

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, School of MPCE
Macquarie University
Sydney, NSW 2109, Australia*

ABSTRACT

A coordination style programming language, *Multran*, and its temporal logic semantics are proposed. *Multran* uses Linda-like tuple space to coordinate concurrent transactions, which could be written in any language as long as they satisfy their pre-conditions and post-conditions. It has an intuitional presentation and enjoys a temporal logic semantics for program verification. 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, temporal logic can be used to reason about the safety and liveness properties of *Multran* programs.

1. Introduction

Multran (Multiset transformation and transactions), a high-level parallel programming language based on multiset transformation and transaction programming paradigm, and its temporal logic semantics are proposed. *Multiset*, or *bag*, is 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 (*Atomicity, Consistency, Isolation, and Durability*)¹⁰.

In *Multran*, the control part and computation part 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 happens 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, so-called *chemical abstract machine*^{5,4}, in which the elements of the tuple space are consumed and generated. The specification of the control part is called a *Multran declaration* or *skeleton*, and the computation part a *transaction program*.

As a language for parallel programming, *Multran* has a formal *background*^a for verification, for it is the only way to achieve robust programs. Temporal logics^{26,15,20,24} are very useful to reason about on-going temporal properties, and are suitable for parallel and distributed situations. They are widely used in program specification and verification, even as programming languages²⁵. A temporal logic deductive system is thus used to provide formal semantics for *Multran*.

There are quite a few high-level parallel programming languages and/or paradigms. *Multran* is deeply rooted in Linda^{13,7}, GAMMA^{1,2,3}, and Unity⁸. To summarize, (i) *Multran* inherits the idea of coordination and tuple space from Linda, (ii) the chemical reaction metaphor comes from GAMMA to control the process of computation, and (iii) the separation of formal verification from program presentation as in Unity will benefit both program verification and development.

^aWe emphasize the “background” because the formal part can be separated from the program presentation.

In addition, Strand ¹² (later PCN ¹¹) and GLU ¹⁶ 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 ²⁷ to coordinate a group of C and/or Fortran procedures.

Compared to other approaches, Multran has an intuitional presentation and yet 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 by temporal logic, and its dynamic behavior can be displayed by a visualization toolset.

The rest of the paper is organized as follows: Multran notation is explained in Section 2 with some examples. 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 in Section 6.

2. Multran Programming Language

Multran has two independent parts: (i) the programming part to write programs and (ii) a proof system to prove the correctness of a program. We mainly concentrate on the verification part in the paper. The programming aspect is described elsewhere ²³.

A Multran program consists of six sections:

1. **Manifest** section is used to define macro and/or constant symbols. For example, if we have $M=5$, M will be written as 5 whenever M is met. This section is optional and the defined symbols are valid globally.
2. **Tuples** section declares all possible tuple types which may appear in the tuple space. We use a similar type system and the syntax to these of the C programming language (without *pointer* type) for tuple declaration. The declaration only specifies the possible tuple types. The number of the declared tuples, when, and where they enter the tuple space depend on the sequence of execution, and they cannot be predicted in advance. The tuple names are also globally valid.

The tuples in Multran are something like `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. **Reactionrules** 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

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

where $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots, y_m$ 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 the transaction T . From the view of reaction rules, 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 itself.

There may be some common tuples among $x_1, x_2, \dots, x_n, y_1, y_2, \dots$, and y_m . This means that more than one tuple of a certain type is needed for the reaction or some selected tuples are sent back to the tuple space without any change, for example, “ $x+x \text{ leadsto } x+y$.” To distinguish different appearances of the same type of tuples, the “ \sim ” operator is used, for example, “ $x^1+x^2 \text{ leadsto } x^3+y$.” Note that the numbers here never mean the order of the tuples. A curly-brace on a tuple name means all tuples of this type together, i.e., selecting them all.

A transaction may not 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 “ $x \text{ leadsto } x+y \text{ by } T$ ” and “! $x \text{ leadsto } y \text{ by } T$.” Before and after any one of the two rules, the tuple space is the same⁵. But the former consumes the tuple x , which means that x is not available during the execution of T and thus prevents other rules from testing x . A new tuple of x , perhaps the same as the consumed one, is re-generated and put back into tuple space again after the execution of T . The latter does not consume x so that it is still available for any other rule during the execution of T . Similarly, a tuple which could or could not be generated by a transaction will lead by a “?” operator.

