関数＋共有データ＝関数型並行プログラム

Functions + Shared Data = Functional Concurrent Programs

片山卓也、宮地利雄
（東京工業大学）

Takuya Katayama, Toshio Miyachi
(Tokyo Institute of Technology)

［概要］

本論文は、並列システムの仕様記述のための関数起動モデルについ述べたものである。関数型あるいは作用型プログラミングでは副作用が許されず、全てのデータは値によってのみアクセスされるのでreferential transparncyが保たれ、プログラムは見易くかつ理解しやすいものになる。しかしながら、この利点は、同時に、データベースの更新、並行処理あるいは実時間処理などのような履歴をもったシステムの処理や記述を不可能にしており、これが関数型プログラミングが現実のデータ処理に使用されない主な理由であると考えられる。本論文は関数型プログラミングと調和したかたち形で『時間』をとり入れ、それによって並行処理の記述を明解に行う方法について述べたものである。簡単にいえば、本方法はペトリネットと関数を結合したものであり、システムは関数とそれらによって共有され、それらの入・出力となる共有データから構成されるネットワークとして表現される。

関数はその入力データが全て用意されると起動され、その計算が終了するとその計算結果によって出力データを更新する。　勿論、起動可能な関数（群）は並行的に動作する。　このモデルによって、共有データ上で動作する並列システムを直観的かつ理解しやすい形に記述することができるが、履歴の全くない関数と履歴そのものである共有データとに分離して記述を行えるのが本方法の最大の特徴である。

以下では、例題、形式化についてまず述べ、次に実現法、検証法について述べる。

1. Introduction

Function activation model for concurrent system specification is stated. It is well recognized that functional or applicative programming offers a clear and comprehensive approach in software specification since every data is treated as value and no side effect is permitted [1]. This merit, however, is cancelled in the situation where databases have to be accessed or where concurrent or real time problems have to be dealt with. This is the very reason that functional or applicative approaches are not actually used in the practical world of data processing.

This paper is intended to propose a concurrent system specification method in which these deficiencies of applicative approach are cured by introducing time system in harmony with it. In short, our approach consist in the combination of functions and Petri-net [5]. Systems are described as networks of functions and shared data interconnected by

edges through which these data are transfered to and from functions.

Computation of the functions are activated when all their input data become available and replaces with their results the values of shared data connected to their output ports. Of course, functions in the network can be activated parallelly when possible. This model provides intuitive means for describing systems operating on shared resources in applicative way.

In the following we start our discussion with a simple example and go to a formalism. Finally, implementation and verification techniques are touched upon.

2. Simple Examples

(1) Producer-Consumer Problem

Let's consider 'arithmetic' producer-consumer problem. This system consists of two functions add1, subtract1 and a shared variable n which takes an integer value between 0 and N.

The function add1 increments the value of n by 1 when requested provided n<N, and subtract1 decrements n by 1 if n>0. This system is modelled as in Fig. 1.

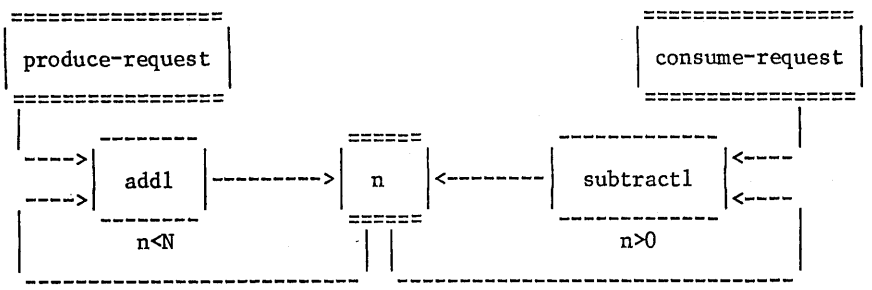


Fig. 1

The shared data n is protected from illegal concurrent updating. That is, when 'produce-request' comes the function add1 seizes the data n if it is not seized by subtract1. Subtract1 cannot obtain the data n until add1 computes the value of n+1 and returns it to the data box. Note that add1 and subtract1 cannot seize the data n unless n<N and n>0 respectively.

The function add1 has two arguments : one to input n and one to receive the signal 'produce-request'.

add1[n; produce-request]=n+1

'produce-request' also represents (shared) data given from outside the system. Only the fact that it is received has importance about it and its value is irrelevant. We call these data <u>signal</u>.

