i < j] \)

Proof:
Consider the first property, \( \Box \Diamond [(i, r) \land (j, w) \rightarrow i < j] \). It can be proved by case analysis. For any two elements of \((i, r)\) and \((j, w)\), we have:

• \( i < j: \Box \Diamond [(i, r) \land (j, w) \rightarrow i < j] \) holds trivially;

• \( i > j: \)
  1. \((i, r) \land (j, w) \land i > j\) given;
  2. \((i, r) \land (j, w) \land i > j \Rightarrow \Diamond [(i, r) \land (j, w) \land i < j]\) df.1
  3. \( \Box [(i, r) \land (j, w) \land i > j \rightarrow \Diamond [(i, r) \land (j, w) \land i < j]] \) def. of \( \Rightarrow \), 2
  4. \((i, r) \land (j, w) \land i > j \rightarrow \Diamond [(i, r) \land (j, w) \land i < j]\) SPEC, 3
  5. \( \Diamond [(i, r) \land (j, w) \land i < j]\) MP, 1, 4
  6. \( \Box \Diamond [(i, r) \land (j, w) \land i < j]\) GEN, 5
  7. \( \Box \Diamond [(i, r) \land (j, w) \rightarrow i < j]\) weakening, 6

The other two properties can be proved in a similar fashion.

Meeting scheduler is to find the minimum u such that \( u = f(u) = g(u) = h(u) \).
After \( t = u \), the tuples F\_changed, G\_changed, and H\_changed are all set to FALSE and the program terminates. Let F\_changed denote F\_changed = TRUE and \( \neg F\_changed \) denote F\_changed = FALSE. Similarly (\( \neg \))G\_changed and (\( \neg \))H\_changed. From the program in figure 3, we obtain the formulas (ms.0)–(ms.3):

(ms.0)
\[
(\sigma, 0) \models \text{time} = 0 \land \text{F\_changed} = \text{TRUE} \land \text{G\_changed} = \text{TRUE} \\
\land \text{H\_changed} = \text{TRUE}
\]

(ms.1)
\[
\text{time} = r \land (\text{G\_changed} \lor \text{H\_changed}) \Rightarrow \Diamond \left[ \text{time} = f(r) \land \text{F\_changed} \right]
\]

(ms.2)
\[
\text{time} = r \land (\text{F\_changed} \lor \text{H\_changed}) \Rightarrow \Diamond \left[ \text{time} = g(r) \land \text{G\_changed} \right]
\]
(ms.3)

\[
\text{time} = \text{r} \land (F\_\text{changed} \lor G\_\text{changed}) \Rightarrow \\
\quad \diamond \quad \left[ (\text{time} = \text{r} \land \neg H\_\text{changed}) \right] \\
\lor \\
\quad \left[ (\text{time} = h(\text{r}) \land H\_\text{changed}) \right]
\]

Let \(TC\) denote the termination condition of the program:

\[
TC \overset{\text{def}}{=} \neg F\_\text{changed} \land \neg G\_\text{changed} \land \neg H\_\text{changed}.
\]

The correctness of the program relies on two assertions: (i) the program will terminate, \(\diamond TC\), and (ii) the value of \(\text{time}\) will reach the value of \(u\).

In the proofs of the following results, we only show the main steps and omit some of the straightforward derivations.

**Proposition 4**

The value of \(\text{time}\) is non-decreasing: \((\text{time} = \text{r}) \Rightarrow \diamond (\text{time} \geq \text{r})\).

