Concurrent Programming in T-Cham

Wanli Ma
Computer Sciences Laboratory
ANU, ACT 0200, Australia
ma@cslab.anu.edu.au

Christopher W. Johnson
Department of Computer Science
ANU, ACT 0200, Australia
Chris.Johnson@cs.anu.edu.au

Richard P. Brent
Computer Sciences Laboratory
ANU, ACT 0200, Australia
rpb@cslab.anu.edu.au

Abstract

A coordination style programming language, T-Cham, *is proposed. It is based on the paradigm of the chemical abstract machine (Cham) and transaction programming paradigm. Hierarchical tuple spaces, where the "molecules" of the Cham reside, are used to coordinate the concurrent transactions, which could be written in any language, such as C, Pascal, or Fortran etc., even T-Cham itself, as long as they satisfy their pre-conditions and post-conditions. A transaction can begin its execution whenever its execution condition is met. T-Cham has an intuitive presentation and yet a formal background. A T-Cham program can be executed in a parallel, distributed, or sequential manner based on the available resources.*

Keywords concurrent programming, parallel and distributed programming, coordination, chemical abstract machine, transaction, tuple space

1 Introduction

T-Cham (Transactions on CHemical Abstract Machine) is a coordination style programming language obtained by extending the *chemical abstract machine* (Cham) [7] with transactions. It is a successor of the programming language Multran [22]. A T-Cham program is composed of concurrent *transactions*, which have the properties of ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*) [1], as fundamental actions and *tuple spaces*, on which the actions take place, and can be executed in a parallel, distributed, or sequential manner based on the available resources, while the execution mode is transparent to programmers. The tuple spaces can be taken as chemical solutions where the floating molecules reside and interact. The elements of a tuple space, i.e., the molecules, are called *tuples*. An action may happen whenever its execution condition is satisfied. It will consume certain tuples from a tuple space, perform its task, and generate new tuples and inject them back into the tuple space for future processing. The Cham computational model resembles a succession of chemical reactions in which the elements of a tuple space are consumed and generated.

The T-Cham approach is inspired by the coordination idea [15] of Linda and hence belongs to the dichotomous paradigms. On the one hand, the control part, which is responsible for coordinating concurrence actions and communications, is based on tuple spaces. *Reaction rules* are used to specify when and which actions (transactions) can happen. On the other hand, the "computational part" could be written in any programming language, even T-Cham itself; thus, transactions can be nested, or are hierarchical. The execution of a T-Cham program starts from a special transaction called `root` — the *main transaction* of the program. It is the only one which could be exempt from termination (not really atomic). The transactions referred to by the reaction rules in a transaction are called *sub-transactions* of it. From the point of view of any transaction, its sub-transactions are atomic operators. A T-Cham program is the specification of its transactions. A transaction, which is written in the language other than T-Cham and hence does not spawn any sub-transactions in the sense of T-Cham, is called a *leaf transaction*; otherwise, a *non-leaf transaction*. A leaf transaction could be a Fortran subroutine or subfunction, a C function, or a Pascal procedure or function with the properties of ACID. A non-leaf transaction is composed of (i) the specification of its tuple space, including the types of tuples and the initial state, (ii) reaction rules, and (iii) sub-transaction interfaces — pre-conditions and post-conditions of its sub-transactions.

There are quite a few high-level concurrent (parallel or distributed) programming languages and/or paradigms, for example, Linda [2, 8], GAMMA [5] and Cham [7], Unity [9], PVM [26], Occam [20] and CSP [16], CCS [24], and Petri Nets [25]. As a newcomer, T-Cham emphasizes *simplicity*, *abstraction*, *efficiency*, and a sound *theoretical background*. The chemical reaction model makes it easy to express concurrent or parallel tasks in T-Cham; the hierarchical transaction structure is good for program abstraction and refinement; the explicit declaration

Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 31–February 2 1996.

of tuple space will help the optimization of data (tuples) and task distribution, and hence the efficient execution of T-Cham programs; finally, reaction rules, with their temporal logic interpretation, make it easy for program verification. Although we do not discuss the formal semantics of T-Cham in this paper, we would like to point out that T-Cham has a formal temporal logic background for program verification [21, 23].

The rest of the paper is organised as follows: a brief introduction of related work is given in Section 2. The basic T-Cham notations are explained in Section 3. Section 4 gives two examples, one for computational and one for reactive programming, of T-Cham programs, and Section 5 focuses on the T-Cham implementation issues, mainly in C-Linda on the AP1000 parallel computer. Section 6 considers the mappings of tuples (data) and hierarchical tuple spaces, i.e., the advanced features of T-Cham. We conclude in Section 7.

2 Related Work

T-Cham is an example of a *multi-lingual* programming paradigm. By multi-lingual, we mean more than one programming language or paradigm symbiosis in one program. The key idea to glue them together is coordination. In addition to Linda, Strand [14] (later PCN [13]) and GLU [18] are also multi-lingual programming languages. Strand uses a simplified Prolog programming language as the coordinator; its computational part, so-called *foreign data* and *foreign code*, are written in C and/or Fortran. GLU is based on an intensional programming language Lucid to coordinate a group of C and/or Fortran procedures.