5. **Termination** conditions give conditions such that whenever any of them is satisfied, the corresponding final action will be taken and the program then terminates. In a reactive program, which does not terminate, there will not be a termination section.
6. **Transactions** specify the pre-conditions, the language used, and the post-conditions of the transactions referred to by reaction rules, for example,

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

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

For brevity, we do not include transactions themselves 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 this transaction concept makes them ACID.

The execution of a Multran program proceeds as follows: before the termination condition is satisfied, all the reaction rules are *fairly* chosen and tested. 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, 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 corresponds to Manna’s fairness condition in ²⁴. The test of a reaction condition is atomic, which means it will lock all tuples needed before a real test begins. If the locking fails, the test is suspended and retried later.

From the view of a Multran program, a transaction is the same as 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.

Here we give three examples.

Example 1 (producer-consumer) *The producer produces one message, a string of at most MAX characters, at one time and the consumer consumes one message at another time. Both producer and consumer are autonomous. The only constraint on them is the capacity of the repository 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 message number 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
    boolean token;
    char msg[MAX];
initialization
    N*(token=1);
reactionrules
    token leadsto msg by prod
    msg leadsto token by cons
transaction
    prod: |token|>0 // C // |token|' = |token|-1, |msg|' = |msg|+1;
    cons: |msg|>0 // C // |token|' = |token|+1, |msg|' = |msg|-1;

```

Figure 1: The Multran Declaration of producer-consumer

We do not care the order of the consumed messages in the program, and so does the proof in the Section . 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, which in turn before Blue ones. The n elements are represented by n tuples in the tuple space. Every tuple has the form of (x, y) , where x is the sequence number of the element in the original array and y is the color of the element. See Figure 2 for the program, please.* ∇

```

reactionrules
    (i,r) + (j,w) leadsto (i,w) + (j,r) when (i>j)
    (i,w) + (j,b) leadsto (i,b) + (j,w) when (i>j)
    (i,r) + (j,b) leadsto (i,b) + (j,r) when (i>j)
termination
    on(forall (i,r), (j,w), (k,b) :: i<j<k) do stop;

```

Figure 2: The Program of Dutch Flag

The above two examples do not have transactions. The next example shows the use of transactions to find a suitable meeting time.

Example 3 (Meeting scheduler) *The problem is to find the earliest common meeting time for a group of people. For the sake of brevity, we suppose there are only three people, F , G , and H , in the group. Extension to more than three people is straightforward. Every member of the group suggests an acceptable meeting time for himself/herself. Finally, the earliest common meeting time is reached, Figure 3. For a more detailed discussion of the problem, we refer the reader to [pp. 13–18]⁸.* ∇

The tuple `time` holds current suggested time for the meeting. It will be changed by F , G , and H according to their own agenda. The transaction `F_time` (resp. `G_time` and `H_time`) withdraws `time` and then matches it to his/her own agenda. If the time is an acceptable time, it remains unchanged and `F_changed` is set to `FALSE`; otherwise, a new time is put back into the tuple space and `F_changed` is set to `TRUE`. The common meeting time will be reached when `F_changed`, `G_changed`, and `H_changed` are all set to `FALSE`.

Actually transaction `F_time` executes the function of f :

$$f : int \rightarrow int.$$

⁸If other rules which deal with tuple x change the tuple space, the result of the two rules could be different.

