# PROTOCOL VALIDATION VIA REACHABILITY ANALYSIS: AN IMPLEMENTATION

by

DANIEL HANG-YAN HUI

B.C.S., Concordia University, 1983

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department Of Computer Science

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1985

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the The University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date: April 1985

Abstract

Reachability analysis is one of the earliest and most common techniques for protocol validation. It is well suited to checking the protocol syntactic properties since they are a direct consequence of the structure of the reachability tree. However, validations of unbounded protocols via reachability analysis always lead to the "state explosion" problem. To overcome this, a new approach in reachability analysis has been proposed by Vuong *et al* [Vuong 82a, 83a]. While not loosing any information on protocol syntactic properties, the reachability tree constructed by the new approach for all non-FIFO and for a particular set of FIFO protocols (called well-ordered protocols) will become finite. This thesis is concerned with the implementation of an integrated package called VALIRA (VALIdation via Reachability Analysis) which bases on both the proposed technique and the conventional technique. Details and implementation of the various approaches used in VALIRA are presented in order to provide an insight to the package. Various features of the package are demonstrated with examples on different types of protocols, such as the FIFO, the non-FIFO, and the priority protocols. The use of VALIRA was found to be practical in general, despite some limitations of the package. Further enhancements on the VALIRA are also suggested.

Table of Contents

## List of Figures

Acknowledgement

I would like to thank Dr. Son T. Vuong, my supervisor, for his guidance and our many long discussions. Special thanks to Dr. Sam T. Chanson for reading and commenting on the final draft and to Dr. Alan K. Mackworth for the financial support of a research assistantship.

Chapter 1

# INTRODUCTION

A protocol is a set of mutually agreed rules governing the interactions between different entities. Communication protocols play an important role in distributed systems and computer networks since they are essential for communications between entities of different conventions.

Realistic protocols are usually complicated and it is hard to validate their total correctness. With the growth in protocol complexity, various protocol analysis techniques have been developed with respect to different types of protocol models [Sunshine 78]. These techniques are related to either protocol validation in the syntactic aspect or protocol verification in the semantic aspect.

In protocol validation, state transition techniques like protocol synthesis and reachability analysis are developed [Zafiropulo 78, 80]. Protocol synthesis can be used to aid developing a protocol in the early stages of design, or to validate a protocol via resynthesizing at a later stage. In an advanced state of development, validation via reachability analysis is more appropriate. Reachability analysis of protocol is done by exhaustively generating all global states of a protocol specification to allow the checking of syntactic design errors. This analysis procedure can be easily automated. However, the non-termination of an unbounded reachability tree due to unbounded channels has been a major problem in the analysis. With the algorithms proposed by Vuong *et al* [Vuong 82a, 83a], all unbounded non-FIFO protocols, and a particular set of unbounded FIFO protocols (called well-ordered protocols, to be defined in Chapter 3), can now be analysed with the reachability tree constructed always to be finite. In this thesis, we shall focus ourselves on the proposed approach and its implementation.

## 1.1 THESIS OBJECTIVE AND CONTRIBUTION

The objective of this thesis is to provide users with an automated protocol validation system which would eventually form part of a larger integrated automated system [1] for protocol processing. VALIRA, a validation tool, has been implemented for this purpose. Using VALIRA, various types of protocols specified in CFSM models with FIFO, non-FIFO, or priority channels can be analysed for potential design errors such as state deadlocks, state ambiguities, unspecified receptions, and non-executable interactions.

## 1.2 THESIS OUTLINE

This thesis describes mainly the background theories and the implementation of VALIRA. Chapter 2 is a brief review of the general background regarding the communicating finite state machine model used in reachability analysis. Chapter 3 describes various algorithms of reachability analysis for protocol validation. Chapter 4 provides the implementation details of VALIRA. Chapter 5 shows some simple but realistic examples in order to illustrate the particular features of the different approaches used in the system. Chapter 6 is the concluding chapter which includes a thesis summary and suggestions for future work.

A user's manual is included in the appendix for those who are interested.

---

[1] Other parts of the system are under research within the department.

Chapter 2

# BACKGROUND STUDIES

Protocol specifications can be formalized by various models. The existing modeling techniques can be classified into the following three main categories [Bochmann 80]:

1. Transition models

   Transition models, such as the communicating finite state machine (CFSM) models [Bochmann 78] and the Petri nets [Merlin 76], have been the most commonly used models. They specify the control aspects of protocols well, but are poor in modeling their semantic aspects. The CFSM model, which is used throughout this thesis, will be described later in more details.

2. Programming language models

   Programming language models, which use high-level programming languages for modeling, are natural representation of protocols. Though they have a full range of applicability, they are not as practical since automated validation is difficult. Examples of this technique are the Gouda's model [Gouda 76] and some high-level programming models [stenning 76].

3. Hybrid models

   Hybrid models are a combination of the transition and the programming language models, so as to obtain their combined advantages. Numerical Petri net [Symon 80a, 80b] is one of the well known examples.

Of the various models, the CFSM model is most applicable to reachability analysis. The remainder of the chapter presents a brief overview of the model and the various types of errors which could be handled.

## 2.1 THE COMMUNICATING FINITE STATE MACHINE (CFSM) MODEL

In reachability analysis, protocol specifications are normally formalized by CFSM models. A CFSM model is a state-event driven type of model. Essentially, it consists of a number of finite state machines (FSM's), with each FSM representing a process in the protocol specified. The FSM's communicate with each other by exchanging messages through single-directional, error-free channels. These channels may belong to one of the following classes:

1. First-in-first-out (FIFO): messages are received in the order that they were sent.
2. Non-FIFO: messages may be received in any arbitrary order.
3. Priority: messages in the channel are ordered by their associated priorities. The first message in the channel is always the next to be received.

In practice, protocols which lie on top of a reliable and order-preserving transport service (provided by the lower level protocols) are considered to have FIFO channels.

As an example, let us look at the Simple Access Authorization Protocol (SAAP) described by Zafiropulo *et al* [Zafiropulo 80]. The protocol consists of a requesting process and an authorizing process (Figure 2.1), each communicating with the other via FIFO channels (not indicated in the figure). In the CFSM model, each message transmission or reception is represented by a transition arc labled with a '-' (transmission) or a '+' (reception) sign followed by the message type. Processes in the model synchronize with each other by sending or receiving a message, which brings the process through the transition arc from one state to another.

Semantically, the SAAP specifies that whenever the requesting process *requests* for connection, the authorizing process can either *refuse* it, bringing both back to the idle states; or *grant* it, advancing both to the connect states. When the requesting process wants to discontinue the connection, it sends a *release* and hence returns both to idle.

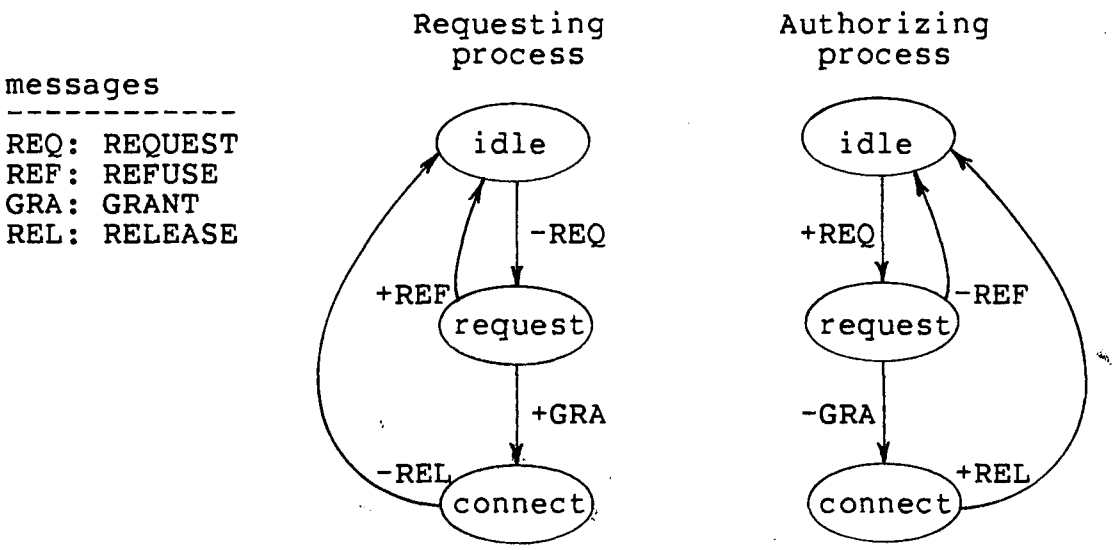



Figure 2.1. A simple access authorization protocol (SAAP).

In the later chapters, we shall see more examples of the CFSM model.

## 2.2 TYPES OF DESIGN ERRORS

With the CFSM model described in Section 2.1, a number of potential design errors can be handled. The four types of design errors, described by Zafiropulo *et al* [Zafiropulo 80], are informally defined as follows:

1. *State ambiguities*

   In general, we say that an N-process [2] **stable-state** tuple $<s_1,....,s_N>$ exists when the states $s_1,....,s_N$ of processes $P_1,....,P_N$, respectively, are reached with all channels empty. In such a case, all the states **coexist stably** with each other in the tuple. A **state ambiguity** exists when a process state appears in more than one stable-state tuple. Thus, when given such a process state, we can no longer determine the exact coexisting stable-state of the other processes.

---

[2] N is refered to as the number of processes throughout this thesis.

State ambiguities do not necessarily represent errors, but their semantic intents must be examined with caution.

2. *State deadlocks*

A **state deadlock** occurs at stably coexisting states where no transmissions are possible. Thus, the processes cannot make any move but to remain indefinitely in their existing states.

State deadlocks usually represent errors, unless the protocol is designed to terminate in that manner.

3. *Unspecified receptions*

An **unspecified reception** occurs when there is possible reception of a message but the corresponding reception arc is not specified.

Unspecified receptions are harmful since the subsequent interactions are unpredictable.

4. *Nonexecutable interactions*

A **nonexecutable interaction** is a reception specified but not possible to occur under normal operating conditions. A protocol is **well-formed** if and only if it contains no unspecified reception or any nonexecutable interaction.

Nonexecutable interactions must be handled with great care since they might indicate the existence of design errors.

As an example to illustrate the various types of design errors, an erroneous protocol in a CFSM model is shown in Figure 2.2. The protocol is a modified version of the SAAP, which also provides the following features:

- Instead of refusing the *request* from the requesting process, the authorizing process now issues a *wait* and enters a wait state until another request is received. The requesting process should return to idle when it receives a *wait*.
- The authorizing process returns to idle upon receiving a *release* while it is in the wait state.

- The authorizing process can *request* for immediate connection when it is in the idle state.