The idea of using (nested) transactions for general purpose programming, in contrast to the traditional roles of transactions in database systems, is not new. Argus [19] is one such example.

Linda [2] is a well-known coordination parallel programming paradigm based on a global tuple space. There are four tuple space operators: `in`, `rd`, `out`, and `eval`. The main idea of Linda is *uncoupling*, including both space-wise uncoupling and time-wise uncoupling. Linda itself is not a full-fledged programming language but coordinates the activities written in ordinary (sequential) computational languages. Taking C-Linda as an example, the activities, or the *chunks* of computation, are written in the C programming language. They interact and communicate with each other on the tuple space by the four Linda operators.

The elegant idea of coordination and uncoupling becomes awkward when the four tuple space operators reside in a sequential host language. The sequential skeleton, i.e., the host language, forces programmers to consider its sequential control structures first instead of the concurrently accessible tuple space. In addition, the syntax structure of an existing host language also blurs the globality of the tuple space. A better way to realize the idea of coordination and uncoupling is to view the tuple space operation as skeleton, and the computational chunks as pieces of flesh, which are fitted into the skeleton. In short, rather than extending a sequential computational programming language with a parallel tuple space, we would like to attach the sequential computational chunk to a concurrent accessible tuple space.

A Cham (chemical abstract machine) [7] is an abstract machine for concurrent computations based on a chemical reaction metaphor, which was first expressed in the GAMMA model [5]. It consists of a *multiset*, which is a *set* except that it may have multiple occurrences of its elements, of *molecules* and a group of *rewriting rules*. The molecules are algebraic terms built on a given syntax, and the rewriting rules, known as *reaction rules*, transform one state of the Cham to another until no further reaction rule can be applied. A state of Cham is a finite multiset of molecules. It is also known as a *solution*, where the molecules are floating and interact with each other according to the reaction rules. This interaction changes one solution to another. When no interaction can happen any more, i.e., no reaction rule can be applied, a stable solution — the result — is arrived. Cham has no explicit termination condition and the operations behind the chemical reactions are also not specified. Its expressibility is equivalent to that of CCS process calculi [24]. It has been also used to specify a simple multiphase compiler by P. Inverardi and A. L. Wolf [17].

Unity [9] is based on Dijkstra's `do` structure [11]. It retains the assignment statements of the imperative programming paradigm but abandons its control parts. A Unity program consists of a group of assignment statements, which are executed infinitely and fairly. A statement can assign different values to different variables in one step. Unity also develops an axiomatic proof system for program verification. It is a fragment of propositional temporal logic with the fundamental operators of *unless* and *ensure*. New operators, such as *stable*, *invariant*, and *leadsto*($\mapsto$), can be defined by the two operators. Although a fix point operator, *FP*, is suggested to decide the termination point of a terminating program, the termination condition is not easy to a programmer either. One of the major contributions of Unity is separating the programming notation from its formal specification symbols (for program verification purposes), although there are one-to-one relationships between them, so that the verification is transparent to the programmers who do not like mathematical reasonings, but the correctness of the program can still be proved by some other people who like the *symbolic games*.

T-Cham is deeply rooted in Linda, Cham, and Unity. In particular, (i) T-Cham inherits the idea of coordination and tuple space from Linda, (ii) the chemical reaction metaphor comes from Cham, which is used to control the process of a computation, and (iii) the separation of formal verification from program presentation, while keeping one-to-one correspondence between them as in Unity, will benefit both program verification and development.

3 The T-Cham Programming Language: Basic Notations

Every T-Cham transaction has its own tuple space, which is created when the transaction becomes active and revoked after the execution of the transaction, to store its tuples (molecules), where the "chemical reactions" take place. The tuple space to a T-Cham transaction is the run time environment to a function or a procedure in an imperative programming language. Informally, a T-Cham transaction (recall that a program is a collection of transactions with a special one named `root`) consists of a name and a body, i.e.,

> `transaction` *my_name*
> *my_body*
> `endtrans`,

where *my_name* is the transaction name and *my_body* is the transaction body, which is composed of the following five sections[1]:

1. **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 C programming language (without *pointer* types) for tuple declarations. The declaration only specifies possible tuple types. How many of the declared tuples, when, and where they enter the tuple space depend on a particular execution, and cannot be predicted in advance. Tuple names are valid to the transaction and its sub-transactions. A tuple in T-Cham is something like a `struct` in C or a `record` in Pascal, but in T-Cham, the keyword `tuple` is used instead, for example,

```
tuples
  tuple { int A[100]; int gridsieved;
  } num;
  int token;
  fifo char msg[256];
```

The keyword `tuple` can be omitted if the tuple has only one field. A leading keyword of `fifo`, `filo`, or `random` can be used to specify the consumption order of the tuples. The default value is `random`. The above declaration defines three tuple names, `num`, `token`, and `msg`. `num` has two fields; `msg`s are consumed in first-in-first-out (`fifo`) order.