**Proof:**

1. From (ms.1) – (ms.3), we can obtain:

\[
\left( \begin{array}{c} 
\text{time} = \text{r} \land (G\_\text{changed} \lor H\_\text{changed}) \\
\lor \\
\text{time} = \text{r} \land (F\_\text{changed} \lor H\_\text{changed}) \\
\lor \\
\text{time} = \text{r} \land (F\_\text{changed} \lor G\_\text{changed})
\end{array} \right) \\
\Rightarrow \\
\left( (\text{time} = \text{r} \land \neg F\_\text{changed}) \lor (\text{time} = f(\text{r}) \land F\_\text{changed}) \\
\lor \\
(\text{time} = \text{r} \land \neg G\_\text{changed}) \lor (\text{time} = g(\text{r}) \land G\_\text{changed}) \\
\lor \\
(\text{time} = \text{r} \land \neg H\_\text{changed}) \lor (\text{time} = h(\text{r}) \land H\_\text{changed})
\right)
\]

\[
(5)
\]

2. The left side of formula 5 can be reduced to

\((\text{time} = \text{r}) \land (F\_\text{changed} \lor G\_\text{changed} \lor H\_\text{changed});\)

3. From the right side of formula 5 and the tautology of \(p \land q \rightarrow p\), we obtain

\((\text{time} = \text{r}) \lor (\text{time} = f(\text{r}) \lor \text{time} = g(\text{r}) \lor \text{time} = h(\text{r}));\)

According to the definition of the functions \(f\), \(g\), and \(h\), we have \(f(\text{r}) > \text{r}\), \(g(\text{r}) > \text{r}\), and \(h(\text{r}) > \text{r}\). Thus, the new right side of formula 5 is

\((\text{time} \geq \text{r});\)

4. Formula 5 becomes

\[
[(\text{time} = \text{r}) \land (F\_\text{changed} \lor G\_\text{changed} \lor H\_\text{changed})] \\
\Rightarrow \diamond (\text{time} \geq \text{r});
\]

\[
(6)
\]
5. Before time $u$ is reached, i.e., $\text{time} = r$, $0 \leq r \leq u$, at least one of $F_{\text{changed}}$, $G_{\text{changed}}$, and $H_{\text{changed}}$ will be true:

$$ (\text{time} = r) \Rightarrow \Diamond(F_{\text{changed}} \lor G_{\text{changed}} \lor H_{\text{changed}}); $$

(7)

6. Applying tautology $(p \rightarrow q) \leftrightarrow (p \rightarrow p \land q)$ to formula 7, and with the definition of "$\Rightarrow$",

$$ (\text{time} = r) \Rightarrow \Diamond[(\text{time} = r) \land (F_{\text{changed}} \lor G_{\text{changed}} \lor H_{\text{changed}})]; $$

(8)

7. Applying $\Diamond T$ to formula 6 and formula 8, we obtain

$$ (\text{time} = r) \Rightarrow \Diamond(\text{time} \geq r). $$

\[ \nabla \]

**Proposition 5 (u reached)** *The value of time will eventually reach the value of u: $\Diamond(\text{time} = u)$.*

**Proof:**

1. $(\sigma, 0) \models \text{time} = 0$, (given, ms.0);

2. $(\text{time} = r) \Rightarrow \Diamond(\text{time} \geq r)$, (Lemma 4), is the same as

$(\sigma, i) \models \text{time} = r$ iff $(\sigma, j) \models \text{time} \geq r$, for some $j, j \geq i$;

3. time is monotonic and increasing, while $u$ is limited. A position $k$ can be found, such that

$(\sigma, k) \models (\text{time} = u)$;

4. From $A2$ ($p \Rightarrow \Diamond p$), we have $\Diamond(\text{time} = u)$. \[ \nabla \]

**Theorem 6 (termination)** *The termination condition will be eventually satisfied: $\Diamond TC$.*

**Proof:**

1. $(\text{time} = r) \land (G_{\text{changed}} \lor H_{\text{changed}}) \Rightarrow

\hspace{2cm} \Diamond[(\text{time} = r \land \neg F_{\text{changed}}) \lor (\text{time} = f(r) \land F_{\text{changed}})]$ (given, ms.1);