This very simple example explains the charactistics of our approach. It consists of

clearly seperated two subsystems :

(1) value subsystem

(2) time subsystem

The value subsystem is composed of functions and values of shared data and it describes how these values are changed when functions are activated. We do not pay attention to how the functions are defined and perform their computations.

The time subsystem describes when functions can start their computation and when shared data become available and unavailable. This subsystem is considered an augumented Petri-net. The function box corresponds to a transition in Petri-net and shared data to place. What is augumented is : (1) values flow instead of stones, (2) functions are activated instead of transitions, and (3) predicates about shared data can be attached to its function boxes. Fig.2 is the Petri-net for our time subsystem. It is to be noted that this net is isomorphic to our original net in graph-theoretical sense. It comes from the functionality of the value sybsystem, i,e., the updating functions add1 and subtract1 are pure functions, which first get the shared data and then return computation results.

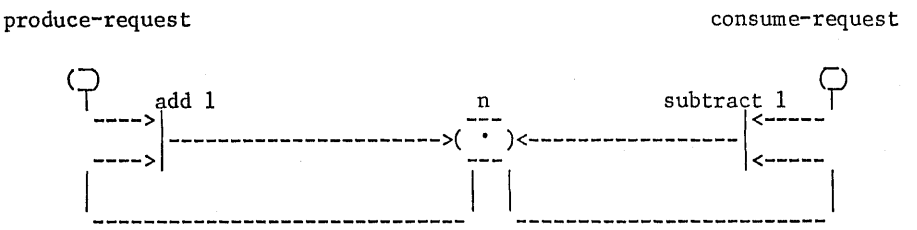


Fig. 2

(2) Readers-Writers Problem

This problem is modelled in the diagram of Fig.3.

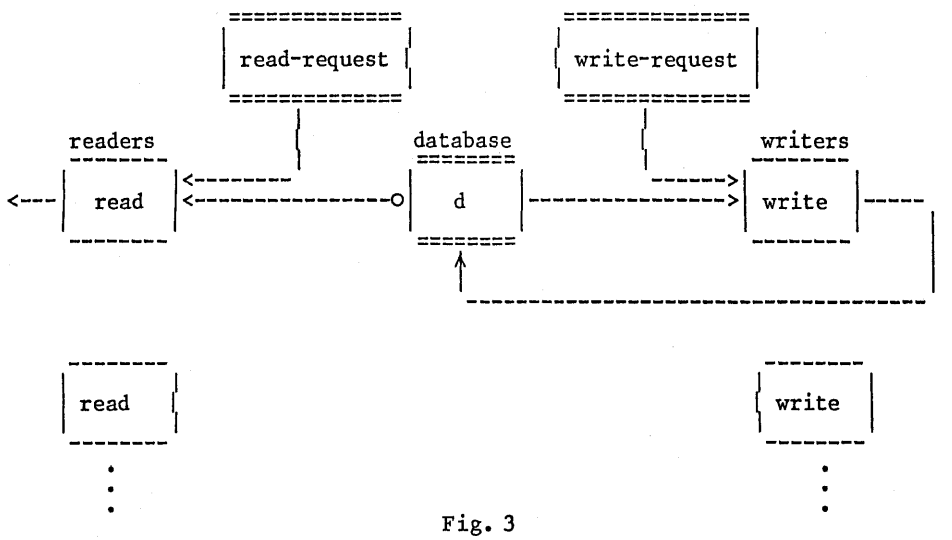


Fig. 3

When a writer attempts to update the database d, it first seizes d and leaves the data box empty so that no other writers or readers can access it. In contrast, when a reader reads, a copy of d is transmitted to the function 'read' and other readers and writers can access d. This fact is represented in the diagram by a circle at the root of an edge from 'd' to 'read', showing the edge is a copy-edge.

Locking of the database is accomplished by assumptions about data boxes and edges :

(1) at most one access to a data box is permitted at any time, and

(2) accessed data is transferred to input port of a function immediately after the instant the access become enable. The time required actually to read or write the data is specified as computation time of the functions 'read' and 'write', and this is the only source of time delay in our network.