2. **Initialization** section sets up the initial state of a tuple space. The initialization could be *passive*, assigning values to tuples, or *active*, calling one or more leaf-transactions, where there could be some input operations to get data from an input device or a file, for example,

```
initialization
  [i:0..9]::token={i*2}; init_num();
```

The initialization of `token` is passive, while the call to `init_num()`, which is a leaf-transaction, to initialize tuple `num` is active. "`[i:0..9]::token={i*2}`" means that for every `i` from 0 to 9, `token={i*2}`, i.e., there are ten `token`s in the initial tuple space and they are even numbers from 0 to 18. `i` is called an *index variable*.

A third initialization method is the mapping of tuples to the tuples in the tuple space of its parent transaction or one of its child's, possibly under some required masks. See Section 6 for more details.

3. **Reaction rules** section consists of a number of reaction rules, beginning with the key word `reactionrules`. The rules operate on the tuple space of the transaction and coordinate its computational chunks — sub-transactions. A reaction rule takes the form of

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

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 the name of a transaction (known as a sub-transaction to this transaction), and $f(x_1, x_2, \cdots, x_n)$ is a boolean expression. The rule means whenever the tuples x_1, x_2, $\cdots$, and 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, $\cdots$, and x_n are selected and consumed, (ii) the transaction T is executed, and (iii) new tuples y_1, y_2, $\cdots$, and y_m are generated and injected back into the tuple space. From the point of view of the transaction which contains the reaction rule, these three actions are indivisible. Both `by`

[1] From now on, a sub-transaction is called a transaction for brevity if there is no confusion.

and **when** qualifiers of a reaction rule can be omitted if the transaction used is null and/or the condition is trivially `TRUE`.

There may be some common tuples among x_1, x_2, $\cdots$, x_n, y_1, y_2, $\cdots$, 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 (possibly without any change), for example, "`x,x leadsto x,y`." To distinguish different appearances of tuples, in the **when** condition and the body of sub-transaction T, the **$** operator is used, e.g., "`when (x$1==x$2-10)`." Note that the numbers here are only syntactic markers and do not imply any dynamic ordering. A curly-brace on a tuple name means all tuples of this type together, i.e., selecting them all.

A transaction need 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, i.e., it is still available in the tuple space. Similarly, a tuple which may conditionally be generated by a transaction is preceded by a "`?`" operator, e.g., "`!x,y leadsto ?z`."

4. **Termination** conditions give conditions such that whenever any of them is satisfied, the corresponding final action will be taken and the transaction then terminates:

```
termination
  on (|token|==0) do output();
```

For an interactive program, which does not terminate, there is no termination section in the `root` transaction. The test of termination conditions, if they exist, takes priority over that of reaction rule conditions.

5. **Sub-transactions** specify the pre-conditions and the post-conditions of the sub-transactions referred to by the reaction rules defined in the `reactionrules` section, for example,

```
subtransactions
  prod:|token|>0//|token|'=|token|-1;
```

where `prod` is the name of the transaction referred to by a reaction rule, "`|token|>0`" is the pre-condition of the transaction, and "`|token|'=|token|-1`" the post-condition. The `'` operator means the values after the execution of the transaction.

The execution of a T-Cham transaction proceeds as follows: unless a termination condition is satisfied, all of its reaction rules are *fairly* 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 it exists, evaluates to `TRUE`, the corresponding sub-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 condition. The test of a reaction condition is atomic, which means it will lock all tuples needed before a real test begins. If locking fails, the test is suspended and retried later. Of course, a T-Cham implementation could use different approaches for the test to achieve a better performance as long as they preserve the semantics.

If a sub-transaction to be executed is written in T-Cham, meaning that a new tuple space is needed to support the execution, a new tuple space is established according to the specification of the sub-transaction. The relations on the tuples of the two level tuple spaces are also established. All the actions of the sub-transaction operate on the new tuple place, which will be revoked after the execution. Section 6 gives the details of a sub-transaction's execution. From the point of view of a transaction, any of its sub-transactions is an "operator" and is executed in exactly "one step."

The number of operators in a transaction reflects its *granularity.* Different granularities, from *fine-grain* to *coarse-grain*, can be obtained by adjusting the granularity of each transaction; in addition, with the transaction concept, the fault-tolerance property of T-Cham programs is straightforward.

4 Programming in T-Cham

Two examples in this section cover both *computational* and *interactive* programs. The former gets a result from input data and then terminates, e.g., the n^{th} *Fibonacci Number.* The latter does not terminate: different parts of a program interact with each other in response to stimuli from the outside world, e.g., *Sleeping Barber.*

Fibonacci numbers are not a commonly used example for concurrent programming, especially in parallel situation, for the definition of the n^{th} Fibonacci number,

$$f(n) = \begin{cases} f(n-1) + f(n-2) & \text{if } n > 2 \\ 1 & \text{if } n = 1 \text{ or } n = 2, \end{cases}$$

suggests a very limited degree of concurrency. We choose the example because (i) an alternative definition of the formula exposes a high degree of concurrency, (ii) it is useful in the explanation of implementation techniques, and (iii) it is neither very complex nor trivial and is short.