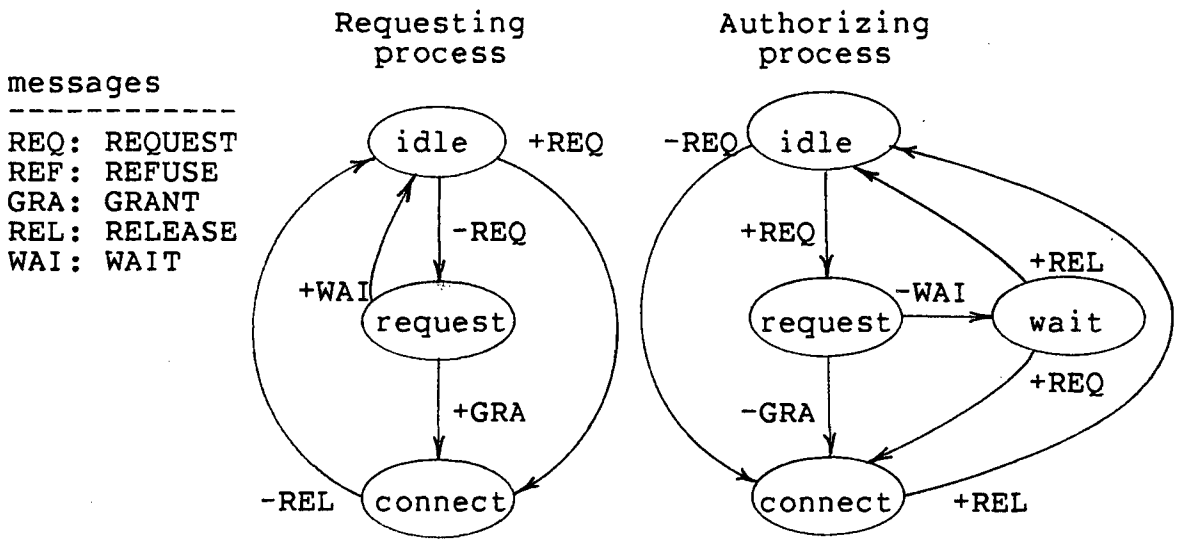



Figure 2.2. A modified SAAP.

It is obvious that the modified SAAP consists of all the four types of design errors, as indicated below:

1. State ambiguity:

   The idle state of the requesting process coexists stably with both the idle state and the wait state of the authorizing process. The latter happens when both processes start from the idle states, then the requesting process issues a *request* and the authorizing process responds with a *wait*.

2. State deadlock:

   Furthermore, if now the requesting process goes to the request state by issuing another *request*, the authorizing process in the wait state will receive the *request* and proceed to the connect state. This results in a deadlock situation in

which both processes are waiting for messages from empty channels.

3. Unspecified reception:

   If both processes issue a *request* simultaneously in their idle state, they both reach a state where the reception of the *request* message from the other process is not specified.

4. Nonexecutable interaction:

   The reception of the *release* message in the wait state of the authorizing process is nonexecutable. This can be easily verfied using reachability analysis, as all executable interactions are included in the reachability tree. Interactions not appearing in the reachability tree are simply nonexecutable interactions.

For most non-trivial protocols, the design errors discussed above are likely to exist, yet difficult to detect. Automated validation tools thus play an important role in protocol validation. We shall see how the above erroneous protocol is validated using a reachability analysis system.

Chapter 3

# REACHABILITY ANALYSIS

Protocol validation via reachability analysis is done by exhaustively generating all global states of a protocol forming a reachability tree to allow the checking of syntactic design errors. The chapter describes the reachability tree construction algorithms for non-FIFO, FIFO, and priority protocols. Proofs of Algorithm 1 and 3 can be found in Vuong 82a, 83a.

## 3.1 REACHABILITY ANALYSIS OF PROTOCOLS WITH NON-FIFO CHANNELS

Before describing the algorithm for constructing a non-FIFO reachability tree, a few definitions have to be introduced.

**Definition 1**

A **global state** in the reachability tree is a **configuration** CF which is a pair <S;T> where

- S is a N-tuple $<s_1,....,s_i,....,s_N>$ with the **local state** $s_i$ representing the current state of process i.
- T is a M-tuple $<t_1,....,t_k,....,t_M>$ where M is the total number of message types, and the **message counter** $t_k$ represents the number of type-k messages currently in the associated channel, say, $c_{ij}$. The notation $(t_{ij})_k$ is used frequently instead of $t_k$ to give a more explicit representation.

**Definition 2**

Let move (CF, $-x_{ij}$) or move (CF, $+x_{ij}$) denotes a move from a given configuration CF by the execution of a transmission $-x_{ij}$ or a reception $+x_{ij}$, respectively, from process i to process j to result in a new configuration CF′. Also, let succ (s, $-x_{ij}$) or succ (s, $+x_{ij}$) denotes the state reached by a process after it

has made the move to transmit or to receive message x in state s. Then, a **valid move** can be defined as follows:

i. move (CF, $-x_{ij}$) is valid if and only if the state succ ($s_i$, $-x_{ij}$) is defined. The resulting configuration CF′ is equal to CF in all elements except:
   - $s'_i$ = succ ($s_i$, $-x_{ij}$) and
   - $(t'_{ij})_x = (t_{ij})_x + 1$

ii. move (CF, $+x_{ij}$) is valid if and only if $(t_{ij})_x > 0$. Thus, a valid reception move corresponds to an executable reception. If the state succ ($s_j$, $+x_{ij}$) is defined, the valid move is said to be **specified**. The resulting configuration CF′ is then equal to CF in all elements except:
   - $s'_j$ = succ ($s_j$, $+x_{ij}$) and
   - $(t'_{ij})_x = (t_{ij})_x - 1$

   However, if succ ($s_j$, $+x_{ij}$) is not defined, the valid move (CF, $+x_{ij}$) is said to be **unspecified.** Clearly, an unspecified move represents an unspecified reception.

**Definition 3**

A reachability tree consists of the following kinds of nodes:

1. **Frontier node** — is a node created by a valid move. It will be renamed to one of the other types of nodes when it has been processed.
2. **Undefined node** — is renamed from a frontier node if it is formed by an unspecified reception.
3. **Duplicate node (repeated node)** — is renamed from a frontier node if it has the same configuration as any of the previously occured nodes. No further moves are necessary from the duplicate node as they will generate the same child nodes as the previously occured one.
4. **Terminal node** — is renamed from a frontier node if no valid move can be initiated. For non-FIFO protocols, a terminal node should simultaneously have its message counters set at zeros (a stable node). In that case, it represents a state

deadlock.

5. **Interior node** — is renamed from a frontier node after it has exhaustively made all valid moves.

With the definitions stated above, an algorithm for constructing a non-FIFO reachability tree is now introduced.

**Algorithm 1 — Reachability tree construction for non-FIFO protocols**

0. The root of the reachability tree is initially defined to be a frontier node with the configuration $CF_0 = <S_0;T_0>$ where $S_0$ represents the initial states of the processes and $T_0$ is the zero message counters.

For each frontier node in the reachability tree, perform the following steps:

1. If it satisfies the condition to be an *undefined node*, rename it and stop processing the node.
2. Likewise if it qualifies to be a *duplicate node.*
3. Likewise if it qualifies to be a *terminal node.*
4. Otherwise, for every valid move from the current configuration, create a new *frontier node* as the child of the current node with the new configuration $CF' = <S';T'>$ set up as described in Definition 2.

   Futhermore, if the new node is a descendant of a node which has a configuration $CF^0 = <S^0;T^0>$ such that $CF' \geq CF^0$, which means $S' = S^0$ (all the corresponding states of $CF'$ and $CF^0$ are equal) and $T' \geq T^0$ (all the message counter components of $CF'$ are greater or equal to the ones of $CF^0$), then those message counters $t'_k$ of $CF'$ such that $t'_k \geq t^0_k \geq 0$ are changed to $t'_k = \omega$, where $\omega$ is a symbol which represents an arbitrarily large number.

For any constant n, we define

$\omega + n = \omega$ ;

$\omega - n = \omega$ ;

$n < \omega$ ; and

$\omega \leq \omega$ [3]

5. If all valid moves from a frontier node had been made, the node is renamed as an *interior node.*

The algorithm halts when all frontier nodes have been processed.

The reachability tree constructed by the above algorithm was proved to be finite for both bounded and unbounded non-FIFO protocols [Vuong 82a]. In the bounded case, all message counters are bounded by the largest corresponding counter value ever found in the tree. If the symbol $\omega$ appears in any message counter, it implies that the counter could contain up to an infinity number of the associated type of messages, and hence the corresponding channel must be unbounded.

## 3.2 REACHABILITY ANALYSIS OF PROTOCOLS WITH FIFO CHANNELS

Unlike the non-FIFO case, the reachability analysis of protocols with FIFO channels also requires the information of message sequencing. To include this information, we can use either the conventional approach which put the messages transmitted explicitly in queues, or the R-state approach which use a state pointer to keep track of the message sequence of each process. The two approaches are described in the rest of the chapter.

---

[3] In other words, if a message counter was set to $\omega$, a transmission or a reception of the associated message type would not change the value of the counter. This allows all arbitrarily large values to be represented by a single symbol. Thus, the configurations with the unbounded message counters being the only varied elements will all be mapped to a single configuration, with the unbounded message counters being set to $\omega$. In this way, the number of nodes generated will become finite.

### 3.2.1 THE CONVENTIONAL APPROACH

Some definitions in the previous section are redefined so that they apply to the conventional approach of reachability analysis for FIFO protocols.

**Definition 4**

A global state configuration CF is a pair <S;C> where

- S has the same meaning as in Definition 1.
- C is an N(N−1)−tuple $<c_{12}, ..., c_{ij}, ..., c_{NN-1}>_{i,j=1,N;\ i \neq j}$ where $c_{ij}$ is the message sequence from process i to process j.

**Definition 5**

A **valid move** in the conventional approach is defined as follows:

i. move (CF, $-x_{ij}$) is valid if and only if the state succ ($s_i$, $-x_{ij}$) is defined. The resulting configuration CF′ is equal to CF in all elements except:
   - $s'_i$ = succ ($s_i$, $-x_{ij}$) and
   - $c'_{ij} = c_{ij}.x_{ij}$

ii. move (CF, $+x_{ij}$) is valid if and only if $x_{ij}$ is the first message in $c_{ij}$. The reception is said to be **specified** if succ ($s_j$, $+x_{ij}$) is defined, resulting in a configuration CF′ equal to CF in all elements except:
   - $s'_j$ = succ ($s_j$, $+x_{ij}$)

   and $c'_{ij}$ and $c_{ij}$ has the relation
   - $c_{ij} = x_{ij}.c'_{ij}$

   If succ ($s_j$, $+x_{ij}$) is not defined, the valid move (CF, $+x_{ij}$) is said to be **unspecified.**

**Algorithm 2 — Reachability tree construction for FIFO protocols: the conventional approach**

The reachability tree construction algorithm in this approach is similar to Algorithm 1 with the following execptions:

- In step 0, the initial configuration should consist of $<S_0;C_0>$ with $S_0$ and $C_0$ representing the initial states of the processes and the empty channels, respectively.
- In step 4, for every valid move from the current configuration, create a new *frontier node* as the child of the current node, with the new configuration $CF' = <S';C'>$ set up as described in Definition 5.