3. Formalism

3.1 Syntax

A function activation model (abbreviated FAM hereafter) G is a triple

$$G=(F,D,E,s_0,S_f)$$

, where (1) F is a set of function boxes, (2) D a set of data boxes , (3) E a set of edges connecting them, (4) s_0 an initial state and (5) S_f a set of final states.

[1] To each function box f∈F are associated

(1) **input-ports**(f)

(2) **output-ports**(f)

(3) **function**(f)

(4) **predicate**(f).

input-ports(f) is a set of input-ports of f, **output-ports**(f) a set of output-ports of f, and **function**(f) is a function which transforms values reached input ports into output values. **predicate**(f) specifies a condition on which f may be activated prodived its input data are available. This condition is specified in terms of values of data boxes in G.

Input and output ports are either for value or for signal. That is,

input-ports(f) = **input-value-ports**(f) ∪ **input-signal-ports**(f)

output-ports(f) = **output-value-ports**(f) ∪ **output-signal-ports**(f)

Let **dom**(P) be a value domain for a value port P, that is,a set of possible data values appears at P, then the **function**(f) is a mapping

$$\textbf{function}(f) : \textbf{dom}(P_{i1})\times \ldots \times \textbf{dom}(P_{in}) \rightarrow \textbf{dom}(P_{o1})\times \ldots \times \textbf{dom}(P_{om}),$$

where

input-value-ports(f) = $\{P_{i1}, \ldots, P_{in}\}$

output-value-ports(f) = $\{P_{o1}, \ldots, P_{on}\}$.

[2] To each data box d∈D are associated

(1) **dom**(d)

(2) **value**(d).

dom(d) is the set of data stored in d. **value**(d) is a data value stored in the box d. We introduce a special symbol ⊥ ∈**dom**(d) and write **value**(d)=⊥ to show that the data box d is empty.

[3] E is a set of directed labeled edges connecting function boxes and data boxes.

$$E \subset D \times \textbf{input-ports} \cup \textbf{output-ports}$$

where,

$$\textbf{input-ports} = \cup_{f \in F}\ \textbf{input-ports}(f)$$

$$\textbf{output-ports} = \cup_{f \in F}\ \textbf{output-ports}(f)$$

When an edge connects a port P of a function box with a data box d it is required that **dom**(P)=**dom**(d).

For each edge e=(d,P)∈D×**input-ports** is associaisted **copy**(e) which shows whether a copy of **value**(d) is transferred or not. That is moved to the input port P of a function leaving d unchanged, whereas the data in d is moved and d becomes empty when **copy** is false.

[4] A set DSTATE of states of data boxes is defined by

$$\text{DSTATE} = \textbf{dom}(d_1) \times \textbf{dom}(d_2) \times \ldots \times \textbf{dom}(d_t)$$

where $D = \{d_1, d_2, \ldots, d_t\}$. Then we have

$$s_0 \in \text{DSTATE} \quad \text{and} \quad S_f \subset \text{DSTATE}.$$

3.2 Semantics

Operational semantic of FAM is given here. It is a state transition model.

State

State space state(G) of FAM $G = (F, D, E's_0', S_f)$ is a subset of the direct product of state spaces of function and data boxes involved. That is,

$$\text{state}(G) \subset \text{state}(f_1) \times \ldots \times \text{state}(f_r) \times \text{state}(d_1) \times \ldots \times \text{state}(d_t)$$

, where $F=\{f_1, \ldots, f_r\}$ and $D=\{d_1, \ldots, d_t\}$.

State space of f∈F and d∈D are given by

state(f) = {activated, quiescent}

state(d) = **dom**(d).

State ⊥ of d means that this data is seized by some function and currently the data box d contains no value.

State Transition

State transition occurs when (1) a function f ∈ F is activated and (2) computation of f is completed.

[Activation Transition] We assume that a function is activated immediately after the

condition for the activation is satisfied. It is that all its input data are available and its activation condition is true. More specifically, it is stated in the following way. Let d[P] denote a data box connected to a port P of the function box. Then the activation condition is,

(1) **value**(d[P]) $\neq \perp$ for all P$\in$ **input-ports**(f)

(2) **predicate**(f) = **true** for the current values in data boxes.