Example 1 (Fibonacci Number) *The n^{th} and the $(n-1)^{th}$ Fibonacci numbers can be defined by the $(n-1)^{th}$ and the $(n-2)^{th}$ numbers, and so on:*

$$\begin{aligned} \begin{pmatrix} f(n) \\ f(n-1) \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f(n-1) \\ f(n-2) \end{pmatrix} = \cdots \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \end{aligned}$$

Let `coef` *be* $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ *and* `fib` *be* $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$*, then the n^{th} and $(n-1)^{th}$ Fibonacci numbers will be the matrix product of $(n-1)$* `coef`*s and one* `fib`*. The T-Cham program is in Figure 1.* ∎

```
transaction root
  tuples
    tuple {int a,b,c,d;} coef;
    tuple {int x,y;} fib;
  initialization
    init_coef(); fib={{1,1}};
  reactionrules
    coef,coef leadsto coef by mc;
    coef,fib leadsto fib by mf;
  termination
    on (|coef|==0) do output_fib();
  subtransactions
    init_coef:p0//q0; mc:p1//q1; mf:p2//q2;
endtrans

transaction init_coef
#language  C
#tuple {int a, b, c, d} coef;
  void init_coef()
    {int i,n;
     scanf("%d", &n);
     for (i=0; i++; i<n)
       coef.a=coef.b=coef.c=1, coef.d=0;
    }
endtrans

transaction mf
#language   C
#tuple {int a, b, c, d} coef;
#tuple {int x, y} fib;
  void mf()
    {fib$2.x=coef.a*fib$1.x,coef.b*fib$1.y;
     fib$2.y=coef.c*fib$1.x,coef.d*fib$1.y;
    }
endtrans

-- mc is omitted to save space
```

Figure 1: The T-Cham Program of Fibonacci Number

The tuple `coef` and tuple `fib` correspond to the two matrices. At first, i.e., the initial tuple space state of transaction `root`, there are $(n-1)$ copies of tuple `coef`s and one `fib`. The reaction rules of `root` say that two copy of `coef`s can be used to produce one `coef` by the transaction `mc`, which calculates the product of the two `coef` matrices, and one `coef` and one `fib` can be made to one new `fib`, the product of the two tuples (matrices) by `mf`. Whenever there is no `coef` left, the program can terminate and output the result — the n^{th} and the $(n-1)^{th}$ Fibonacci numbers, contained in the remaining `fib` tuple.

The leaf transactions `init_coef` and `mf` are written in the C programming language ("`#language C`" in the transactions). The former reads the number n from keyboard and then generates n copies of `coef` tuples to the `root`'s tuple space; and the latter calculates the matrix product of a `coef` and a `fib`. The `coef` and the `fib` in the transaction `mf` are not taken as arguments. They are the resources prepared for the transaction before its execution and consumed by it after the execution. We avoid to view them as ordinary arguments because different programming language has different argument passing rules, which make the "argument passing" to a transaction very confusing. Similarly, the tuples generated by a transaction are not the function value returned but the new resources. Some extra efforts are needed to achieve the mechanism in addition to a normal C compiler. Finally, "`#language`" specifies the language used to write the transaction (default is T-Cham itself) and the "`#tuple`" line is used to provide the type information of a tuple to the enhanced compiler.

`p0`, `p1`, `p2`, `q0`, `q1`, and `q2` are the pre-conditions and post-conditions of the three transactions of `init_coef`, `mc`, and `mf`. They are

$\mathtt{p0} \equiv \mathtt{TRUE}$,

$\mathtt{p1} \equiv \mathtt{coef\$1} = \begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} \wedge \mathtt{coef\$2} = \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix}$,

$\mathtt{p2} \equiv \mathtt{coef} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \wedge \mathtt{fib\$1} = \begin{pmatrix} x \\ y \end{pmatrix}$,

$\mathtt{q0} \equiv |\mathtt{coef}| = n \wedge \forall \mathtt{coef} \in \mathcal{T} : \mathtt{coef} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
$\quad \wedge |\mathtt{fib}| = 1 \wedge \mathtt{fib} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$;

$\mathtt{q1} \equiv \mathtt{coef\$3} = \begin{pmatrix} a1a2 + b1c2 & a1b2 + b1d2 \\ c1a2 + d1c2 & c1b2 + d1d2 \end{pmatrix}$;

$\mathtt{q2} \equiv \mathtt{fib\$2} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$,

where $\mathcal{T}$ denotes the tuple space. These pre-conditions and post-conditions are used for program verification purposes. If a programmer finds it is difficult to provide them for every transaction or is unwilling to do so, just simply have the conditions be trivially `true`.