  Furthermore, if the number of messages contained in any channel of the new node exceeds a predefined **channel bound**, then the new node is renamed as an **unbounded node** which will be handled like a *terminal node.*

This algorithm is mainly for bounded FIFO protocols. As for protocols with unbounded channels, the prespecified channel bound has to be set carefully so that only those repeated message sequences would be cut off. The channel bound is necessary because without it the unbounded reachability tree will keep expanding forever.

Another way of stopping an unbounded reachability tree from expanding endlessly is to enforce a level limit. This is not as applicable as the channel bound method, since an unbounded channel might appear at a lower level than other bounded branches. If such a condition happens, an overspecified level limit will lead to large amount of unnecessary nodes generated while an underspecified level limit will result in lost of information.

Yet another alternative is to use a completely different approach called the R-state approach (a name given to the approach used in Vuong 83a). Basically, it uses the technique applied in Algorithm 1 (for non-FIFO protocols) along with an

additional set of state pointers which keep track of the message sequencing in the FIFO channels. The R-state approach is described in the following section.

### 3.2.2 THE R-STATE APPROACH

The R-state approach is basically an extension of the approach used in Algorithm 1 so that it also applies to FIFO protocols. In addition to the original <S;T> configuration, this approach uses a set of receive-state pointers (called r-states) which keep track of the message sequencing, with one r-state per channel. Some new definitions for this approach are introduced as follows:

**Definition 6**

A global state configuration CF is a triple <S;R;T> where:

- S has the usual meaning as in Definition 1.
- R is an N-tuple $<R_1,...,R_i,...,R_N>$ where $R_i$ represents a set of N-1 r-states $<r_{i1},...,r_{ij},...,r_{iN}>_{i\neq j}$ that indicates which messages are to be received next from process i by every other processes. More precisely, $r_{ij}$ represents a state in process i which specifies the set of messages (denoted by $\%r_{ij}$) that can be received next by process j. Message x is in $\%r_{ij}$ if and only if succ ( $\bar{r}_{ij}$, -x) is defined for some $\bar{r}_{ij}$ R-equivalent to $r_{ij}$.
- T has the same meaning as in Definition 1.

**Definition 7**

Two r-states $r_{ij}$ and $\bar{r}_{ij}$ are said to be **R-equivalent** (receive-equivalent) with respect to process j, in notation,

$$r_{ij} :=: \bar{r}_{ij}$$

if and only if

$$r_{ij} = \text{succ}(\bar{r}_{ij}, X) \quad \text{or}$$

$$\bar{r}_{ij} = \text{succ}(r_{ij}, X)$$

for some sequence X which can be empty or contains no transmission of any message

to process j.

**Definition 8**

A valid move in the R-state approach is defined as follows:

i. move (CF, $-x_{ij}$) is valid if and only if the state succ ($s_i$, $-x_{ij}$) is defined. The resulting configuration CF′ is equal to CF in all elements except:

- $s'_i$ = succ ($s_i$, $-x_{ij}$) and
- $(t'_{ij})_x = (t_{ij})_x + 1$

ii. move (CF, $+x_{ij}$) is valid if and only if $x \in \%r_{ij}$ (i.e. the message $x_{ij}$ should be in the set of messages $\%r_{ij}$ that process j can received) and $(t_{ij})_x > 0$. A reception is said to be **specified** if succ ($s_j$, $+x_{ij}$) is defined, resulting in a configuration CF′ equal to CF in all elements except:

- $s'_j$ = succ ($s_j$, $+x_{ij}$)
- $r'_{ij}$ = succ ($\bar{r}_{ij}$, $-x_{ij}$) for some $\bar{r}_{ij} :=: r_{ij}$; and
- $(t'_{ij})_x = (t_{ij})_x - 1$

If succ ($s_j$, $+x_{ij}$) is not defined, the valid move (CF, $+x_{ij}$) is said to be **unspecified.**

**Algorithm 3 — Reachability tree construction for FIFO protocols: the R-state approach**

The reachability tree construction algorithm in the R-state approach is identical to Algorithm 1 except:

- In step 0, the initial configuration also includes the initial r-states which are the initial state of their associated process.
- In step 4, the condition CF′ $\geq$ $CF^0$ requires not only

  $S' = S^0$ and

  $T' \geq T^0$

  but also

  $R' :=: R^0$

In this approach, the reachability tree construction algorithm for non-FIFO protocols (Algorithm 1) is enhanced with the addition of the r-states so that message sequencing is maintained as with the channels in the conventional approach. It has an advantage over the conventional approach in the sense that the reachability tree constructed will always be finite (as in the non-FIFO case).

However, in some cases, the reachability tree of some protocols might contain configurations in which more than one valid reception (specified or unspecified) can be made. That is, there might exist a situation such that for some messages x and y, and processes i and j,

$$x, y \in \%r_{ij} \quad \text{and} \quad (t_{ij})_x > 0 \quad \text{and} \quad (t_{ij})_y > 0.$$

In such a case, the message sequencing is lost and the algorithm fails to provide a definite reachability tree. Protocols which lead to this kind of ambiguity are called **non-well-ordered** protocols. On the other hand, the complementary set of protocols are called **well-ordered** protocols.

## 3.3 REACHABILITY ANALYSIS OF PROTOCOLS WITH PRIORITY CHANNELS

The reachability analysis of protocols with priority channels can be performed using the conventional approach (Algorithm 2) with the following modification:

- In Definition 5, the relation

  $$c'_{ij} = c_{ij} . x_{ij}$$

  upon transmission of a message $x_{ij}$, should be changed to

  $$c'_{ij} = \text{insert } (c_{ij}, x_{ij})$$

  where the insert function returns a message sequence with the message $x_{ij}$ inserted in $c_{ij}$ according to its priority.

An example of a reachability tree constructed using this algorithm will be presented in Chapter 5.

Chapter 4

# IMPLEMENTATION OF THE VALIRA PACKAGE

The VALIRA package is designed based on the algorithms described in the previous chapter. Besides the space and efficiency considerations, the system is implemented with the philosophy that:

- It must be modular, so that debugging and modifying would be easy.
- Its data structures must allow future extensions.
- It should be portable to any system that supports C.

For simplicity, the CFSM models used in the system (and here-in-after) are labeled numerically. Processes and message types are numbered consecutively from '1' onwards, whereas the process states are numbered in a similar way but with the initial state of each process labeled as state 0. The two versions of SAAP in Figure 2.1 and Figure 2.2 thus become the ones shown in Figure 4.1 and Figure 4.2 respectively.

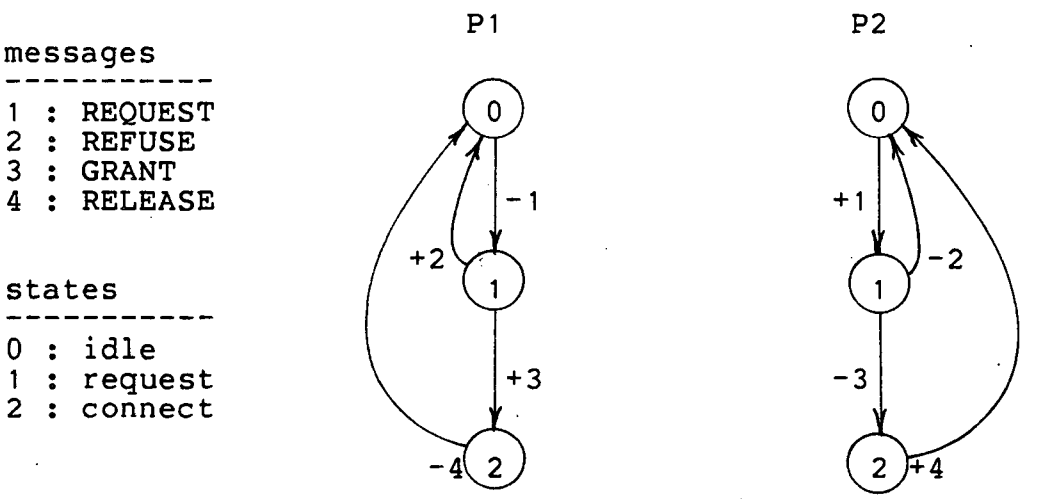



Figure 4.1. The SAAP.

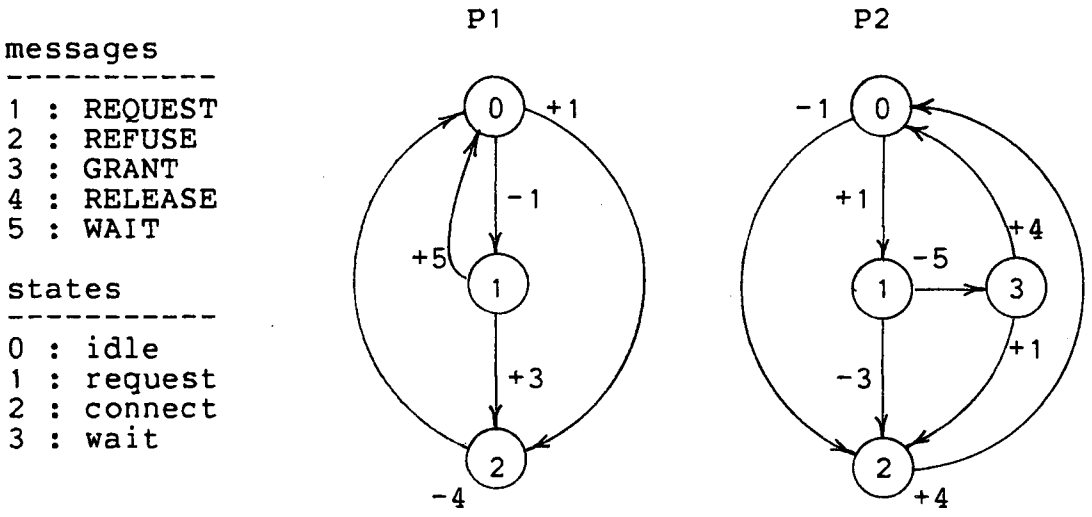



Figure 4.2. The modified SAAP.

The VALIRA is intended to be a stand-alone protocol validation system, which can also form part of an integrated automated system for protocol processing. The entire package was implemented with approximately 2,000 lines of C, which take up 36K bytes of memory. Currently, it is running under UNIX on a VAX 11/750, but it is easily portable to any system that supports C.

## 4.1 SYSTEM OVERVIEW

The VALIRA is essentially composed of five modules, namely, the **command interpreter**, the **transition editor**, the **execution module**, the **I/O-routine module**, and the **storage-allocation-routine module**. The interaction between these modules is shown in Figure 4.3, with the arrows indicating the invocation of modules.