```

initialization
    time=0;
    F_changed=G_changed=H_changed= true;
reactionrules
    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+!F_changed+!G_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
end

```

Figure 3: The Program of Meeting Scheduler

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

In fact, we can use a unique transaction `my_time` instead of `F_time`, `G_time`, and `H_time`. The transaction executes the function ϕ :

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

If we apply ϕ to F , G , and H , 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 . The result of program in Figure 3 is the time u which satisfies $u = f(u) = g(u) = h(u)$.

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 time. It could have the value TRUE at a given instant of time, but FALSE at another instant.

Temporal logics are widely used in computer science today to express *temporal properties*, which depend on time instants during a computation. They can express properties, such as safety, liveness, precedence, and response, in a natural and succinct way. There are many kinds of temporal logics, linear, branching, or interval time, discrete time or dense time, etc. Gotzhein's paper ¹⁵ is a good introduction to temporal logic. For our purposes, we adopt Manna-Pnueli temporal logic ²⁴ which is linear, discrete, and non-negative time with an original time point 0. In other words, the time domain 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*, and so are *state predicate*, *state term* and so on. 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 is *always* or *henceforth*. $\Box p$ says that p is TRUE from now on;
2. $\Diamond q$ means q will be TRUE *eventually*;
3. $p \mathcal{U} q$ means p is TRUE and holds *until* q eventually becomes TRUE; and
4. \mathcal{W} is called *weak until* — $p \mathcal{W} q$ means p is TRUE and holds *until* q eventually becomes TRUE or p holds permanently if q cannot become TRUE.

We do not use the **next** (\bigcirc) operator in our logic system, because it is hard to define the exact meaning of next state for the reaction rules are all autonomous and the execution order of the reactions 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 system¹⁹.

Example: Let p and q are formulas, we can get temporal formulas: $\Box p$, $\Diamond q$, $\Box \Diamond p$, $\Box p \vee \Diamond q$, $\Box(p \vee q)$, $p \mathcal{U} q$, and $p \mathcal{W} q$.

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 \mathcal{M} , using a sequence of states $\sigma = s_0 s_1 s_2 s_3 \dots$ along the time axis, where $s_i (i \geq 0)$ denotes the state at time instant i . State formula 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 s_i , or equivalently, as a mapping between propositional variables and boolean values.

Formally, supposing $\sigma = s_0 s_1 s_2 s_3 \dots \in \mathcal{M}$, the meaning of temporal formulas is defined as follows (read iff as “if and only if”):

- $(\sigma, j) \models p$ iff p is TRUE in s_j if p is a state formula;
- $(\sigma, j) \models \neg p$ iff $(\sigma, j) \not\models p$;
- $(\sigma, j) \models p \wedge q$ iff $(\sigma, j) \models p$ and $(\sigma, j) \models q$;
- $(\sigma, j) \models \Box p$ iff $(\sigma, k) \models p$ for every $k \geq j$;
- $(\sigma, j) \models \Diamond p$ iff $(\sigma, k) \models p$ for some $k \geq j$;
- $(\sigma, j) \models p \mathcal{U} q$ iff there is a $k \geq j$, such that $(\sigma, k) \models q$, and for every $i, j \leq i < k$, $(\sigma, i) \models p$;
- $(\sigma, j) \models p \mathcal{W} q$ iff $(\sigma, j) \models p \mathcal{U} q$ or $(\sigma, j) \models \Box p$.

The operators of \Box , \Diamond , \mathcal{U} , and \mathcal{W} are not all independent. Actually we can only use \mathcal{W} as a primitive operator and define:

$$\Box p \equiv p \mathcal{W} \text{FALSE} \quad \Diamond p \equiv \neg \Box \neg p \quad p \mathcal{U} q \equiv p \mathcal{W} q \wedge \Diamond q$$

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

A deductive system consists of a set of axioms, say \mathcal{A} , and a set of rewriting rules, called *inference rules*, which govern the deductive process. In other words, a deductive process is defined in purely syntactical terms. We do not deal with the issue of the completeness and soundness of a deductive system in this paper. We refer readers to the related papers^{21,24,26,18}. For a 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), it 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 $\mathcal{A} \vdash p$, or simply $\vdash p$, to mean that p can be proved in our deductive system from given \mathcal{A} .

