Programming with Transactions and Chemical Abstract Machine

Wanli Ma — Comp. Sci. Lab.
Christopher W. Johnson — Dept. of Comp. Sci.
Richard P. Brent — Comp. Sci. Lab.

The Australian National University
Canberra, ACT 0200, Australia

Abstract

The coordination style programming language T-Cham *extends chemical abstract machine (Cham) with transactions. The Cham is an interactive computational model based on chemical reaction metaphor, where a computation proceeds as a succession of chemical reactions. A transaction is a piece of sequentially executed codes and could be written in any language, such as C, Pascal, or Fortran etc., as long as it satisfies its pre-condition and post-condition. Every transaction begins its execution whenever its execution condition is satisfied. A T-Cham program can be executed in a parallel, distributed, or sequential manner based on the available computer resources.*

1 Introduction

A plethora of concurrent (including parallel and distributed) programming languages has been proposed, yet concurrent programming is still far more difficult than sequential one. The differences between them are not lying in the single thread versus the multi-thread with communications, but a functional program versus a reactive one [4]. A functional program is the one which maps an input into an output, while a reactive one highlights the interactions among the components of the program. Accordingly, a concurrent programming language should not be judged only by its abilities of thread control and communications, but also the abilities to express the interactions. In other words, a concurrent programming language should not be just an extension to an existed sequential programming language with some thread control and communication facilities, but a new one based on an interactive computation model.

T-Cham (Transactions on Chemical Abstract Machine) is a coordination style programming language obtained by extending *chemical abstract machine* (Cham) [3] with transactions and hierarchical tuple spaces. It is a successor of our earlier programming language Multran [6]. A Cham is an abstract model for concurrent computations. It is based on the chemical reaction metaphor, which was first put out in GAMMA model [2]. A T-Cham program is composed of concurrent *transactions* and hierarchical *tuple spaces*, where the actions take place, and can be executed in a parallel, distributed, or sequential manner based on the available computer resources. Its execution mode is transparent to programmers. Transactions (more precisely, *leaf transactions*) are the sequentially executed tasks with the properties of ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*) [1]. 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 computational model, i.e., Cham, resembles a succession of chemical reactions in which the elements of a tuple space are consumed and generated.

2 The T-Cham Programming Language

The execution of a T-Cham program starts from a special transaction called `root`—the *main transaction* of the program. The transactions referred to by the reaction rules in a transaction are called *sub-transactions* of it. 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 non-leaf transaction (recall that a program is a collection of transactions with a special one named `root`) consists of a name and a body:

```
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.

   ```
   tuples
   tuple {
     int A[100]; int gridsieved;
     } num;
     boolean 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 respectively. `i` is called an *index variable*.

3. **Reactionrules** section consists of a number of reaction rules, beginning with the key word `reactionrules`. The rules operate on the tuple space of a transaction and coordinate its computational actions—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` 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 (with or without changes), for example, "`x,x leadsto x,y`." To distinguish the different appearances of tuples, in the `when` condition and the body of a sub-transaction, the "`$`" operator is used, e.g., "`when (x$1==x$2-10)`." Note that the numbers here never mean the order of the tuples.

   A pair of curly-braces on a tuple name, say `{x}`, means all tuples of this type together, i.e., selecting them all, and a pair of |'s, |`x`|, means the population of this kind of tuples currently in the tuple space. 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 could or could not be generated by a transaction will be preceded by a "`?`" operator, e.g., "`!x,y leadsto ?z`."

4. **Termination** section gives conditions such that whenever any of them is satisfied, the corresponding final action is committed and the transaction then terminates:

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

```
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** section specifies 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.

A leaf transaction looks like this:

> `transaction` *my_name*
> `#language` *my_language*
> `#tuple` *tuple_desp*
>   *my_code*
> `endtrans`,

where *my_language*, known as a *guest language*, is the programming language used to code this transaction, *tuple_desp* provides the type information of a tuple to the transaction, and *my_code* is a programming unit written in "*my_language*". There could be no or many "`#tuple`" lines. The tuples described in the line(s) are resources passed to the transaction before its execution and consumed by it after the execution.

The execution of a T-Cham transaction proceeds as follows: before a termination condition is satisfied, all of its 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 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, 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. The tuple space to a T-Cham transaction is the run time environment to a function or a procedure in an imperative programming language. From the point of view of a transaction, any of its sub-transactions is an "operator" and is executed in exactly "one step."

3 Conclusions

The T-Cham programming language is introduced. For more details about the language and the related examples, we refer reads to our paper [5].

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] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Comm. ACM*, 36(1):98–111, January 1993.

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

[4] L. Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W. P. de Rover, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pages 347–374. Springer-Verlag, 1993.

[5] W. Ma, C. W. Johnson, and R. P. Brent. Concurrent programming in T-Cham. In Kotagiri Ramamohanarao, editor, *The Proceedings of the 19th Australasian Computer Science Conference (ACSC'96)*, pages 291–300. 1996.

[6] 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.