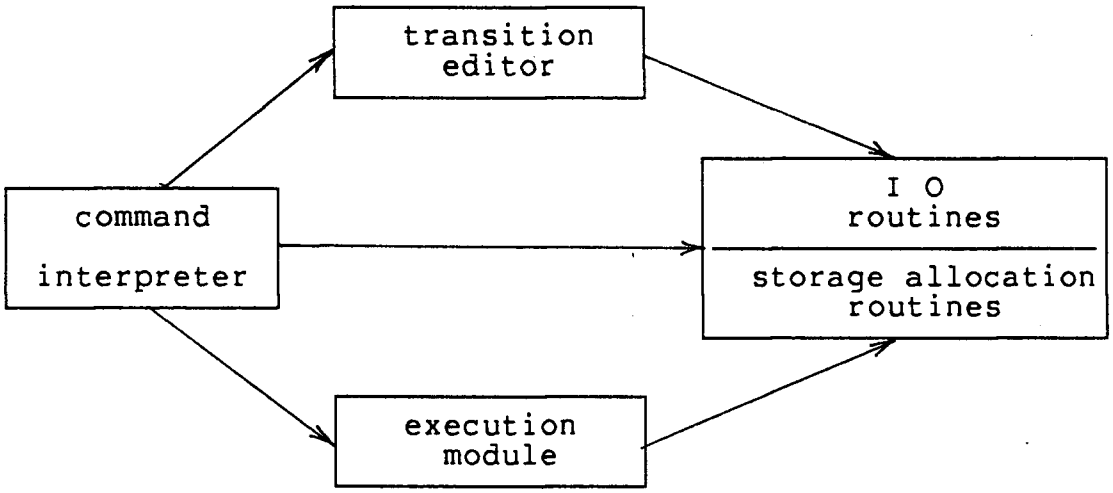



Figure 4.3. Internal organization of the VALIRA.

### 4.1.1 THE COMMAND INTERPRETER

The command interpreter is the interface between the VALIRA and the user. It provides an environment in which the user can refine a protocol specification by invoking the transition editor and the execution module. When interfacing with the interpreter, the user will either be prompted to response with "y" or "n", or, be given a list of commands for selection. Detailed examples for using the system can be found in Appendix A.

### 4.1.2 THE TRANSITION EDITOR

The transition editor, when being invoked by the command interpreter, provides primitive commands such as "replace", "delete", "insert", and "clear" for the user to make appropriate modifications on the set of input transition arcs. The transition arcs are internally kept as linked lists, with one list of transitions per process. Thus, modifications can be done easily on any list of transitions. The transition editor returns

control to the command interpreter when the stop command is received.

### 4.1.3 THE EXECUTION MODULE

The execution module is the main component of the VALIRA. Upon invocation, the execution module generates all global states of a given protocol, forming a reachability tree, and at the same time checks if there is any design error.

The implementation details of the execution module will be described in Section 4.2.

### 4.1.4 THE I/O-ROUTINE MODULE AND THE STORAGE-ALLOCATION-ROUTINE MODULE

The I/O-routine module and the storage-allocation-routine module both provide utility routines for all the other three modules. The I/O-routine module consists of all primitive input and output routines. These primitive routines are invoked frequently by other modules. They may be tailored in other modules in order to fit any specific requirements. The storage-allocation-routine module consists of all storage allocation and storage release routines. All dynamic storage assignments are thus done by invoking the routines in this module.

The usage of the routines in the two named modules are documented internally in the system source code.

## 4.2 DESIGN AND IMPLEMENTATION OF THE EXECUTION MODULE

The execution module is basically a collection of routines implemented based on the algorithms described in Chapter 3. The main design issues of the module are on its data structure design and the implementation methods. The design and implementation details regarding these two aspects are discussed in the sections followed.

### 4.2.1 DATA STRUCTURES

1. Data structure of the reachability tree

The data structure of the reachability tree is simple and intuitive. Each node of the tree consists of some pointer fields and data fields, as shown in Figure 4.4. The pointer fields consist of pointers pointing to the parent, the child, and the sibling of the node; and also a pointer pointing to a global state configuration (represented as a list of states, channels or message counters) whose structure depends on the alogorithm used.

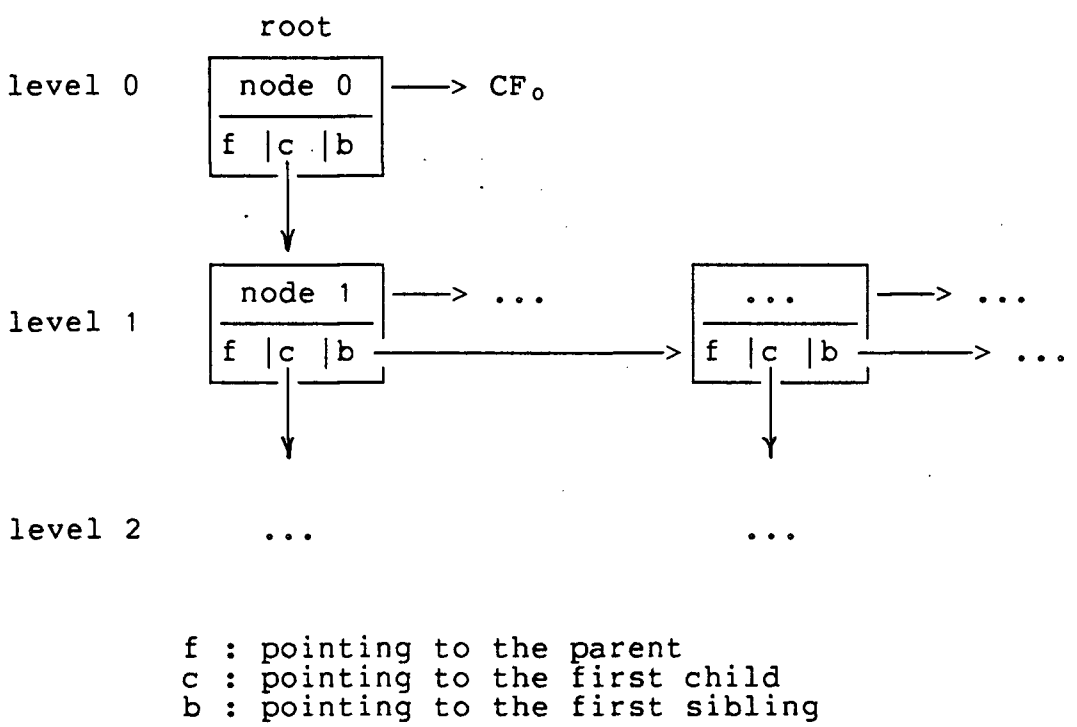



Figure 4.4. Data structure of the reachability tree.

## 2. List structures

The VALIRA uses a number of linked lists to store both user supplied information and self-generated data. The major list structures are described below:

a. *The transition lists*: The transition arcs entered by the user are stored in the transition lists, with one transition list per process. Each node in the list stores information such as the initiating state, the message type (transmitted or received, indicated by the associated sign), and the resulting state of a transition.

b. *The stable-state list*: The stable-state list carries pointers directing to the stable-state nodes in the reachability tree, with each node in the list carring a pointer pointing to a stable-state node.

c. *The state-deadlock list*: Similar to the stable-state list, the state-deadlock list carries pointers directing to the state-deadlock nodes in the reachability tree, with one pointer per node.

d. *The r-state lists*: The r-state lists sustain a record of the R-equivalent states and their associated set of messages $\%r_{ij}$ for all process states in each process with respect to every other processes. There are N - 1 r-state lists per process. Each r-state list is a set of state nodes for all states in the associated process. Each state node in the r-state list consists of:

   - A state in the associated process, say, process i.
   - A sublist of the R-equivalent states of the associated state, with respect to another process, say process j.
   - A sublist of the messages $\%r_{ij}$ receivable at process j from the associated state. The r-state succ $(\bar{r}_{ij}, -x_{ij})$ of each message in $\%r_{ij}$ is also kept with the message in the sublist.

   The r-state lists are used only in the R-approach. With all these

information, conditions such as

$$x \in \%r_{ij} \quad \text{and} \quad \bar{r}_{ij} := r_{ij},$$

and expressions like

$$\text{succ}\ (\bar{r}_{ij}, -x_{ij})$$

can be evaluated easily by refering to the r-state lists.

The construction of these r-state lists will be described in the next section.

### 4.2.2 IMPLEMENTATION STRATEGIES

1. Construction of the r-state lists

The r-state lists are generated by the execution module before the construction of the reachability tree. The structure of the r-state lists was defined previously. The generation of the R-equivalent states and the sets $\%r_{ij}$'s associated with each process state are described below.

a. Generation of the R-equivalent states (function "setrstates")

The R-equivalent states of a given state, say, of process i, with respect to process j, are found by tracing all transition sequences X's which contain no transmission of any message to process j. The trace starts from the given state, and stops when either a message transmission to process j is encountered or a looping of transitions exists. The states reached by any of these sequences X's are the R-equivalent states of the given state.

b. Generation of the set of messages $\%r_{ij}$'s (function "getrstates")

Recall that the notation $\%r_{ij}$ denotes the set of messages associated to a given r-state in process i which can be received next by process j. Thus, the set of messages $\%r_{ij}$ for a given r-state consists of all those messages which are transmitted to process j from the R-equivalent states of the given r-state. While generating the R-equivalent states, these set of

messages can be found at the same time.

Furthermore, each of the r-states succ $(\bar{r}_{ij}, -x_{ij})$ for a message $x_{ij}$ in $\%r_{ij}$ are also generated and kept in the r-state list. The r-state succ $(\bar{r}_{ij}, -x_{ij})$ of a given r-state $r_{ij}$ is generated according to the following steps:

i. Move $r_{ij}$ through the transition $-x_{ij}$ to the next state, say, $r'_{ij}$ (state 1 in Figure 4.5 (i), (ii), and (iii)).

ii. Starting from $r'_{ij}$, move along the transition until either:

- A transmission arc to process j is encountered (state 2 in Figure 4.5 (i)). The r-state succ $(\bar{r}_{ij}, -x_{ij})$ is then set to the state where the trace stopped;
- or, a transition branching exists (state 1 in Figure 4.5 (ii) and (iii)).

  If any of the branches consists of a message transmission to process j, the r-state succ $(\bar{r}_{ij}, -x_{ij})$ is set to the branching state (state 1 in Figure 4.5 (ii)).

  Otherwise, follow the branches to a state where the branches join together (state 0 in Figure 4.5 (iii)), and repeat step ii, with $r'_{ij}$ set to the state where the branches joined. However, if the branches do not join together (and they do not contain any message transmission to process j), the r-state succ $(\bar{r}_{ij}, -x_{ij})$ stays at $r'_{ij}$.