Example 2 (Sleeping Barber) [2] *A barber provides hair-cutting service in his shop, where there are two doors — one for entrance and the other for exit — and N chairs for waiting customers. Only one customer can receive the service on the barber's chair at a time. When there are no customers in the shop, the barber will fall asleep on his chair; otherwise, he continuously provides hair-cutting service until no customers are left. The barber spends all his life serving customers or sleeping.*

When a customer arrives and finds the barber sleeping, he wakes up the barber and has his hair cut on the barber's chair. After the service, the customer gets out of the shop by the exit door. If the barber is busy when a customer comes, the customer will take a seat provided that there is an empty chair and wait for the barber. If all chairs are occupied, the new customer has to wait until a chair is available. The T-Cham program is given in Figure 2. ∎

```
transaction root
  tuples
    fifo boolean pin, pwt, pcut, pout;
    boolean bsp, bwk, bfin, chair;
  initialization
    [i:1..N]::chair=TRUE; bsp=TRUE;
  reactionrules
    nil leadsto pin;
    pin, bsp leadsto pcut, bwk;
    pin, chair leadsto pwt when (~bsp);
    pcut, bwk leadsto pout, bfin;
    pwt, bfin leadsto pcut, chair, bwk;
    bfin leadsto bsp when (|pwt|==0);
    pout leadsto nil;
endtrans
```

Figure 2: The T-Cham Program of the Sleeping Barber Problem

The tuple space of this program simulates the barber's shop. The tuples in the tuple space denote the states of each customer, each chair, and the barber. A `pin` in the tuple space means that a new customer is coming, `pwt` a customer is waiting on a chair, `pcut` a customer is sitting down on the barber's chair and having his hair cut, and `pout` means a customer leaving the barber's shop. `bsp` denotes that the barber is sleeping, `bwk` the barber is working, and `bfin` the barber has just finished cutting the hair of a customer. `chair` in the tuple space means a chair is available for a new customer.

The tuple `nil` is a special symbol of T-Cham. There is no real tuple called `nil` in a tuple space at all. It means a *condition* which is satisfied automatically. Thus "`nil leadsto pin`" generates a new customer, and "`pout leadsto nil`" means a customer is leaving — vanishing from the tuple space.

Initially, there are N chairs available in the tuple space (barber's shop) and the barber is sleeping.

The first reaction rule generates a new customer, and the second, if a customer (not necessarily the one just generated) finds the barber sleeping, he wakes him up and has his hair cut, i.e., makes the barber busy. The third reaction rule says that if a new coming customer finds that the barber is busy (or not sleeping) and a chair is available, he will sit down on the chair and wait for the barber. By the reaction rule four, the busy barber will finish his service to the customer who is having his hair cut so that the customer is ready to go. A waiting customer is asked to sit down on the barber's chair to have his hair cut according to reaction rule five; as the consequence, an occupied chair is available again. By the reaction rule six, when there are no waiting customers, the barber is going to sleep on his chair. The last reaction rule makes a customer who has had his hair cut disappear from the tuple space.

As there are no sub-transactions in the program, the `subtransactions` section is omitted. Other solutions of the problem can be found in Bacon [4, pp 266 – 267] and Andrews [3, pp 290 – 294]. We believe readers will find the T-Cham program given here has a more natural and intuitive presentation.

5 T-Cham Implementation

Data (tuple) distribution and task balance are the main problems of a T-Cham implementation. Hierarchical transaction structure and its corresponding hierarchical tuple spaces make it even more difficult. To avoid attacking all the difficulties in one step, our first T-Cham implementation is built on C-Linda, where there is a logical shared tuple space and the task distribution is also taken care by a Linda implementation. Although Linda uses a monolithic tuple space, the nested T-Cham tuple space structure can be fitted in by rewriting it to a flat one. We only discus the implementation of flat T-Cham programs, i.e., a `root` transaction with a number of leaf transactions, in this section.

The execution model of T-Cham on Linda comprises three independent parts: a *tester*, a group of *task generators*, and a number of *task executors*, Figure 3.

Supposing we have a group of reaction rules. One of them is R_s:

$$x_1, x_2, \cdots, x_n \ \texttt{leadsto} \ y_1, y_2, \cdots, y_m \ \texttt{by} \ T \ \texttt{when} \ f;$$

The tester continuously withdraws, or reads, tuples from the Linda tuple space. Whenever a group of

[2] We simply assume that the barber and all his customers are male.

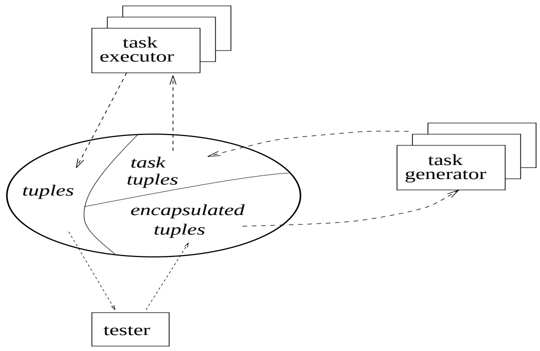


Figure 3: T-Cham Execution Model on Linda

tuples, say x_1, x_2, ..., and x_n, makes the left-hand-side (LHS) of the reaction rule R_s, and its `when` condition is also satisfied by the tuples, an encapsulated tuple, $(\text{“}R\_s\text{”}, x_1, x_2, \cdots, x_n)$, is generated.

For every reaction rule, there is a task generator, which inputs the encapsulated tuple of its form from the tuple space and generates a task tuple. Taking the task generator of reaction rule R_s for example, it repeatedly inputs the tuple of $(\text{“}R\_s\text{”}, ?arg_1, ?arg_2, \cdots, ?arg_n)$ and generates a task T for the tuple, i.e., $T(arg_1, arg_2, \cdots, arg_n)$.