Here we list some of temporal logic axioms and inference rules. For more details, we refer readers to the corresponding papers ^{24,15}. Supposing p and q are temporal logic formulas, we write:

$$p \Leftrightarrow q \quad \text{for} \quad \Box[(p \rightarrow q) \wedge (q \rightarrow p)],$$

and

$$p \Rightarrow q \quad \text{for} \quad \Box(p \rightarrow q)$$

$p \Rightarrow q$ is a stronger version of logic implication, which is known as *entailment*.

Some of general axioms^c of temporal logic are:

| | |
|-----|---|
| A0 | axioms and tautologies of underlying logic |
| A1 | $\Box p \Rightarrow p$ |
| A2 | $p \Rightarrow \Diamond p$ |
| A3 | $\Box \Box p \Leftrightarrow \Box p$ |
| A4 | $\Diamond \Diamond p \Leftrightarrow \Diamond p$ |
| A5 | $\Diamond \Box \Diamond p \Leftrightarrow \Box \Diamond p$ |
| A6 | $\Box \Diamond \Box p \Leftrightarrow \Diamond \Box p$ |
| A7 | $\Box p \rightarrow p$ |
| A8 | $\bigcirc \neg p \Leftrightarrow \neg \bigcirc p$ |
| A9 | $\Box(p \rightarrow q) \Rightarrow (\Box p \rightarrow \Box q)$ |
| A10 | $\Box p \rightarrow \Box \bigcirc p$ |
| A11 | $(p \Rightarrow \bigcirc p) \rightarrow (p \Rightarrow \Box p)$ |
| A12 | $p \mathcal{W} q \Rightarrow (q \vee (p \wedge \bigcirc(p \mathcal{W} q)))$ |
| A13 | $\Box p \Rightarrow p \mathcal{W} q$ |

Inference rules are:

1. **Generalization(GEN)**: for a state formula p which is satisfied in every state,

$$p \vdash \Box p;$$

2. **Specialization(SPEC)**: for a state formula p ,

$$\Box p \vdash p;$$

3. **Modus ponens(MP)**: for any formula p_1, \dots, p_n and q ,

$$(p_1 \wedge \dots \wedge p_n) \rightarrow q, p_1, \dots, p_n \vdash q;$$

4. **Entailment modus ponens(EMP)**: for any formula p_1, \dots, p_n and q ,

$$(p_1 \wedge \dots \wedge p_n) \Rightarrow q, \Box p_1, \dots, \Box p_n \vdash \Box q;$$

5. **Entailment transitivity(ET)**: for any formula p, q , and γ ,

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

6. **$\Diamond T$** : for any formula p, q , and γ ,

$$p \Rightarrow \Diamond q, q \Rightarrow \Diamond \gamma \vdash p \Rightarrow \Diamond \gamma.$$

^cSome of them are theorems, which can be derived from axioms. For brevity, we treat them all as axioms in the paper. The same method is applied to basic inference rules and derived inference rules.

4. Temporal Model of Multran

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

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 tuple space. Its purpose is twofold. On the one hand, it designates the tuples currently in the tuple space. For any tuple $t \in T$, the characteristic function $C(t) = \text{TRUE}$ if t is currently present in the tuple space, i.e. $t \in \mathcal{TS}$; 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 the set of reaction rules. The elements of R come from **reactionrules** section;
4. I ($I \subseteq T$): 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,

$$\sigma = (s_0 s_1 s_2 s_3 \dots).$$

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. For any reaction rule, say,

$$\begin{aligned} x_1 + x_2 + \dots + x_n \text{ leadsto } y_1 + y_2 + \dots + y_m \\ \text{by } T \text{ when } f(x_1, x_2, \dots, x_n), \end{aligned} \quad (1)$$

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

$$[[\bigwedge_{i=1}^n (|x_i| \geq 1)] \wedge f(x_1, x_2, \dots, x_n) \wedge p] \Rightarrow \Diamond[(\bigwedge_{i=1}^n |x_i|^{-1}) \wedge (\bigwedge_{j=1}^m |y_j|^{+1}) \wedge q], \quad (2)$$

where $|x|$ means the number of x tuples currently in the tuple space, and $|x|^{-t}$ means the number of tuple x decreases t , while $|x|^{+t}$ increases t . If there is no multiplicity of occurrences of a same type tuple, Formula 2 can be written shortly as:

$$[\bigwedge_{i=1}^n x_i \wedge f(x_1, x_2, \dots, x_n) \wedge p] \Rightarrow \Diamond[\bigwedge_{j=1}^m y_j \wedge q], \quad (3)$$

where x_i or y_j means the tuple is current in 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 of Multran such as*

$$x_1 + x_2 + \dots + x_n \text{ leadsto } y_1 + y_2 + \dots + y_m \text{ by } T \text{ when } f(x_1, x_2, \dots, x_n),$$