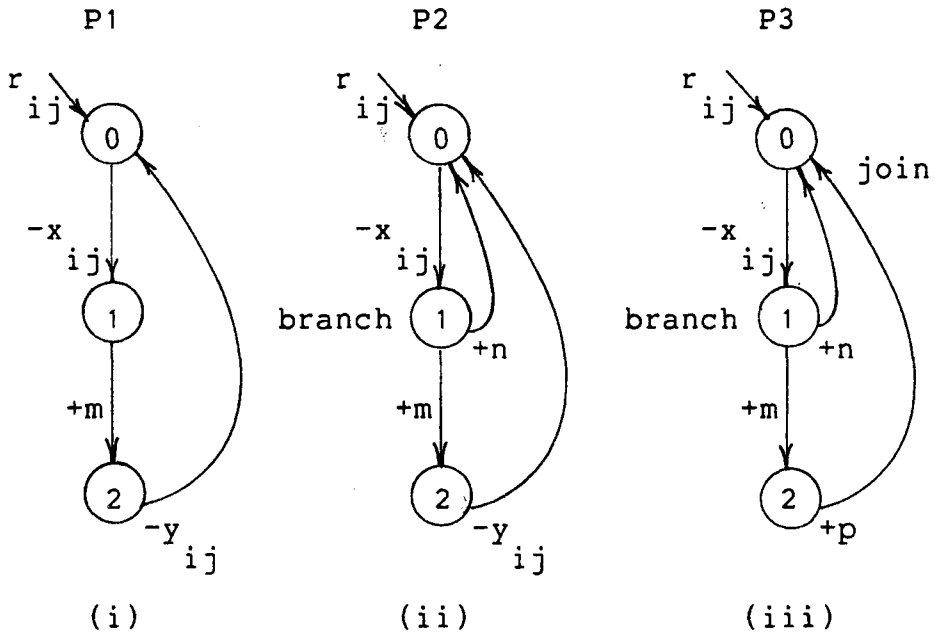



Figure 4.5. Illustration of the r-state searching strategy.

2. Construction of the reachability tree

The construction of the reachability tree is done by the depth-first strategy. The configuration of the initial node of the reachability tree depends on the algorithm used, as defined in Chapter 3. The generation of child nodes from each frontier node is done systematically according to the following orders:

i. Valid moves are checked at the current state of each process in a sequential order from process 1 to process N.

ii. Of the valid moves, message transmissions are performed first, in the same order as were the transmission arcs entered by the user. Message receptions are performed next in the same manner.

The checkings of design errors and unbounded channels are performed while generating every new nodes according to the following sequence:

i. Unspecified receptions

Unspecified receptions are checked at the moment when a valid reception is performed. Note that the checking only applies to message receptions.

ii. Unbounded channels

The content of each message channel (used in Algorithm 2) or message counter (used in Algorithm 1 and 3) is checked every time a new node is created. For Algorithm 2, if the number of messages in any channel exceeds the prespecified channel bound, the new node will be terminated. For Algorithm 1 and 3, the unbounded message counters will be set to the value $\omega$, as described in the algorithms.

iii. Duplicate (repeated) nodes

The configuration of the newly generated node is checked throughout the entire reachability tree to see if there is repetition. The new node is terminated if it is repeated.

iv. Stable nodes

If the newly generated node has all its message channels or message counters empty, the node represents a stable state and is included in the stable-state list.

v. State deadlocks

If the node is a stable node while no valid move can be initiated, the node represents a state deadlock and is included in the state-deadlock list.

The manipulations of r-states in the R-state approach are done according to the following rules:

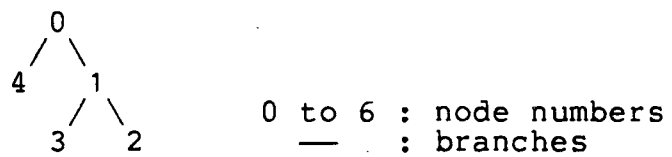
- The initial r-states of each process are set to state 0.
- A r-state does not move in the case of message transmission, unless the transmission is initiated in an empty channel. If a message $x_{ij}$ is transmitted from proess i to process j while the channel $c_{ij}$ is empty (or the associated message counters $t_{ij}$'s all equal zero), then the r-state $r_{ij}$ is set to the state $s_i$ where move $(s_i, -x_{ij})$ is the transmission that is taking place.
- The r-state $r_{ij}$ in process i moves to succ $(\bar{r}_{ij}, -x_{ij})$ when a reception of message x in process j takes place. The r-state succ $(\bar{r}_{ij}, -x_{ij})$ is found by refering to the r-state list associated to the channel $c_{ij}$.

The reachability tree construction procedures mentioned above are implemented in functions "generate", "move", "setbound", "repeated", and "stable" of the execution module.

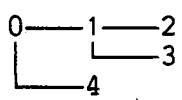
Chapter 5

# RESULTS AND EVALUATION

In order to illustrate the particular features of various approaches used in VALIRA, test runs of protocols with different properties are examined in this chapter. Each of the examples illustrated includes a protocol specification, represented by a CFSM model, and the corresponding reachability tree. The reachability tree will be printed horizontally from left to right. For example, the tree:




will be printed as:




For simplicity, the branches in the reachability tree will not be shown on the diagrams that appear later or on the actual reachability tree print out generated by the VALIRA. More detailed explanations on all representations will be given in Example 1.

## 5.1 EXAMPLES

The sample protocols given in this section are designed to illustrate the particular features of various approaches used in VALIRA, with as little complexity involved as possible.

**Example 1** — An unbounded non-FIFO protocol

The protocol given in this example (Figure 5.1) is to demonstrate the competency of Algorithm 1 for unbounded non-FIFO protocols. In the given protocol, process 1 can keep sending a request message (message 1) until it is acknowledged by process 2 (via message 2). Since there is no limitation on the number of type 1 message sent, $c_{12}$ would be an unbounded channel.

Figure 5.2 (i) on the following page shows the reachability tree generated for the protocol, with each global state in the tree represented by a node number. Each repeated node in the reachability tree is signified by a **transition** followed by the node number of the previous occured node. For example, (P1-1)[ 1] in the first branch of the tree represents that node 1 is repeated through the transition "process 1 transmits message 1". The global state value where a numbered node represented can be found in its corresponding entry within the node table shown in Figure 5.2 (ii). Say, entry 0 in the node table (i.e. (root)00000) corresponds to node 0; entry 1 (i.e. 0(P1-1)00ω00) corresponds to node 1, and so on. The first number in each entry (0 for entry 1) is the parent node number of the associated node. This is followed by a transition and the global state value of the node. For instance, (P1-1) in entry 1 represents the transition "process 1 transmits message 1" initiated by the parent node (node 0). This results in a current global state value 00ω00. Recall that the configuration <S;T> is

```
messages
-----------
1 : REQUEST
2 : GRANT
3 : RELEASE

states
-----------
0 : idle
1 : connect
```

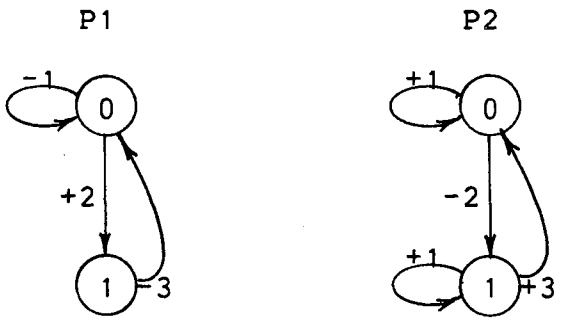
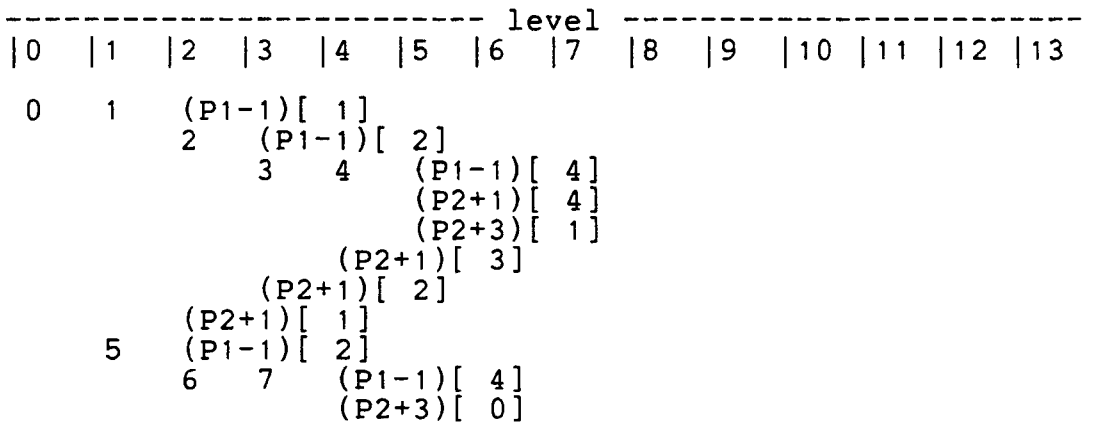
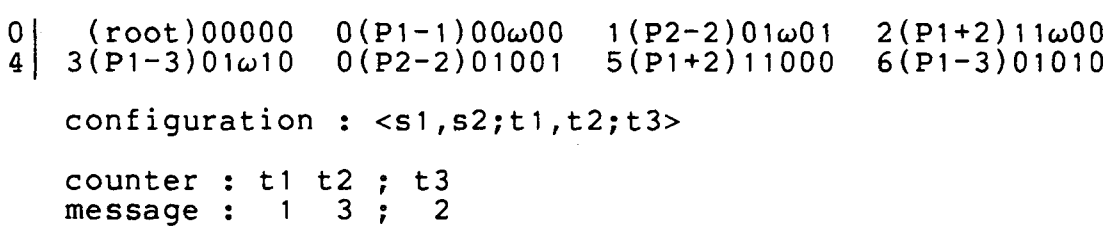



Figure 5.1. An unbounded non-FIFO protocol.

used in Algorithm 1. Thus, in this protocol, 00ω00 corresponds to <s1,s2;t1,t2;t3>, with message counter t1, t2, and t3 associated with message type 1, 3, and 2, respectively. Note that the symbol ω (generated according to the algorithm) in t1 indicates that the type 1 message is unbounded and consequently $c_{12}$ as well. Besides this, the protocol is validated to be free of any design error.




(i) The reachability tree




(ii) The node table

Figure 5.2. The reachability tree for Example 1.

**Example 2** — A bounded FIFO protocol

In Section 2.2, an erroneous FIFO protocol was given as an example (Figure 2.2, Figure 4.2) to illustrate the four types of design errors. Here, the protocol is validated using the conventional approach (Algorithm 2). The results obtained are shown in Figure 5.3 and 5.4.

```
------------------------- level -----------------------
|0  |1  |2  |3  |4  |5  |6  |7  |8  |9  |10 |11 |12 |13

 0   1   2   (P1+1)[UR]
             (P2+1)[UR]
         3   4   5   6   7   (P2+4)[ 1]
                         (P2+4)[ 0]
             8   9   10  11
        12  (P1-1)[ 2]
            (P1+1)[ 5]
```

(i) The reachability tree

```
 0|   (root)0000000   0(P1-1)1010000   1(P2-1)1210100
 3|  1(P2+1)1100000   3(P2-3)1200010   4(P1+3)2200000
 6|  5(P1-4)0201000   6(P1-1)1211000   3(P2-2)1300001
 9|  8(P1+2)0300000   9(P1-1)1310000  10(P2+1)1200000
12|  0(P2-1)0200100

     configuration : <s1,s2;t1,t2;t3,t4,t5>

     counter : t1 t2 ; t3 t4 t5
     message :  1  4 ;  1  3  2
```

(ii) The node table

Figure 5.3. The reachability tree for the modified SAAP.