State change caused by the activation of f is described by

value(d[P]) = $\perp$

for all P$\in$ **input-ports**(f) such that **copy**(e) = false, e=(d[P],P)

[Completion Transition] When the computation of f is completed it destributes the values computed to data boxes connected to output ports of f. They overwrites on the values stored in the boxes. When **input-ports**(f) = (P_{i1}, ..., P_{in}) and **output-ports**(f) = (P_{o1}, ..., P_{om}), this state change is stated by

$$(\textbf{value}(d[P_{o1}]), \ldots, \textbf{value}(d[P_{om}])) = \textbf{function}(f)[\textbf{value}'(d[P_{i1}]), \ldots, \textbf{value}'(d[P_{in}])].$$

where **value**'(d[P_{ij}])'s are **values** of d[P_{ij}]'s at the time the function f is acticted.

We assume no time is required for transitions. Time delay is introduced only by computation of functions.

Behavior

A FAM G = (F,D,E,s_o,S_f) starts its computation with s_o as their initial values for its data boxes D and repeats transitions as in the manner described above. The behaviour is necessarily parallel and nondeterministic.

When a state is reached from which no transition is possible, we say that G has terminated successfully if the state is in S_f, otherwise deadlocked. G is deadlock free if any computation started from s_0 leads to successful termination. G is determinate if G is deadlock free and s_0 leads to a unique final state.

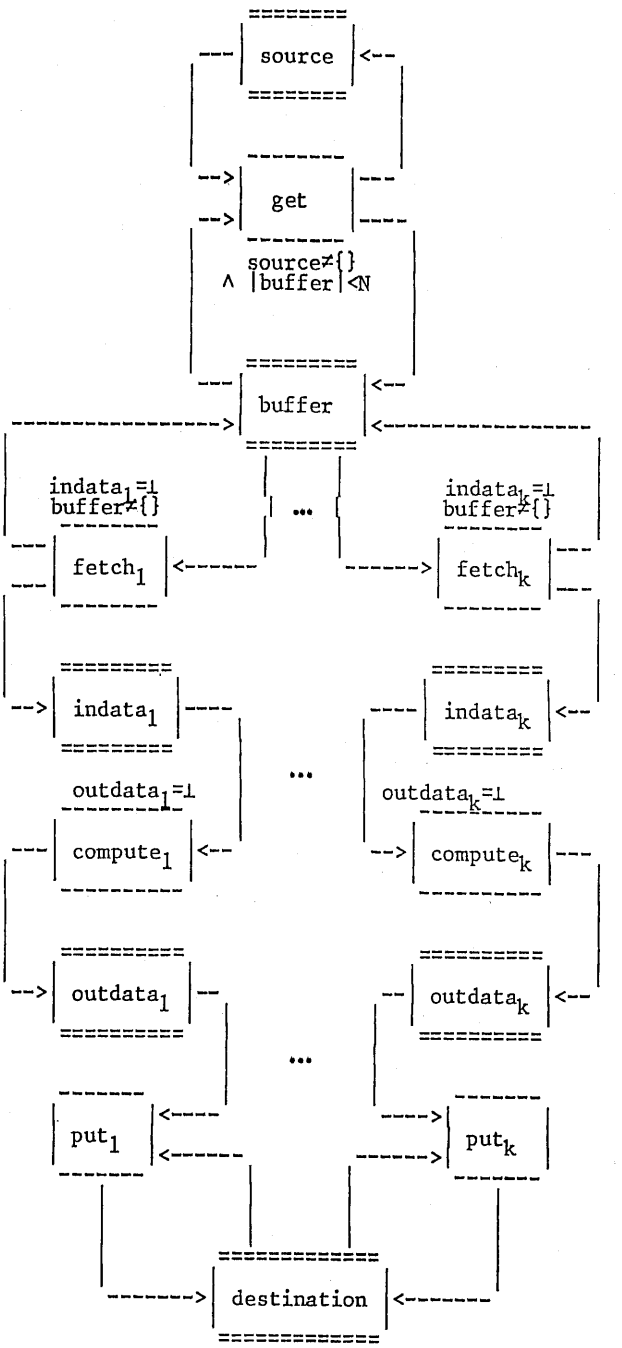
4. Buffered Input-Output System

As yet another example, we give a description of a buffered input-output system in Fig.4. A set of data is first given in the data box 'source'. A function 'get' reads a data and put it into a buffer of size N. This buffer is represented by a set of size N. Note that the function 'get' operates on 'source' and 'buffer' and updates them in such a way that a data in 'source' is moved to 'buffer'.