Task executors are actually the processors, which execute the transactions, of a computer system. A task executor gets the active tuples from the tuple space, executes their function, and generates new tuples — for example, $y_1, y_2 \cdots, y_m$ after task T — and injects them back to the tuple space.

Task generators are quite easy to code. Every generator consists of a permanent loop, within which there are only two actions: withdrawing a correspondent encapsulated tuple from the tuple space, e.g., `in`$(\text{”}R\_s\text{”}, ?arg_1, ?arg_2, \cdots, ?arg_n)$, and then output an active tuple, `eval`$(T(arg_1, arg_2, \cdots, arg_n))$. A task executor is exactly Linda's mechanism of transforming an active tuple to passive one, i.e., task execution.

The tester uses a Rete-like network [12] for the many-object/many-pattern matching. The LHSs and conditions of every reaction rule are compiled into a testing net, and then translated to C-Linda code. Figure 4(a) gives an example of compiling a reaction rule to part of a test net. It is a straight-forward rewriting of reaction rules to a graph form. Figure 4(b) is the test net of Fibonacci number (Example 1).

The tester continuously transverses the test net so that groups of tuples can be transformed to corresponding encapsulated tuples. In every node of the net reside some tokens, which decide the state transition and the next action of the tester. In Figure 4(a), node 1 to node n are input nodes. A u token will be added to node u if a tuple x_u $(1 \leq u \leq n)$ is input (withdrawn or read). The middle level node i, if approached, will test the `when` condition, $f(x_1, \cdots, x_n)$, of the reaction rule. If the value of the test is `TRUE`, a token will be passed to node j, which is a leaf node; otherwise, the tester will try other branches. If all branches fail, the tester will input more tuples to make a transition possible. Every arrow of a test net has its own weight, for example, w_1, w_2, $\cdots$, and w_n in Figure 4(a). The default value is 1, which can be omitted. The weight w_1 means that the transition from node 1 to i needs at least w_1 tokens in node 1. Only after it has received tokens from its all predecessors can node i test the condition of f and then generate a token for node j. If a token reaches a termination node, e.g., node 4 in Figure 4(b), the tester will try to terminate the execution of the program.

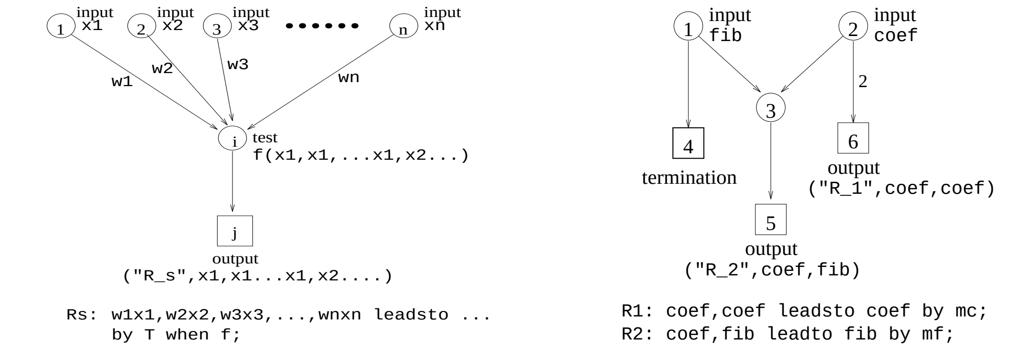
It is worthwhile to pay a special attention to the test of termination conditions, if they exist. As T-Cham has no central control, there is no way to take a "snapshot" of the global state of an execution at a time instance. For example, the condition of "`|coef|==0`", if evaluated to `TRUE` by the tester at some time, could be falsified later by a currently executing task `mc`. It suggests that the test of termination condition can only be taken under the *stable state* of an execution. By stable, we mean that there is no communication traffic among computation nodes and all the nodes are waiting for message or have finished their executions. The tester can *suggest* that a termination point has come, but an independent *arbiter* will check if the global state is stable. If yes, it terminates; otherwise, the tester will continue its execution and the suggestion until a real termination point comes. For more general discussion of distributed termination problem, we refer readers to [6, pp125–138].

The prototype execution model is conducted on an AP1000, which is a distributed memory MIMD parallel computer developed by Fujitsu. The AP1000 C-Linda was implemented by Robert Cohen for the CAP project [10]. Fortunately, AP1000 has a "`waitstable()`" system call. It helps to check the global stable state efficiently.

6 Tuple Mappings and Hierarchical Tuple Spaces: The Advanced Features

There are some very large tuples which may include natural substructures, such as a large matrices with their row, column, or sub-matrice structures. T-Cham provides the concept of *tuple mapping* to decompose a large tuple to a number of smaller pieces or *vice versa*.

