# Semantics for an Asynchronous Message Passing System

A.V.S. Rajan, S. Bavan, G. Abeysinghe
School of Computing Science
Middlesex University
The Borroughs, London NW4 4BT UK

*Abstract*-**The main focus of this paper is to define the operational semantics for the message passing strategy called Asynchronous Message Passing System (AMPS) used in the distributed programming language, LIPS (Language for Implementing Parallel/distributed Systems). AMPS is a point-to-point message passing system that does not use any message buffers. It is based on simple architecture and interfaces. In order to adequately provide implementation information for the message passing strategy, we have defined the operational semantics and the codes needed for the abstract machine of LIPS.**

**Keywords:** Operational semantics, Asynchronous Message Passing, point-to-point message passing, abstract machine.

## I. Introduction

Operational semantics describes the executional behaviour of a programming language and gives a computational model for the programmers to refer to. This paper presents the operational semantics that models the asynchronous message passing behaviour of LIPS, a Language for Implementing Parallel/distributed Systems [1].

LIPS has the following properties:

- Communication by assignment
- Separation of communication from computation
- A data flow nature coupled with asynchronous message passing.
- Portability
- A program is made up of processing nodes (processes) which are linked by unidirectional data channels that carry messages between cooperating nodes.

The detailed explanation of LIPS can be found in [1].

A LIPS program consists of a network of nodes described by a network definition and node definitions. A network definition describes the topology of the program by naming each node (representing a process) and its relationships (in terms of input and output data) to other nodes in the system. A node consists of one or more guarded processes which perform computations using the data that arrive as input and produces output that are sent to other relevant nodes. We formally specify the message passing between the various nodes in LIPS using the Specification of Asynchronous Communication System (SACS) [2], a synchronous variant of Synchronous Calculus of Communicating System (SCCS) [3] (network definition in a LIPS program). The design techniques of SACS allow the programmer to develop programs that are virtually free of livelock and deadlock conditions.

The main focus of this paper is to define the operational semantics for the message passing strategy called Asynchronous Message Passing System (AMPS) [4] used in LIPS. AMPS is a point-to-point message passing system that does not use any message buffers and is based on a simple architecture and interfaces.

The operational semantics of LIPS is described using an evaluation relation **(P, s)** $\Downarrow$ **(P', s'),** where a program expression, P in state s is evaluated to P' with a change of state to s'. The particular style of operational semantics and the abstract machine we have adopted was inspired by Crole [5]. This style uses an evaluation relation to describe the operational semantics by showing how an expression evaluates to a result to yield a change of state and a compiled Code Stack State (CSS) machine for an imperative toy language called IMP.

We have defined the abstract machine for LIPS called LIPS Abstract Machine (LAM) that executes instructions using re-write rules. A re-write rule breaks the execution step into number of sub-steps and transform the given expression P into a final value V ( $P \Downarrow^{e} V$ ) as follows:

$$P_0 \mapsto P_1 \mapsto P_2 \mapsto \ldots \mapsto V$$

In order to define the AMPS using the LAM, we need a few preliminary definitions. The LAM consists of rules for transforming the LAM configurations. Each configuration in the LAM is a triplet, **(C,S,s)** where

- C is the Code to be executed
- S is a Stack which can contain a list of integers, real numbers, Booleans, characters, or strings
- s is a state which is the same as that defined in the LIPS operational semantics.

The remainder of this paper is structured as follows: Section II describes AMPS, the architecture of the virtual message passing system which has been developed to pass messages asynchronously without message buffers. Section III describes the operational semantics and the codes needed for the AMPS. Section IV concludes the discussion.

## II. The Amps Of Lips

The Asynchronous Message Passing System (AMPS) of LIPS has been developed in order to achieve asynchronous message passing effectively across different platforms without any message buffers. AMPS consists of a very simple Data



*T. Sobh (ed.), Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, 83–88.

Structure (DS) and a Driver Matrix (DM). The LIPS compiler automatically generates the DS, DM and required AMPS interface codes. With network topology and the guarded processes, it is easy to identify the variables participating in the message passing. This section first describes the DS and the DM and then goes on to describe how AMPS transfers data to and from the nodes using simple interfaces.