Figure 5.3 shows the reachability tree of the protocol. Unspecified receptions are denoted by [UR]'s, and with each [UR] preceded by the transition that was taken place. Also note that the reachability tree generated by the R-state approach for the same protocol will be different from this one, since different node configurations are being used in the two approaches.

```
S U M M A R Y

1. The list of deadlock node(s) :-
   11

2. Stable states :-

   P1  P2
  ------
    0   0    (node 0)
    1   1    (node 3)
    2   2    (node 5)
    0   3    (node 9)
    1   2    (node 11)

3. Non-executable instruction(s) :-
   process 2 :    3 -> 0    +4

4. The first unspecified reception is detected at node 2.
   The lowest level UR is detected at node 2 (level 3).

5. The reachability tree is bounded at level 6.

Max channel queue length = 2.
```

Figure 5.4. An error summary for the modified SAAP.

Figure 5.4 is an error summary generated by the VALIRA, in which state deadlocks and non-executable instructions in the protocol are reported. State ambiguities are revealed in the stable-state table in the error summary. If there exists a column such that two entries containing the same states, state ambiguity occurs. For example, the pair (0,0) and (0,3) in the stable-state table indicates that state 0 in process 1 can coexist stably with either state 0 or state 3 in process 2. The above observation can be extended to the multi-process case.

The results as noticed in the error summary are consistant with those discussed in Section 2.2.

**Example 3** — An unbounded FIFO protocol

A simple unbounded FIFO protocol is given here as an example to illustrate how Algorithm 3 (the r-state approach) works on the unbounded protocols. The protocol, as shown in Figure 5.5, consists of two states in each process. Since process 1 can transmit message 1 and 2 repeatedly, $c_{12}$ is clearly an unbounded channel. In this example, the r-state approach is applied, with the results shown in Figure 5.6.

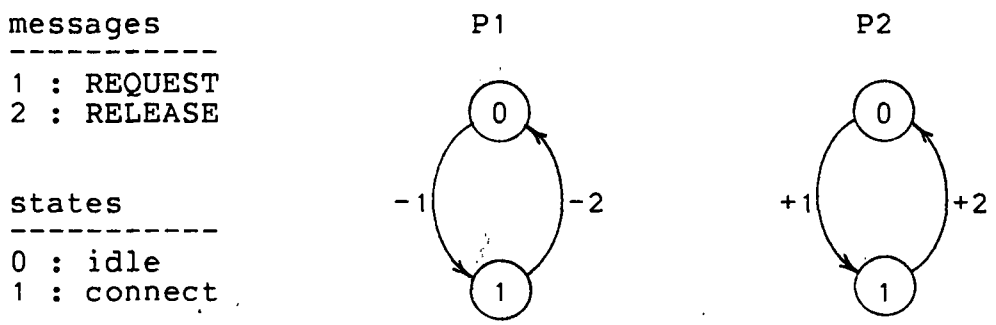



Figure 5.5. An unbounded FIFO protocol.

```
------------------------- level -------------------------
|0  |1  |2  |3  |4  |5  |6  |7  |8  |9  |10 |11 |12 |13

 0   1   2   3   (P1-2)[ 2]
                 4   5   (P1-1)[ 4]
                             (P2+2)[ 2]
                         (P2+3)[ 3]
             (P2+1)[ 5]
         6   7   8   (P1-2)[ 5]
                     (P2+1)[ 1]
                 (P2+2)[ 0]
```

(i) The reachability tree

```
0|    (root)000000    0(P1-1)100010    1(P1-2)0000ωω
3|  2(P1-1)1000ωω    3(P2+1)1110ωω    4(P1-2)0110ωω
6|  1(P2+1)110000    6(P1-2)011001    7(P1-1)111011

    configuration : <s1,s2;r12;r21;t1,t2>

    counter : t1 t2
    message :  1  2
```

(ii) The node table

Figure 5.6. The reachability tree for Example 3.

**Example 4** — An unbounded non-well-ordered FIFO protocol

The protocol shown in Figure 5.7 is similar to the one discussed in Example 3, but with an extra message transmitting at state 0 of process 1. It is obvious that the protocol is non-well-ordered for it is possible that both message 1 and message 3 exist in the channel simultaneously. Thus, the conventional approach has to be applied. The reachability tree constructed by the conventional approach, with a prespecified channel bound of 3, is shown in Figure 5.8. The protocol is validated to be free of design errors.

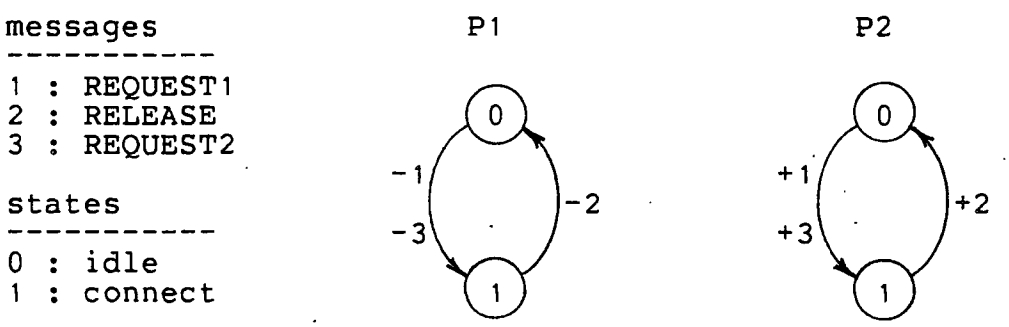



Figure 5.7. An unbounded non-well-ordered FIFO protocol.

```
------------------------ level ------------------------
|0  |1  |2  |3  |4  |5  |6  |7  |8  |9  |10 |11 |12 |13

 0   1   2   3   (P1-2)[UC]
             4   5   (P1-1)[UC]
                     (P1-3)[UC]
                     (P2+2)[ 2]
                 (P2+2)[ 1]
             6   (P1-2)[UC]
                 7   8   (P1-1)[UC]
                         (P1-3)[UC]
                         9   10  (P1-2)[UC]
                                 (P2+3)[ 4]
                             11  (P1-2)[UC]
                                 (P2+3)[ 7]
                             12  (P1-1)[ 4]
                                 (P1-3)[ 7]
                                 (P2+2)[ 0]
                         13  (P1-2)[ 9]
                             14  (P1-2)[12]
             (P2+1)[12]
         (P2+1)[14]
     (P1-3)[13]              [UC] : channel bound exceeded
```

(i) The reachability tree

```
 0|  (root)00000  0(P1-1)10100  1(P1-2)00101  2(P1-1)10201
 4| 3(P2+1)11101  4(P1-2)01102  2(P1-3)10111  6(P2+1)11011
 8| 7(P1-2)01012  8(P2+2)00011  9(P1-1)10111  9(P1-3)10021
12| 9(P2+3)01001  7(P2+2)10010 13(P2+3)11000

    configuration : <s1,s2;t1,t2,t3>

    counter : t1 t2 t3
    message :  1  3  1
```

(ii) The node table

Figure 5.8. The reachability tree for Example 4.

**Example 5** — A priority protocol

The model shown in Figure 5.9 is a priority protocol with the priority of message 2 greater than that of message 1. The reachability tree generated is shown in Figure 5.10, with no error detected.

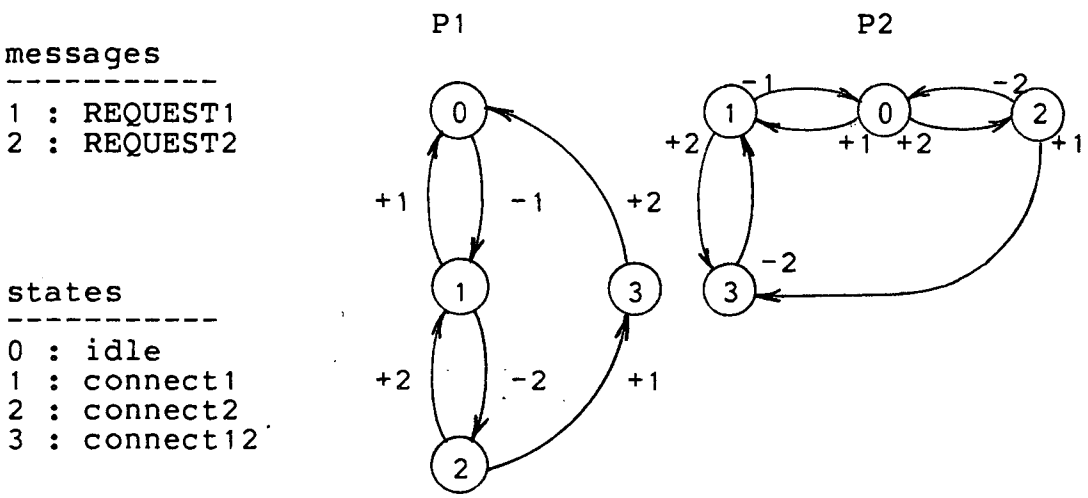
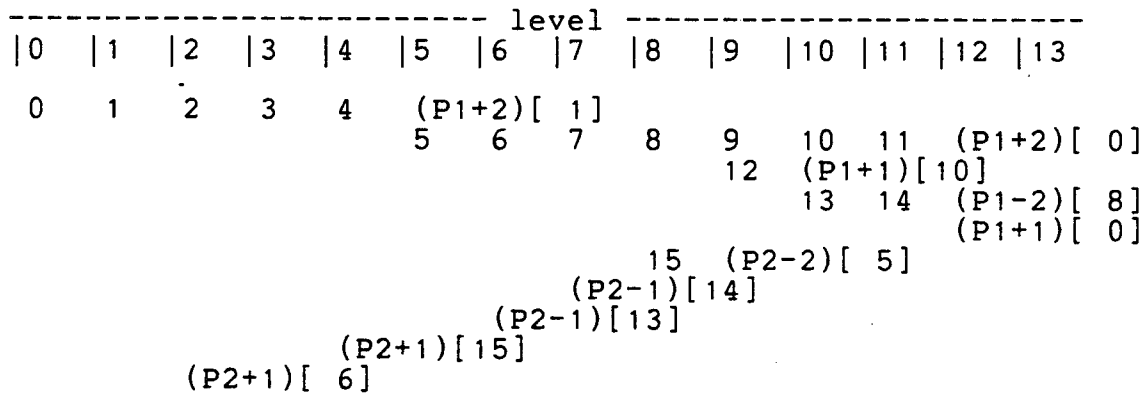



Figure 5.9. A priority protocol.