The mapping from one tuple to another (or others) provides different views on a large tuple. The view can be achieved by a special data type — *mask*, which is a kind of template, or window



(a) A Reaction Rule and Its Branches in a Testing Net (b) The Testing Net of Fibonacci Number

Figure 4: The Testing Nets of T-Cham

(i.e. view), from which part of original data can be seen, or by which the original data can be transformed. In an implementation, masks may be kept as "bitmap" to save computer memory, but for programmers, a mask can be treated the same as an ordinary data tuple. A value 0 means the underlying data can not be seen, while a non-zero value is a hole on the mask so that the underlying data can be read out through it.

A tuple is a sequence of memory cells on which each field of the tuple is stored, and a mask is a window through which a part of these cells can be seen. A mask has no knowledge of the underlying tuple, and can be applied to any kind of tuples. It is the programmer's responsibility to guarantee that the data read out through the mask make sense. Two special values related to the data under masks are "`dontcare`" and "`undefined`", which are used for the sake of semantic completeness. Suppose we have a tuple `x` and a mask `m`,

```
tuple {
  char name[]="data";
  int i=15;
  float a[20]={76.8,3.5,4,...};
} x;

mask {
  {0,0,1,1}*char,
  1*int,
  {0,1,18*0}*float
} m;
```

a new tuple, `("ta",15,3.5)`, can be got by viewing the tuple `x` through the mask `m`, i.e. `x\m`, read as "tuple `x` under mask `m`."

The mappings among tuples are classified into two categories: the tuples on the same tuple space or on the different levels of tuple spaces. The former is called *horizontal mapping*, the latter *vertical mapping*. Two horizontal mappings are *decomposition*, or *heating*, which breaks down a large data tuple to smaller pieces, and *aggregate*, or *cooling*, which assembles a group of small pieces of data to make a large one. Both of the mappings are guided by some masks. For example, consider tuples `x`, `x1`, `x2`, and `x3`, and masks `m1`, `m2`, and `m3` on `x`. We can get `x1`, `x2`, and `x3` by decomposing `x`, written as "`<<x: x1\m1,x2\m2,x3\m3>>`". `x1`, `x2`, and `x3` are read out from `x` through the windows of `m1`, `m2`, and `m3` respectively. With aggregate, we can build `x` from `x1`, `x2`, and `x3`, "`>>x: x1\m1,x2\m2,x3\m3<<`". Similarly, `x1`, `x2`, and `x3` are filled into `x` through the windows of `m1`, `m2`, and `m3` respectively.

The vertical mappings establish the relations between the tuples on a tuple space and its parent's or child's tuple space, Figure 5. They can be used to initialize the child's tuple space, the *down mapping*, and return the result tuples to the parent's tuple space, the *up mapping*. Supposing in a transaction `T`, we have a reaction rule:

```
X,Y leadsto Z by t.
```

To execute the action, a T-Cham sub-transaction `t` is invoked, and `X` and `Y` are mapped (also decomposed) into `x1`, `x2`, `x3`, `y1`, and `y3` on a new tuple space correspondingly according to the specified relations between `X`, `Y` and `x1`, `x2`, `x3`, `y1`, and `y3`. After the execution of `t`, i.e., the reactions on `x1`, `x2`, `x3`, `y1`, and `y3` yielding `z1` and `z2`, which make a `Z`, a new `Z` is generated. Transaction `T`'s tuple

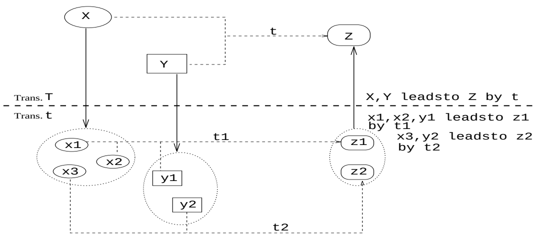


Figure 5: The Vertical Mappings of Tuples

space could have other tuples than `X`, `Y`, and `Z`, but in `t`, only the three of that tuple space can be seen. Of course, `t` may have its own private tuples on its tuple space. Similarly, they can not be seen from outside either.

x1, x2, x3, y1, and y3 can be considered as a detailed view on X and Y. In transaction T, the tuples of X and Y are indivisible and one of each together can produce an indivisible tuple Z by an atomic transaction t; but from the point of view of t, the structures of X, Y, and Z are revealed, and the indivisible action of "X,Y leadsto Z by t" is executed by some concurrent executed reactions, i.e., t1 and t2, which in turn could also have their own sub-transactions, on the different parts of the three original tuples. Just imaging a chemical reaction, on molecular level, we can have "*two hydrogen molecules and one oxygen molecule make two water molecules* ($2H_2 + O_2 = 2H_2O$)", but different actions can also be reported if we are from different points of views, such as these of atom, electron, and so on.

The mapping among tuples and hierarchical tuple spaces provide T-Cham with program abstract and refinement. They are very useful and also very nice properties for a practical programming language, both in program development and verification.