get : source × buffer → source × buffer

The data is, in turn, fetched by a group of functions 'compute's which produce output data. This data is finally stored into a set 'destination' of output data

by 'put'.


source
get
source≠{}
∧ |buffer|<N
buffer
indata1=⊥
buffer≠{}
fetch1
indatak=⊥
buffer≠{}
fetchk
indata1
indatak
outdata1=⊥
compute1
outdatak=⊥
computek
outdata1
outdatak
put1
putk
destination

In this example, the function 'get' can be activated only when 'source' is not empty and 'buffer' is not full and this fact is represented by a predicate

source≠{} ∧ |buffer|<N

attached to 'get'. In the similar way 'fetch' may be activated when 'buffer' is not empty and 'indata' is empty.

Mutual exclusion about 'buffer' is achieved by that when a 'get' obtains a data in 'buffer' it leaves ⊥ there and any other 'get' cannot be activated until the first 'get' returns an updated result. It is similar with 'destination'.

'fetch' --> 'indata' --> 'compute' --> 'outdata' --> 'put'

is an example of pipelining process, and this is accomplished by that functions cannot be activated while their input data boxes are empty and their activation condition is false.

5.Implementation

In principle, it is not difficult to translte a FAM network into a set of concurrent processes operating on shared data. The first step of the translation is : (1) to assign a boolean variable empty[d] and a binary semaphore sem[d] to each data box d, and (2) to associate process PROCESS[f] with each function box f.

The process PROCESS[f] first waits for the simultaneous availability of semaphores

S={sem [d] | d ∈ IN(f) ∪ PRED(f)},

where (1)IN(f) and (2)PRED(f) are sets of data boxes which are (1) connected to input ports of f, and (2) referred in the predicate **predicate**(f) respectively. We also denote by OUT(f) a set of data boxes connected to the output ports of f. It then tests whether the condition B for activation of F holds.

B = $\wedge_{d\in IN(f)}$ **not** empty[d] ∧ **predicate** (f).

When B is true, it executes the computation specified by f and stores the computed values to output data boxes. Finally, PROCESS[f] releases semaphores S. After all, PROCSS[f] looks like :

```
process PROCESS[f]
    loop
      P(sem[d] | d∈IN(f)∪PRED(f)) ;
      if B then
          V(sem[d] | d∈PRED(f)-IN(f) ;
          R ← function(f)[IN(f)] ;
          P(sem[d] | d ∈ OUT(f)-IN(f)) ;
          OUT(f) ← R ;
          forall d ∈IN(f)-OUT(f)
              do empty[d] = not copy((d,f)) ;
          forall d∈OUT(f)
```

```
                do empty[d] = false ;
            V(sem[d] | d∈IN(f)∪OUT(f)) ;
        else V(sem[d] | d∈IN(f)∪PRED(f)) ;
        fi ;
    endloop
end
```

The entire FAM network is, then, translated into the following program.

```
program
  var d1, d2, ...
  var empty1, empty2, ... : boolean
  semaphore sem1, sim2, ...
  process PROCESS[f1]
                .
                .
  process PROCESS[f2]
                .
                .
  parbegin
            PROCESS[f1] ;
            PROCESS[f2] ;
                .
                .
  parend
end
```

The most difficult problem about the above implementation scheme might be how to execute the body of PROCESS[f]

OUT[f] ← **function**(f)[IN(f)]

efficiently. As **function**(f) is a pure function which operates on values in data boxes in IN(f) and returns values computed to OUT(f), it causes problems when big data, such as files and databases, are stored in data boxes. Although this is one of the largest problems about functional programming, we have, in most cases, means to transform this costly 'by-value' operation into 'by-update' operations where only a portion of entire data is efficiently handled [4]. The technique of lazy evaluation also offers a mechanism to efficiently execute the above computation [3].

6. Verification

Semantics of a FAM network can be easily translated into a set of formulas in a temporal logic and proof techniques for the logic can be applied to prove properties of FAM networks. Of course, this applicability of temporal logic is not specific to FAM

network and other models for the description of concurrent systems also enjoy this verification strategy [6].