2. Referring to the reasoning in the proof of Theorem 5, we obtain

$$ (\text{time} = r) \Rightarrow \Diamond[(\text{time} = r \land \neg F_{\text{changed}}) \lor (\text{time} = f(r) \land F_{\text{changed}})]; $$

3. Before $u$ has been reached, i.e., $0 \leq r < u$, from proposition 4 we obtain:

$$ (\text{time} = r) \Rightarrow \Diamond(\text{time} = f(r) \land F_{\text{changed}}); $$
4. When $\text{time} = u$, we have that:

$$(\text{time} = u) \Rightarrow \diamond (\text{time} = r \land \neg F_{\text{changed}}),$$

i.e., $$(\text{time} = u) \Rightarrow \neg F_{\text{changed}};$$

5. The same reasoning gives us

$$(\text{time} = u) \Rightarrow \neg G_{\text{changed}} \quad \text{and} \quad (\text{time} = u) \Rightarrow \neg H_{\text{changed}}$$

6. Put the three formulas together, we obtain

$$(\text{time} = u) \Rightarrow \diamond (\neg F_{\text{changed}} \land \neg G_{\text{changed}} \land \neg H_{\text{changed}});$$

7. From proposition 6 and the rule of $\diamond T$, we have the proof:

$$\diamond (\neg F_{\text{changed}} \land \neg G_{\text{changed}} \land \neg H_{\text{changed}}).$$

\[\nabla\]

6. Conclusion and Future Work

Multran programming language has been outlined. We have a prototype implementation of Multran running on the connection machine CM-5. A temporal logic deductive system is being developed for verifying Multran programs. In contrast to other approaches, Multran has the following features:

1. Multran uses a tuple space to coordinate a number of sequential transactions. Programmers should consider parallel programming first and then sequential programming (in transactions) instead of adding parallel facilities to a sequential program;

2. Temporal logic semantics provides a means for verifying Multran programs. The temporal deductive system is separated from programming.

3. It uses transactions as the fundamental computational unit, and thus provides fault-tolerance for Multran programs;

4. Multran is multi-paradigm, for transactions can be written in any programming language;

5. It enables the programmer to adjust the granularity of a Multran program by changing operations in transactions, for example, a transaction can do a very complex function (coarse-grain), or only a simple summation (fine-grain);

6. Its visualization tools can help the programmers to analyze, tune and debug programs;
7. It is portable. The programmer need not know the structure of the underlying architecture, let it be parallel, sequential, or networked;

8. It is very versatile. It can be used in sequential, concurrent, distributed, or parallel computations, in terminating or non-terminating computations, and computation intensive or interactive computations.

Our experience with the implementation is preliminary, and the deductive system is not yet complete to cover the situation of races among reaction rules under the multiple occurrences of the same kind of tuples. In addition, Multran does not support compositional and hierarchical program structures at the current stage. However, we believe that the results obtained so far have already demonstrated the potentials of Multran. Future work includes two main tasks: a full-blown implementation and the development of the proof system.

The major difficulties in developing the proof system have come from dealing with races between reaction conditions. Non-determinism and multisets contribute to the complexity of the proof system. We have attempted to solve the problem of proofs under race with the assumption that there are no multiple tuples of the same type, but the general answer is still unknown.

We have observed that Multran is closely related to colored Petri nets\textsuperscript{17} and linear logic\textsuperscript{14}. We believe that further research will shed light on the connection.

Finally, we hope the work on Multran can lead us to the way of separating (i) the logic of a program from its implementation, (ii) its correctness from its efficiency, and (iii) the rigid formal reasoning aspect from a practical intuitional presentation.

7. Acknowledgements

Wanli Ma thanks the Australian Government for an Overseas Postgraduate Research Scholarship (OPRS) and the Australian National University for a PhD scholarship.

The work reported in the paper is a development on the earlier papers\textsuperscript{22,23} written by W. Ma under the supervision of Prof. E. V. Krishnamurthy. The authors thank him for his earlier contribution that led to this work.

8. References