(i) The reachability tree

```
 0|   (root)000000    0(P1-1)101000    1(P1-2)201100
 3|  2(P2+2)221000    3(P2-2)201001    4(P2+1)210001
 6|  5(P1+2)110000    6(P1-2)210100    7(P2-1)200110
 9|  8(P1+1)300100    9(P2+2)320000   10(P2-2)300001
12|  8(P1+2)220010   12(P2-2)200011   13(P1+2)100010
15|  7(P2+2)230000

     configuration : <s1,s2;t1,t2;t3,t4>

     counter : t1 t2 ; t3 t4
     message :  1  2 ;  1  2
```

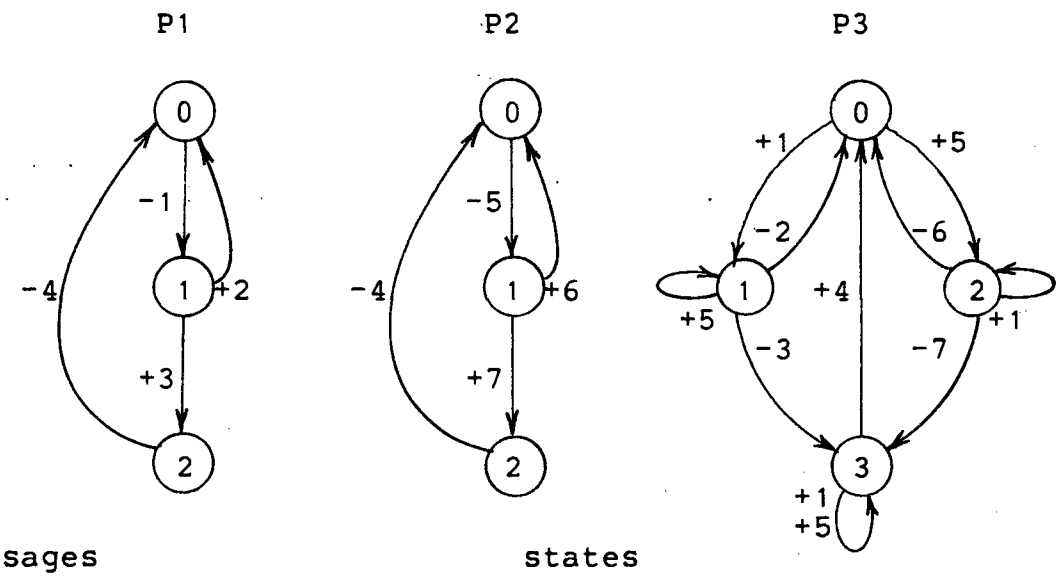
(ii) The node table

Figure 5.10. The reachability tree for Example 5.

**Example 6** — A multi-process FIFO protocol

The three-process FIFO protocol shown in Figure 5.11 is a multi-process version of the SAAP, with process 1 and 2 being the requesting processes. Note that each message in the protocol is transmitting only to a single recipient, which is a limitation of the current implementation. In other words, multi-casting is not allowed in the current version of VALIRA. As a result of the non-multi-casting nature, unspecified receptions in the protocol can be handled in the same way as in the two-process case.

The reachability tree generated for this protocol consists of twenty-one levels and includes up to eighty-one different global states. It is free of most design errors except for state ambiguities as shown from the stable-state table in figure 5.12.




| messages | states | | |
|---|---|---|---|
| 1 : request1 | Process 1 | Process 2 | Process 3 |
| 2 : refuse1 | 0 : idle | 0 : idle | 0 : idle |
| 3 : grant1 | 1 : request1 | 1 : request2 | 1 : request1 |
| 4 : release | 2 : connect | 2 : connect | 2 : request2 |
| 5 : request2 | | | 3 : connect |
| 6 : refuse2 | | | |
| 7 : grant2 | | | |

Figure 5.11. A multi-process FIFO protocol.

```
   Stable states :-

 P1  P2  P3
---------
  0   0   0    (node 0)
  0   1   2    (node 6)
  1   0   1    (node 10)
  2   0   3    (node 13)
  1   1   1    (node 19)
  0   1   0    (node 21)
  2   1   3    (node 23)
  1   2   3    (node 54)
  0   2   3    (node 56)
  1   1   2    (node 70)
  1   0   0    (node 72)
```

Figure 5.12. The stable-state table for Example 6.

The print out of the tree and the node table is not included because of the size involved by the reachability tree.

## 5.2 EVALUATION

As an example to illustrate the performance of the current installation of VALIRA running on a VAX 11/750, the packet level of the CCITT Recommendation X.75 protocol was tested during light system-load period. The CFSM model of the X.75 protocol used in the validation followed the one given in Vuong 83b, which consists of twelve process states and ten message types in each of the two processes. Since the protocol is found to be both unbounded and non-well-ordered, the conventional approach was used instead of the R-state approach. With a channel bound of two, VALIRA created 1,232 different global states and went up to as high as level 121, within approximately 1,750 CPU sec, or two hours real time. The results obtained contain neither state deadlock nor non-executable interactions, but several unspecified receptions and state ambiguities exist as indicated in Vuong 83b.

In order to explore the physical limit of the current installation, the X.75 model was run with a high channel bound until the available memory space was exhausted. This gave a total of approximately 15,000 different global states for the entire run.

The above results only give a rough estimation of the global state limit when the conventional approach is applied. The actual global state limit, however, depends on a number of variables such as the number of processes and message types in the protocol. All these would affect the memory space required in each global state configuration.

When considering the space and time issues, notice that the approach used in Algorithm 1 and 3 is superior to the conventional approach in the following aspects:

1. Less space is required for each global state, as message counter is used instead of message channel.
2. There is more efficiency in comparing two global state values because no channel queue is required.

As a consequence, FIFO protocols are more desirable to be validated using the R-state approach. The conventional approach, however, is used in complement to the R-state approach whenever a given protocol is found to be non-well-ordered.

Chapter 6

# CONCLUSIONS

## 6.1 THESIS SUMMARY

The VALIRA, a protocol validation package which based on the conventional approach and a newly developed approach in reachability analysis has been implemented. This package provides users with an automated protocol validation environment which can be used either for practical purposes or as an educational tool.

The various approaches used in VALIRA have been described in Chapter 3 of the thesis. In summary, the completeness of the algorithms used with respect to each type of protocol is recapitulated in the table below:

| | *type of protocol* | *algorithm/configuration* | | *completeness* |
|---|---|---|---|---|
| 1. | non-FIFO | Algm 1 | <S,T> | complete |
| 2. | FIFO (well-ordered) | Algm 3 | <S,R,T> | complete |
| 3. | FIFO (non-well-ordered, bounded) | Algm 2 | <S,C> | complete |
| 4. | FIFO (non-well-ordered, unbounded) | Algm 2 | <S,C> | no |
| 5. | priority (bounded) | Algm 2 | <S,C> | complete |
| 6. | priority (unbounded) | Algm 2 | <S,C> | no |

A reachability tree construction algorithm is "complete" with respect to a type of protocol if and only if all possible global states of any given protocol of the corresponding type can be generated by the algorithm within a finite bound. As noticed from the table, certain types of unbounded protocols cannot be completely validated. Their protocol syntactic properties have been known to be undecidable [Brand 80, 81]. The level of completeness of these types of protocols, as validated via the conventional approach (Algorithm 2), thus depends on the value of the channel bound enforced in the analysis. In turn, this depends on the memory space available in the machine used.

When a protocol to be validated is large and complex, such as the X.75 protocol, one can apply a decomposition technique as described in Vuong 82b, 83b. The decomposition technique splits a large and complex protocol into smaller components so that it becomes more manageable. Incorporation with this technique, the application of VALIRA is greatly extended.

The competency of the reachability analysis system itself mainly depends on the memory space and the CPU time available. These two factors are always being considered to be the major barriers in reachability analysis. Nevertheless, with the trend of modern technology development, the two limitations should hopefully not be the dominating factors in the near future.

## 6.2 FUTURE WORK

Currently, VALIRA is running under UNIX on a VAX 11/750, yet it is portable to any system that supports C. Further enhancements of the system are possible, for instance:

1. The current implemention can be extended to cover the multi-process multi-casting protocols.
2. The R-state approach could be enhanced by introducing a reception counter D in the existing <S;R;T> configuration [Vuong 83a]. This would give an expansion to the set of well-ordered protocols.
3. Since the system is designed to be portable, with slight modifications, it can be available as a commercial production system.

## BIBLIOGRAPHY

[Bochmann 78]

Bochmann, G. V., "Finite State Description of Communication Protocols," *Computer Networks,* pp. 361-372 (October 1978).

[Bochmann 80]

Bochmann, G. V. and Sunshine, C., "Formal Methods in Communication Protocol Design," *IEEE Trans. on Communications* COM-28(4). pp. 624-631 (April 1980).

[Brand 80]

Brand, D. and Zafiropulo, P., "Synthesis of Protocols for Unlimited Number of Processes," *Proc. Trends and Applications: 1980 Computer Network Protocols, NBS,* (May 1980).

[Brand 81]

Brand, D., "On Communicating Finite-State Machines," IBM Research Report RZ 1053 (January 1981).

[Gouda 76]

Gouda, M. G. and Manning, E. G., "On the Modeling, Analysis, and Design of Protocols - A Special Class of Software Structures," *Pro. 2nd International Computer on Software Engineering,* (October 1976).

[Merlin 76]

Merlin, P. M. and Farber, D. J., "Recoverability of Communication Protocols - Implementations of a Theoretical Study," *IEEE Trans. Communications* COM-24(9) pp. 1036-1043 (September 1976).

[Rudin 78]

Rudin, H., West, C., and Zafiropulo, P., "Automated Protocol Validation: One Chain of Development," *Computer Networks,* pp. 373-380 (October 1978).

[Schwabe 81]

Schwabe, D., "Formal Specification and Verification of a Connection Establishment Protocol," *Proceedings, Seventh Data Communications Symp.,* Mexico City

(October 1981).

[Stenning 70]

Stenning, N. V., "A Data Transfer Protocol," *Computer Networks,* (1) pp. 99-110 (1976).

[Sunshine 78]

Sunshine, C. A., "Survey of Protocol Specification and Verfication Techniques," *Computer Networks,* (October 1978).

[Symons 80a]

Symons, F. J. W., "Introduction to Numerical Petri Nets, A General Graphical Method of Concurrent Processing Systems," *Australian Telecommunication Research* 14(1) pp. 28-32 (1980).

[Symons 80b]

Symons, F. J. W., "The Verification of Communication Protocols Using Numerical Petri Nets," *Australian Telecommunication Research* 14(1) pp. 34-38 (1980).

[Vuong 82a]

Vuong, S. T. and Cowan, D. D., "Reachability Analysis of Protocols with non-FIFO Channels," *Proc. COMPCON Fall 82,* (September 1982).

[Vuong 82b]

Vuong, S. T. and Cowan, D. D., "A Decomposition Method for the Validation of Structured Protocols," *Proc. INFOCOM Conference,* (April 1982).

[Vuong 83a]

Vuong, S. T. and Cowan, D. D., "Reachability Analysis of Protocols with FIFO Channels," *Proc. ACM SIGCOMM '83 Symp. on Communications Architecture and Protocols,* (March 1983).

[Vuong 83b]

Vuong, S. T. and Cowan, D. D., "Automated Protocol Validation via Resynthesis: The CCITT X.75 Packet level Recommendation as an Example," *Journal of Telecommunication Networks,* pp. 153-176 (Summer 1983).

[West 78]

West, C. H. and Zafiropulo, P., "Automated Validation of a Communication Protocol: The CCITT X.21 Recommendation," *IBM J. Research and Development* 22 pp. 60–71 (January 1978).

[Zafiropulo 78]

Zafiropulo, P., "Protocol Validation by Dulogue–Matrix Analysis," *IEEE Trans. Communications* pp. 1187–1194 (August 1978).