FAM network has another advantage with respect to its verification. That is, it admits to verify its time-independent properties without regard to its time subsystems, therefore, without resorting to temporal logic. Suppose $Q(\mathbf{d})=Q(d_1, \ldots, d_m)$ is a predicate about values stored in data boxes $D=\{d_1, \ldots, d_m\}$ and Q is true for any sequence of possible function activations. Then, Q can be proved by the function activation induction given below.

$$\frac{Q(\mathbf{d}_0),\ \wedge_{f\in F}[Q(\mathbf{d})\wedge \mathbf{predicate}(f) \supset Q(\mathbf{d}')]}{Q(\mathbf{d})}$$

where $\mathbf{d}_0 = s_0$ is the initial values stored in D and $\mathbf{d}'$ is obtained by replacing d 's in OUT(f) by **function**(f)[**d**] and d's in IN(f)-OUT(f) by $\perp$'s.

$\mathbf{d}' = [\mathbf{d}]_{OUT(f)\leftarrow \mathbf{function}(f)[\mathbf{d}],\ IN(f)-OUT(f)\leftarrow \perp's}$

This merit of FAM network comes directly from that it value subsystem and time subsystem is clearly separated in our formalism and the value subsystem is based on pure functions.

As an example, consider the arithmetic producer-consumer problem given in section 2. The value of n should be kept between 0 and N during execution provided the initial value n_0 satisfies $0\le n_0\le N$. The property to be verified is

$Q(n)$: $0\le n\le N$.

To establish Q(n), the function activation induction principle directs to prove that the following formula is valid.

$$[\ 0\le n\le N \wedge n<N \supset 0\le \mathrm{add1}(n)\le N\ ]$$
$$\wedge\ [\ 0\le n\le N \wedge n>0 \supset 0\le \mathrm{subtract1}(n)\le N\ ].$$

As it obviously holds the property is verified.

7. Concluding Remarks

A new spesification method for concurrent systems is introduced and its implementation and verification are considered. This method is based on mathematical functions and Petri-net and provides means to describe complex concurrent systems in comprehensive and well-structured way. One of its characteristics is that its value subsystem is based on functions and shared data and this will be the most natural way to extend the domain of functional programming to cope with applications involving, say, concurrency and real-time.

Pure functions(without side effects) and Petri-net are unified very harmoniously in

our model. Function must seize data when it wants to operate on them and this expresses the mutual exclusion control on data naturally. Of course, this corresponds to the firing of transitions in Petri-net model.

Computation model which is most closely related to ours is the data flow model [2]. Although both models are founded on Petri-net and functions, their objectives are quite different. The data flow model is intended to carry out computations without introducing inessential ordering on data operations and perform them as parallel as possible while they flow from the input terminals to the output terminals of the network. On the other hand, our FAM approach aimes at the description of concurrent processes operating on shared data and contending to obtain them. This difference on their objectives make the structures of their networks different from each other in the way that, for example, (1) FAM has data boxes to store data values whereas data flow model doesn't and (2) the outdegree of places of the corresponding Petri-net is restricted to one in data flow models whereas it is not in FAM.

Functional approach offers metrits in verification. That is, time-independent properties can be proved by computation induction principle in value domain without resorting to temporal logic. We consider this one of its major advantages of our approach since verification of concurrent systems is, in general, very difficult when their value subsystem and time subsystem are not clearly separated.

References

1. Backus, J. : 1977 ACM Turing Award Lecture : Can Programming Be Liberated from the Von Neumann Style ? A Functional Style and its Algebra of Programs. C.ACM Vol.21 No.8 (Aug,1978)

2. Dennis,J.B. : First Version of a Data Flow Procedure Language.
MIT/LCS/TM-61 (May,1975)

3. Henderson,P. & Morris J.H. : A Lazy Evaluator. Proc. of 3rd ACM Symp. on Principles of Programming Languages (1976)

4. Katayama,T. : Treatment of Big Values in an Applicative Language
HFP--Translation of By-value Access to By-Update Access. Tech. Rep. CS-K8202, Dept. of Comp. Sci., Tokyo Inst. of Tech., (1982)

5. Peterson,J.L. : Petri Net Theory and the Modeling of Systems.
Prentice-Hall (1981)

6. Pnueli,A. : The Temporal Logic of Programs. Proc. of 19th Annual Symposium of Foundations of Computer Science. (Nov., 1977)