the tuples which appear on the left hand side of leadsto are called x -type tuples. They may repetitively appear on the right hand side of leadsto. The multiset $\{x_1, x_2, \dots, x_n\}$ is called x -type multiset, because it is composed by x -type tuple. The tuples which appear on the right hand side of a reaction rule and do not appear on the left hand side are called y -type tuples. The multiset $\{y_1, y_2, \dots, y_m\} - \{x_1, x_2, \dots, x_n\}$ is called y -type multiset. ∇

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

Supposing $\tilde{x}_1, \tilde{x}_2, \dots$, and $\tilde{x}_{\tilde{n}}$ are the elements of the representative set of x -type multiset; $\tilde{y}_1, \tilde{y}_2, \dots$, and $\tilde{y}_{\tilde{m}}$ are of y -type's; and the repetition number of \tilde{x}_i in the x -type multiset is r_i^d and \tilde{y}_j is s_j . The more general form of the formula is:

$$[[\bigwedge_{i=1}^{\tilde{n}} (|\tilde{x}_i| \geq r_i)] \wedge f(\tilde{x}_1, \tilde{x}_2, \dots, \tilde{x}_{\tilde{n}}) \wedge p] \Rightarrow \Diamond[(\bigwedge_{i=1}^{\tilde{n}} |\tilde{x}_i|^{-r_i}) \wedge (\bigwedge_{j=1}^{\tilde{m}} |\tilde{y}_j|^{+s_j}) \wedge q], \quad (4)$$

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 the cases of race, we bunch the reaction rules to several groups according to the race conditions, and then translate the groups of reaction rules to groups of temporal logic formulas. We will discuss the situation in a latter paper.

5. Multran Program Verification

Safety and liveness are two fundamental temporal properties¹⁹. Safety ensures that something bad never happens ($\Box P$ in temporal logic formula), while liveness guarantees that something good will eventually happens ($\Diamond Q$ or $P \rightarrow \Diamond Q$). A specification (or program) disciplines the behavior of a computation. 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 the corresponding safety properties²⁰.

To prove the correctness of *producer-consumer*, we first convert its Multran declaration in Figure 1 to temporal logic formulas, which will be used as additional axioms of the “producer-consumer” deductive system. The axioms, referred as \mathcal{A}_{pc} , are:

$$\begin{aligned} (\sigma, 0) &\models (|\text{token}| = n) \wedge (|\text{msg}| = 0) & (0) \\ |\text{token}| > 0 &\Rightarrow \Diamond(|\text{token}|^{-1} \wedge |\text{msg}|^{+1}) & (1) \\ |\text{msg}| > 0 &\Rightarrow \Diamond(|\text{msg}|^{-1} \wedge |\text{token}|^{+1}) & (2) \end{aligned}$$

Two main properties of the producer-consumer problem are *reactivity*, which means there always are *prod* or *cons* actions, $\Box(\Diamond \text{prod} \vee \Diamond \text{cons})$, and *progress* — every produced message (by *prod*) will be eventually consumed (by *cons*), that is, $\text{prod} \Rightarrow \Diamond \text{cons}$. The two properties can certainly be represented by assertions on tuples *token* and *msg*, but it is natural to reason on the actions of *prod* and *cons*. “ $|\text{token}| > 0 \Rightarrow \Diamond(|\text{token}|^{-1} \wedge |\text{msg}|^{+1})$ ” thus is written as: “ $|\text{token}| > 0 \Rightarrow \Diamond \text{prod}$ ” and “ $\text{prod} \Rightarrow \Diamond(|\text{token}|^{-1} \wedge |\text{msg}|^{+1})$ ”. Besides, from \mathcal{A}_{pc} , we can also have “the total number of tokens and msgs are n ,” i.e., $\mathcal{A}_{pc} \vdash \Box(|\text{Token}| + |\text{Msg}| = n)$, which is also called an *axiom* in the following proof for brevity. Thus, we have the new set of axioms:

$$\begin{aligned} \Box(|\text{Token}| + |\text{Msg}| = n) & & (\text{pc.1}) \\ (|\text{Token}| + |\text{Msg}| = n) &\Rightarrow \Diamond \text{prod} \vee \Diamond \text{cons} & (\text{pc.2}) \\ \text{prod} &\Rightarrow \Diamond(|\text{Msg}| > 0) & (\text{pc.3}) \\ \text{cons} &\Rightarrow \Diamond(|\text{Token}| > 0) & (\text{pc.4}) \\ (|\text{Msg}| > 0) &\Rightarrow \Diamond \text{cons} & (\text{pc.5}) \\ (|\text{Token}| > 0) &\Rightarrow \Diamond \text{prod} & (\text{pc.6}) \end{aligned}$$

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

Proof:

$$\begin{aligned} 1 \quad &\Box(|\text{Token}| + |\text{Msg}| = n) & (\text{pc.1}) \\ 2 \quad &(|\text{Token}| + |\text{Msg}| = n) \Rightarrow \Diamond \text{prod} \vee \Diamond \text{cons} & (\text{pc.2}) \\ 3 \quad &\Box(\Diamond \text{prod} \vee \Diamond \text{cons}) & \text{EMP, 1, 2} \end{aligned}$$

^dThe number of x -type tuples which appear on the right hand side of *leadsto* has been counted in r_i , e.g., if $x+x+x \text{ leadsto } x+y$ then $r_x = 2$.

▽

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

Proof:

- | | | |
|---|---|--------------------|