### *A. The Data Structure (DS) of AMPS*

The Data Structure is a doubly linked list where all the nodes in the network including the host node are linked to the other nodes. Each node has the following six components:

1. A node number (**NodeNum** – *an integer*) – a unique number is assigned to each node.
2. Name of the function it is executing (**name** – *a symbolic name*)
3. A pointer to the next node in the system
4. Further two pointers:
   i. A pointer to a list of input channel variables associated with that node.
   ii. A pointer to a list of output channel variables associated with that node.
5. Input channel variable – each input channel variable consists of a data field giving the channel number (**vnum** – *an integer*) and two pointers, one pointing to the next channel variable in the list and the other points to a record with the following fields
   i. Channel name (**var1**-*symbolic name*). This is used for debugging purposes.
   ii. Currency of the data present – old data (status = 0) or new data (status = 1). Only data with status = 1 is passed on to a node for processing.
   iii. Value of the data. (**value** – *actual value of specified type*)
6. Output channel variable - each output channel variable consists of a data field giving the channel number (**vnum** – *an integer*) and two pointers, one pointing to the next channel variable in the list and the other points to a record with the following fields:
   i. Channel name (**var1**-*symbolic name*)
   ii. The number of nodes that are to receive the data (**counter** – *0..n*), which will be decremented as a copy of the data is transferred to a destination node. New data is only accepted (written) when this counter is 0.
   iii. Value of the data. (**value** – *actual value of specified type*).

### *B. The Driver Matrix (DM) of AMPS*

The DM facilitates the distribution of messages and contains the details of the channel variables in the network which are as follows:

i. The channel number, vnum (Integer),
ii. The node number (Integer) from where the channel variable originates (Source node),
iii. The data type of each channel variable (Integer value – 0 to 8) and
iv. The nodes where they are sent as input, the destination nodes (either 1 or 0 in the appropriate column). A '1' in a column indicates that the corresponding destination receives a copy of the input and a '0' otherwise.

All the values in the matrix are integers. The integer values given to the source nodes and destination nodes are same as the node numbers used in the DS.

### *C. The Operation of AMPS*

When a node outputs a message, a message packet in the following format is generated.

| `Src_Node_Number` | `Vnum` | `Type` | `data` |
|---|---|---|---|

**Message packet – 1** sent from a node

Once a piece of data is ready, the process in the source node makes the following call to AMPS:

```
Is_ok_to_send(Src_node_number, vnum)
```

When this call is received, the AMPS checks the DS to see if the output channel of the node has its copy value set to zero. If it is set to zero, it returns a value 1 else a 0. If the value received is 0, the sending process waits in a loop until a 1 is received. When a value 1 is received, the sender node sends the message in a packet using the following call.

```
Send(src_node_number, vnum, type, data)
```

On the receipt of this packet, the AMPS checks the DS to see whether the **`vnum`** and the **`type`** are correct, stores the data in the appropriate field. The copy counter is set to the number of nodes that are to receive the data by consulting the DM. The **`Send`** function returns a 1 to indicate success else a 0 to indicate a failure.

After storing the data, the AMPS consults the DM, distributes the data to other DS nodes and decrements the copy counter accordingly. Here the data is written to the input channel variable of a receiving DS node, provided the status counter of that input channel variable is 0 (that is, the channel is free to receive new data). Once the data is received the status is set to 1. If any of the DS destination nodes were unable to receive the new data, AMPS periodically checks whether they are free to accept the data.

When a guard in a node requires an input, it makes the following call to the AMPS:

```
Is_input_available(Rec_node_number, vnum)
```

The AMPS checks the appropriate DS node and the channel variable number, **`vnum`**. If the status is 1, the function returns a `'1'` to tell the caller that data is available, else it returns a 0. In the event of a `'0'` return, the DM is consulted to find the source. If the data is available, the system transfers a copy to the input channel of the waiting DS node and waits for the corresponding process to make another call of **`Is_input_available`** function.

If the receiving call gets a `'1'` in return, then the node makes a request to AMPS to send the data. The AMPS extracts the data from the appropriate channel of the DS and