7 Conclusion and Future Work

The new coordination style programming language T-Cham is proposed and tested by our prototype implementation. It is based on the metaphor of chemical reactions, which are inherently concurrent and localized. We avoid raising the issues of sequential, parallel, or distributed in the programming language level because we think they belong to the issues of a program execution on different computation resources. From the point of view of a T-Cham program, only reactions exist. The realization of these reactions leads to a sequential, parallel, or distributed execution of the program. In contrast to other approaches, T-Cham has the following features:

1. T-Cham uses tuple space to coordinate a number of transactions. Programmers should consider parallel structures first and then sequential tasks instead of adding parallel facilities to a sequential program;

2. Hierarchical transaction and tuple space structures provide dynamic and abstract views to transactions and their tuple spaces, and also the means for program refinement; besides, it is a high-level portable language: a programmer need not know the structure of the underlying machine: *parallel*, *sequential*, or *networked*;

3. Multi-lingual transactions make T-Cham multi-paradigm. A programmer can take the advantages of different program languages without worrying about their interferences as they are integrated orthogonally by the tuple spaces;

4. Transaction granularity can be easily adjusted by changing the operators contained in the 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;

5. Formal temporal logic semantic enables the correctness proofs for T-Cham programs. The proofs are separated from programming. A programmer need not to touch this notoriously hard aspect if he/she does not want to do;

6. Using the transaction as the fundamental computational unit introduces fault-tolerance and recovery to T-Cham programs naturally.

Our implementation of T-Cham is somewhat preliminary: it mainly depends on Linda's tuple space, even the tuple mappings and hierarchical tuple spaces, which are implemented by simulation (rewriting the hierarchical tuple space structure to a flat one), and the Tester will become a bottleneck when a program scales up. However, our results obtained so far have already demonstrated the effectiveness of T-Cham and its great potentiality. Our future work includes two main streams: implementation and proof system development.

A new execution model will be carried out directly on an MIMD computer, e.g., AP1000, where the Tester will be decomposed, so that we have more opportunities to optimize the data and task distribution and also a good scalability. Dynamic load balancing is also needed if the depth of non-leaf transaction calls can not be predicted statically. A more sophisticated mask mechanism will be introduced to provide T-Cham a natural way of handling irregular data structures, such as multi-dense grid.

Finally, we expect that T-Cham can lead us to a new 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 intuitive presentation, without heavy penalties on execution efficiency.

Acknowledgements

Wanli Ma thanks the Australian Government for an Overseas Postgraduate Research Scholarship (OPRS) and the Australian National University for a PhD scholarship. The authors thank the ANU–Fujitsu CAP Research Project for the use of the AP1000 multicomputer system.

References

[1] D. Agrawal and A. El Abbadi. Transaction management in database systems. In Ahmed K. Elmagarmid (editor), *Database Transaction Models: For Advanced Applications*, pages 1–32. Morgan Kaufmann Publishers, San Mateo, California, USA, 1992.

[2] S. Ahuja, N. Carriero and D. Gelernter. Linda and friends. *IEEE Computer*, pages 26–34, August 1986.

[3] G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[4] J. Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems.* Addison-Wesley, 1993.

[5] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Comm. ACM*, Volume 36, Number 1, pages 98–111, January 1993.

[6] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice-Hall, 1990.

[7] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, Volume 96, pages 217–248, 1992.

[8] N. Carriero and D. Gelernter. Linda in context. *Comm. ACM*, Volume 32, Number 4, pages 444–458, 1989.

[9] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[10] R. Cohen and B. Molinari. Implementation of C-Linda for the AP1000. In *The Proceedings of the Second ANU/Fujitsu CAP Workshop.* 1991.

[11] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[12] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, Volume 19, pages 17–37, 1982.

[13] I. Foster, R. Olson and S. Tuecke. Productive parallel programming: the PCN approach. *Scientific Programming*, Volume 1, Number 1, pages 51–66, 1992.

[14] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[15] D. Gelernter and N. Carriero. Coordination languages and their significance. *Comm. ACM*, Volume 35, pages 96–107, 1992.

[16] C. Hoare. Communicating sequential processes. *Comm. ACM*, Volume 21, Number 8, pages 666–677, August 1978.

[17] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, Volume 21, Number 4, pages 373–386, April 1995.

[18] R. Jagannathan and A. A. Faustini. The GLU programming language. Technical Report SRI–CSL–90–11, SRI International, Menlo Park, CA, USA, 1990.

[19] B. Liskov. Distributed programming in Argus. *Comm. the ACM*, Volume 31, Number 3, pages 300–312, March 1988.

[20] Inmos Ltd. *Occam 2 Reference Manual.* Prentice-Hall, 1988.

[21] 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.

[22] W. Ma, V. K. Murthy and E. V. Krishnamurthy. Multran — A coordination programming language using multiset and transactions. In S. K. Aityan, L. T. Hathaway et al. (editors), *Proceedings of the First International Conference on Neural, Parallel, and Scientific Computations*, pages 301–304. Dynamic Publishers, Inc., Atlanta, Georgia, USA, 1995.

[23] W. Ma and M. Orgun. Verifying Multran programs with temporal logic. In M. Orgun and E. Ashcroft (editors), *The Proceedings of the Eighth International Symposium on Languages for Intensional Programming*, pages 120–134. World-Scientific (in process), 1995.

[24] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[25] W. Reisig. *Petri Nets.* Springer-Verlag, 1985.

[26] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, Volume 2, Number 4, pages 315–339, December 1990.