| 1 | $\text{prod} \Rightarrow \Diamond(\text{Msg} > 0)$ | (pc.3) |
| 2 | $(\text{Msg} > 0) \Rightarrow \Diamond \text{cons}$ | (pc.5) |
| 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 Multan declaration in Figure 2, the temporal logic formula we get has the form of:

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

Please 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 tradition and also for brevity, we drop the universal quantifier \forall whenever there is no confusion. Thus the above formula becomes:

$$(i, r) \wedge (j, w) \wedge i > j \Rightarrow \Diamond[(i, r) \wedge (j, w) \wedge i < j] \quad (\text{df.1})$$

Theorem 3 (Dutch flag) *When the program terminates (no rules are applicable any further), we expect that all Red elements come first, then White ones, and Blue ones come last. The properties can be described by the following temporal logic formulas:*

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

Proof:

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

- $i < j$: $\Box \Diamond[(i, r) \wedge (j, w) \rightarrow i < j]$ holds trivially;
- $i > j$:

- | | | |
|---|---|--------------------------|
| 1 | $(i, r) \wedge (j, w) \wedge i > j$ | given; |
| 2 | $(i, r) \wedge (j, w) \wedge i > j \Rightarrow \Diamond[(i, r) \wedge (j, w) \wedge i < j]$ | df.1 |
| 3 | $\Box[(i, r) \wedge (j, w) \wedge i > j \rightarrow \Diamond[(i, r) \wedge (j, w) \wedge i < j]]$ | def. of $\Rightarrow, 2$ |
| 4 | $(i, r) \wedge (j, w) \wedge i > j \rightarrow \Diamond[(i, r) \wedge (j, w) \wedge i < j]$ | SPEC, 3 |
| 5 | $\Diamond[(i, r) \wedge (j, w) \wedge i < j]$ | MP, 1, 4 |
| 6 | $\Box \Diamond[(i, r) \wedge (j, w) \wedge i < j]$ | GEN, 5 |
| 7 | $\Box \Diamond[(i, r) \wedge (j, w) \rightarrow i < j]$ | 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 $\text{time} = u$, the tuples $F_changed$, $G_changed$, and $H_changed$ are all set to **FALSE** and the program terminates. Let $F_changed$ denotes to $F_changed = \text{TRUE}$ and $\neg F_changed$ to $F_changed = \text{FALSE}$, so do $(\neg)G_changed$ and $(\neg)H_changed$. From the program (Figure 3), we get:

$$(\sigma, 0) \models \text{time} = 0 \wedge \text{F_changed} = \text{TRUE} \wedge \text{F_changed} = \text{TRUE} \wedge \text{F_changed} = \text{TRUE} \quad (\text{ms.0})$$

$$\text{time} = r \wedge (\text{G_changed} \vee \text{H_changed}) \Rightarrow \Diamond \left[\begin{array}{c} (\text{time} = r \wedge \neg \text{F_changed}) \\ \vee \\ (\text{time} = f(r) \wedge \text{F_changed}) \end{array} \right] \quad (\text{ms.1})$$

$$\text{time} = r \wedge (\text{F_changed} \vee \text{H_changed}) \Rightarrow \Diamond \left[\begin{array}{c} (\text{time} = r \wedge \neg \text{G_changed}) \\ \vee \\ (\text{time} = g(r) \wedge \text{G_changed}) \end{array} \right] \quad (\text{ms.2})$$

$$\text{time} = r \wedge (\text{F_changed} \vee \text{G_changed}) \Rightarrow \Diamond \left[\begin{array}{c} (\text{time} = r \wedge \neg \text{H_changed}) \\ \vee \\ (\text{time} = h(r) \wedge \text{H_changed}) \end{array} \right] \quad (\text{ms.3})$$

Let TC denotes the termination condition of the program,

$$TC \equiv_{\text{def}} \neg \text{F_changed} \wedge \neg \text{G_changed} \wedge \neg \text{H_changed},$$

the correctness of the program relies on (i) the program will terminate, $\Diamond TC$, and (ii) the value of time will reach the value of u .

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

Lemma 1 (time non-decreasing) *The value of time is non-decreasing: $(\text{time} = r) \Rightarrow \Diamond(\text{time} \geq r)$.*

Proof:

1. From (ms.1) – (ms.3), we can get:

$$\left(\begin{array}{c} \text{time} = r \wedge (\text{G_changed} \vee \text{H_changed}) \\ \vee \\ \text{time} = r \wedge (\text{F_changed} \vee \text{H_changed}) \\ \vee \\ \text{time} = r \wedge (\text{F_changed} \vee \text{G_changed}) \end{array} \right) \Rightarrow \Diamond \left(\begin{array}{c} (\text{time} = r \wedge \neg \text{F_changed}) \vee (\text{time} = f(r) \wedge \text{F_changed}) \\ \vee \\ (\text{time} = r \wedge \neg \text{G_changed}) \vee (\text{time} = g(r) \wedge \text{G_changed}) \\ \vee \\ (\text{time} = r \wedge \neg \text{H_changed}) \vee (\text{time} = h(r) \wedge \text{H_changed}) \end{array} \right); \quad (5)$$

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

$$(\text{time} = r) \wedge (\text{F_changed} \vee \text{G_changed} \vee \text{H_changed});$$

3. From the right side of Formula 5 and the tautology of $p \wedge q \rightarrow p$, we get

$$(\text{time} = r) \vee (\text{time} = f(r) \vee \text{time} = g(r) \vee \text{time} = h(r)).$$

According to the definition of the functions f , g , and h , we have $f(r) > r$, $g(r) > r$, and $h(r) > r$. Thus, the new right side of Formula 5 is

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

4. Formula 5 becomes

$$[(\text{time} = r) \wedge (\text{F_changed} \vee \text{G_changed} \vee \text{H_changed})] \Rightarrow \Diamond(\text{time} \geq r); \quad (6)$$

5. Before time u is reached, i.e., $\text{time} = r$, $0 \leq r \leq u$, at least one of F_changed , G_changed , and H_changed will be true:

$$(\text{time} = r) \Rightarrow \Diamond(\text{F_changed} \vee \text{G_changed} \vee \text{H_changed}); \quad (7)$$

6. Applying tautology $(p \rightarrow q) \leftrightarrow (p \rightarrow p \wedge q)$ to Formula 7, and with the definition of “ \Rightarrow ”,

$$(\text{time} = r) \Rightarrow \Diamond[(\text{time} = r) \wedge (\text{F_changed} \vee \text{G_changed} \vee \text{H_changed})]; \quad (8)$$

7. Applying $\Diamond T$ to Formula 6 and Formula 8, we can get

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

▽

Theorem 4 (*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 1), is the same as

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

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 $\Diamond(\text{time} = u)$.

▽

Theorem 5 (termination) *The termination condition will be eventually reached: $\Diamond TC$.*

Proof:

1. $(\text{time} = r) \wedge (\text{G_changed} \vee \text{H_changed}) \Rightarrow \Diamond[(\text{time} = r \wedge \neg \text{F_changed}) \vee (\text{time} = f(r) \wedge \text{F_changed})]$ (given, ms.1);

2. Referring the reasoning in the proof of Theorem 4, we get

$$(\text{time} = r) \Rightarrow \Diamond[(\text{time} = r \wedge \neg \text{F_changed}) \vee (\text{time} = f(r) \wedge \text{F_changed})];$$

3. Before u has been reached, i.e., $0 \leq r < u$, from Lemma 1:

$$(\text{time} = r) \Rightarrow \Diamond(\text{time} = f(r) \wedge \text{F_changed});$$

4. When $\text{time} = u$,

$$(\text{time} = u) \Rightarrow \Diamond(\text{time} = r \wedge \neg \text{F_changed}), \text{ i.e., } (\text{time} = u) \Rightarrow \Diamond \neg \text{F_changed};$$

5. The same reasoning can give us

$$(\text{time} = u) \Rightarrow \Diamond \neg \text{G_changed} \text{ and } (\text{time} = u) \Rightarrow \Diamond \neg \text{H_changed}$$

6. Put the three formulas together, we get

$$(\text{time} = u) \Rightarrow \Diamond(\neg \text{F_changed} \wedge \neg \text{G_changed} \wedge \neg \text{H_changed});$$

7. From Theorem 5 and the rule of $\Diamond T$, we have

$$\Diamond(\neg \text{F_changed} \wedge \neg \text{G_changed} \wedge \neg \text{H_changed}).$$

6. Conclusion and Future Work

The Multran programming language has been outlined. Prototype implementation experiments have been conducted on CM-5 parallel machine. A temporal logic deductive system is also developed for verifying Multran programs. In contrast to other approaches, Multran has the following features:

1. Multran uses tuple space to coordinate a number of sequential transactions. Programmers should consider parallel first and then sequential instead of adding parallel facilities to a sequential program;
2. Formal temporal logic semantics provides a means of correctness proof for Multran programs. The proof is separated from programming. A programmer need not touch this notoriously hard aspect if he/she does not want to.
3. It uses transaction as the fundamental computational unit, and thus provides fault-tolerance to Multran programs;
4. Multi-lingual transactions make Multran multi-paradigm;
5. Its granularity can be easily adjusted by changing the operators contained in transactions, for example, a transaction can do a very complex function (coarse-grain), or only a simple summation operation (fine-grain). The changing of one transaction has nothing to do with the others;
6. Its visualization tools can help the programmers to analyze, tune and debug the programs;
7. It is a high-level portable language. A programmer need not know the structure of the underlying machine: parallel, sequential, or networked;
8. It is very versatile. It can be used in sequential, concurrent, distributed, or parallel situation, terminating or non-terminating situation, and computational or interactive situation.

Our experience of the implementation is somewhat preliminary, and the deductive proof 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, our results obtained so far have already demonstrated the great potentiality of Multran. Our future work includes two main streams: *implementation* and *proof system development*.

The difficulties we met in developing the proof system come from the dealing with race between reaction conditions. Non-determinism and multisets contribute to the complexity of the proof system. We have solved the problem of proof under race with the assumption of no multiple tuples of the same type, but the general answer is still unknown.

We have observed that Multran has a close relation to *colored Petri nets*¹⁷ and *linear logic*¹⁴, but their connections to Multran are not clear and will require further research.

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) correctness from efficiency, and (iii) the rigid formal reasoning aspect from a comfortable 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 ^{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

1. J.-P. Banâtre, A. Coutant, and D. Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer System*, (4):133–144, 1988.
2. J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, (15):55–77, 1990.
3. J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Comm. ACM*, 36(1):98–111, January 1993.
4. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
5. G. Boudol. Some chemical abstract machines. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pages 92–123. Springer-Verlag, 1993.
6. J. P. Burgess. Basic tense logic. In D. M. Gabbay and F. Guethner, editors, *Handbook of Philosophical Logic, Vol. II*, pages 89–134. D. Reidel Publishing Company, 1984.
7. N. Carriero and D. Gelernter. Linda in context. *Comm. ACM*, 32(4):444–458, 1989.
8. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
9. B. F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.
10. A. K. Elmagarmid, Y. Leu, J. G. Mullen, et al. Introduction to advanced transaction models. In Ahmed K. Elmagarmid, editor, *Database Transaction Models: For Advanced Applications*, pages 33–52. Morgan Kaufmann Publishers, San Mateo, California, USA, 1992.
11. I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: the PCN approach. *Scientific Programming*, 1(1):51–66, 1992.
12. I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.
13. D. Gelernter and N. Carriero. Coordination languages and their significance. *Comm. ACM*, 35:96–107, 1992.
14. J. Girard. Linear logic: A survey. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 63–112. Springer-Verlag, 1991.
15. R. Gotzhein. Temporal logic and application — a tutorial. *Computer Networks and ISDN systems*, 24:203–218, 1992.
16. R. Jagannathan and A. A. Faustini. The GLU programming language. Technical Report SRI-CSL-90-11, SRI International, Menlo Park, CA, USA, 1990.
17. K. Jensen. An introduction to the theoretical aspects of coloured Petri nets. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pages 230–272. Springer-Verlag, 1993.
18. F. Kröger. *Temporal Logic of Programs*. Springer-Verlag, 1987.
19. L. Lamport. What good is temporal logic. In R. E. A. Mason, editor, *Information Processing, IFIP 83*, pages 657–668. Elsevier Science Publishers B. V., North-Holland, 1983.
20. L. Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pages 347–374. Springer-Verlag, 1993.
21. J. V. Leeuwen. *Handbook of Theoretical Computer Science: Formal Models and Semantics*, volume B. Elsevier, The MIT Press, 1990.
22. W. Ma, E. V. Krishnamurthy, and M. A. Orgun. On providing temporal semantics for the gamma programming model. In C. Barry Jay, editor, *CATS: Proceedings of Computing: the Australian*

- Theory Seminar*, pages 121–132. University of Technology, Sydney, Australia, 1994.
23. W. Ma, V. K. Murthy, and E. V. Krishnamurthy. Multtran — A coordination programming language using multiset and transactions. To appear in the First International Conference on Neural, Parallel, and Scientific Computations (ICNPSC), Atlanta, USA, May 28–31, 1995.
 24. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
 25. M. Orgun and W. Ma. An overview of temporal and modal logic programming. In Dov M. Gabbay and H. J. Ohlbach, editors, *The First International Conference on Temporal Logic*, LNAI 827, pages 445–479. Springer-Verlag, 1994.
 26. N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, 1971.
 27. W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.