[Zafiropulo 80]

Zafiropulo, P., West, C. H., Cowan, D. D. and Brand, D., "Towards Analyzing and Synthesizing Protocols," *IEEE Trans. Communications* COM–28(4) pp. 651–660 (April 1980).

# APPENDIX A - VALIRA USER'S MANUAL

## I. INTRODUCTION

VALIRA is a protocol validation package that takes the transitions of a protocol (specified in a CFSM model) and performs the validation via reachability analysis. The syntactic properties of the protocol, such as the state ambiguities, the state deadlocks, the unspecified receptions, the non-executable interactions, and the unbounded channels, will be analysed and reported.

The commands used in VALIRA are self-explanatory. Basically, a user can apply the package without reading the command explanation part of the manual (however, the output representation part must be read). Commands can be entered in either upper or lower case. Illegal commands or inputs will be signalled by a "bid" sound for re-entering.

This manual is intended as a user reference. As for the internal structures or background theories of VALIRA, interested users can refer to the corresponding part of the thesis.

## II. HOW TO RUN VALIRA

VALIRA is currently installed on our department's CS VAX. It is invoked with the command

```
/user/vuong/protoval/package
```

VALIRA provides a two-level interactive environment which includes :

1. The interpreting level, where, in general, the user can monitor the validation process.
2. The editing level, where the user can use the editing commands provided to make changes on the entered transitions. The editing level is invoked on the interpreting level, so it is one level below.

**A. The invocation phase**

Upon invocation of the package, the user automatically enters the interpreting level, and will be prompted by the interpreter to respond with either "y" or "n" for information required by the package. The examples below show the initial invocation phase for each type of protocols.

**Example A.1** Running a two-process priority protocol

```
Execution begins

How many processes ? 2
Priority channels ? [y|n|q] y
Please enter channel bound ? 3
```

**Example A.2** Running a two-process non-FIFO protocol

```
Execution begins

How many processes ? 2
Priority channels ? [y|n|q] n
FIFO channels ? [y|n|q] n
```

**Example A.3** Running a two-process FIFO protocol

```
Execution begins

How many processes ? 2
Priority channels ? [y|n|q] n
FIFO channels ? [y|n|q] y
Please enter channel bound ? 3
```

At this point, the user can choose between using the R-state approach or the conventional approach. In general, the R-state approach should be applied, since it is more efficient. However, the R-state approach cannot validate every

types of FIFO protocols. The message "non-well-ordered protocol" will appear if the R-state approach fails to work. In that case, the conventional approach has to be used.

**B. The input phase**

When the queries in the invocation phase have responded, the interpreter will enter the input phase where the user is prompted to input the transitions for each process.

```
Enter transitions for process 1 :-
>0 1 -1
>1 0 1
>
```

The prefix character in the input phase is a ">", and the transitions are entered in the form:

0 1 -1

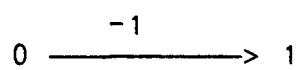
where

0 is the initiating process state of a transition

1 is the resulting process state of a transition

-1 is the transition arc between the two process states

Thus, the above input corresponds to the transition:




The input phase is terminated by entering a null line.

The input of the process states and the message types have the following restrictions:

- process state: must be numbered from 0 upto at the maximum of 35. Note that the initial state of each process must be state 0.
- message type: must be numbered within 1 and 35.

**C. The editing phase**

Every time an input phase is terminated, the interpreter will automatically invoke the transition editor so that changes are possible. The prefix character given by the transition editor is a ":", which is followed by entering one of the following commands:

```
h ... print the message
s ... stop editing
c ... clear all transitions
p ... print all transitions
r n . replace transition n
d n . delete transition n
i n . insert transition(s) after transition n
```

The usage of the above commands is like in any simple line editor.

**D. The execution phase**

The interpreter starts the execution phase when the transitions of all processes are entered. The information entered are echoed to the user, followed by the queries :

```
Enter node limit of the tree :
Do you want the reachability tree printed ? [y|n|q] :
```

They should be replied accordingly. The node limit entered would be used as the global state limit of the reachability tree.

The generation of the reachability tree starts at this point. Shown on the screen will be the print out of a reachability tree, a node table, and an error summary. Some special notations used in the output will be explained futher in section III.

**E. The command phase**

The interpreter enters the command phase after the execution on generating the reachability tree. A list of commands will be printed for selection, as shown below:

```
1. Print the reachability tree
2. Print the node table
3. Print the summary
4. Edit transitions
5. Run
6. Quit

Item number ?
```

These commands provide the user with an environment in which the protocol entered can be modified and rerun. The commands are selected by entering its item number. Say, if "4" is entered, the interpreter will prompt

```
process number ?
```

The process number of the desired process should then be entered. The transition editor (Section II C) will be invoked to edit the transitions of that process. Upon termination (by the "s" command in the transition editor), the control will be returned to the interpreter in the command phase.

### III. Output representations

This section describes the representations used in printing the reachability tree, the node table, and the error summary. To illustrate this, the sample run of an erroneous protocol (Figure A.1) is shown in Figure A.2.

```
messages
-----------
1 : REQUEST
2 : WAIT
3 : GRANT
4 : RELEASE

states
-----------
0 : idle
1 : request
2 : connect
3 : wait
```

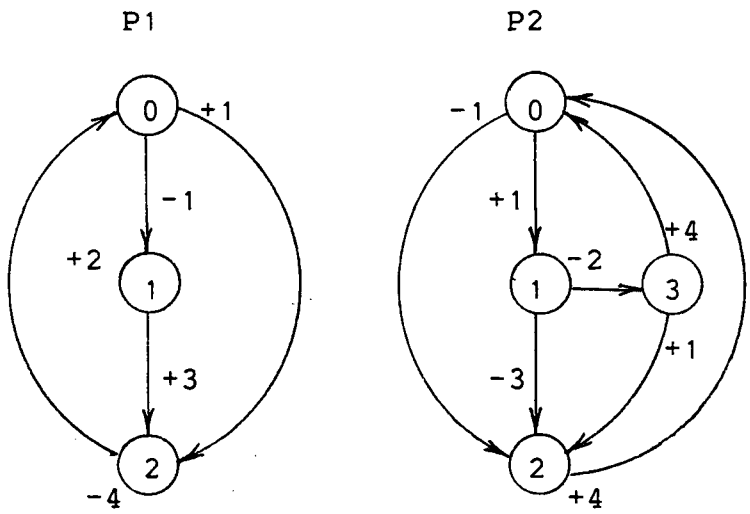



Figure A.1. An erroneous protocol.

Figure A.2. A sample terminal session.

```
Execution begins

VALIRA Version 1

How many processes ? 2
Priority channals ? [y|n|q] n
FIFO channels ? [y|n|q] y
Do you want the R-state approach ? [y|n|q] n
Please enter channel bound ? 3
Enter transitions for process 1 :-
Sample input : 0 1 -1
(terminate with a null line)
> 0 1 -1
> 0 2 1
> 1 0 2
> 1 2 3
> 2 0 -4
>
:s
Enter transitions for process 2 :-
Sample input : 0 1 -1
(terminate with a null line)
>0 1 1
>0 2 -1
>1 2 -3
>1 3 -2
>3 0 4
>3 2 1
>2 0 4
>
:s

Reachability analysis of protocol with FIFO channels

Process 1 :
0 -> 1  -1
0 -> 2  +1
1 -> 0  +2
1 -> 2  +3
2 -> 0  -4

Process 2 :
0 -> 1  +1
0 -> 2  -1
1 -> 2  -3
1 -> 3  -2
3 -> 0  +4
3 -> 2  +1
2 -> 0  +4

process 1 transmits 2 message(s) :  1  4
process 2 transmits 3 message(s) :  1  3  2
```

```
Enter node limit of the tree : 100
Do you want the reachability tree printed ? [y|n|q] y


------------------- level -----------------
|0   |1   |2   |3   |4   |5   |6   |7   |8   |9   |10

 0    1    2    (P1+1)[UR]
                (P2+1)[UR]
           3    4    5    6    7    (P2+4)[ 1]
                                    (P2+4)[ 0]
                8    9    10   11
      12   (P1-1)[ 2]
           (P1+1)[ 5]

There are a total of 12 different nodes
Do you want the node table ? [y|n|q] y

Node representation :
(s1,s2;t1,t2;t3,t4,t5)

        Message counter :  t1  t2  t3  t4  t5
Associated message type :   1   4   1   3   2

 0|   (root)0000000  0(P1-1)1010000  1(P2-1)1210100
 3|  1(P2+1)1100000  3(P2-3)1200010  4(P1+3)2200000
 6|  5(P1-4)0201000  6(P1-1)1211000  3(P2-2)1300001
 9|  8(P1+2)0300000  9(P1-1)1310000 10(P2+1)1200000
12|  0(P2-1)0200100

E R R O R   S U M M A R Y

1. The list of deadlock node(s) :-
   11

2. Stable states :-

   P1 P2
  ------
    0  0    (node 0)
    1  1    (node 3)
    2  2    (node 5)
    0  3    (node 9)
    1  2    (node 11)

3. Non-executable instruction(s) :-
   process 2 : 3 -> 0   +4

4. The first unspecified reception is detected at node 2.
   The lowest level UR is detected at node 2 (level 3).

5. The reachablity tree is bounded at level 6.

Max channel queue length = 2.
```

```
Choose one item from the list below :
1. Print the reachablisty tree
2. Print the node table
3. Print the summary
4. Edit transitions
5. Run
6. Quit

Item number ? 6

End of run
```

**A. Representation of the reachability tree**

As noted in Figure A.2, the reachability tree is printed horizontally from left to right, with each node in the tree represented by a node number. Figure A.3 shows the same reachability tree along with branches connecting between every pair of parent and child nodes. For simplicity, these branches are not shown in the actual print out.

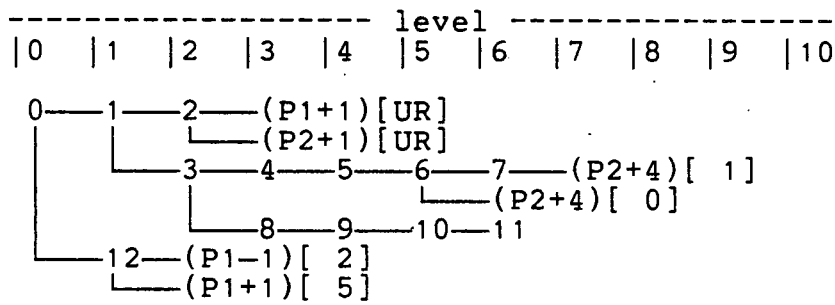



Figure A.3. A reachability tree with branches shown.

In most cases, the reachability tree printed would go beyond the right margin. To solve this problem, the portions which go beyond the right margin will be restarted printing from the left margin, on a new level scope. An example of this is shown in Figure A.4 and Figure A.5.

```
                                                        margin-->|
------------------- level ------------------                    |
|0   |1   |2   |3   |4   |5   |6   |7   |8   |9   |10          |
                                                                |
 0    1    2     (P1+1)[UR]                                     |
             (P2+1)[UR]                                         |
           3    4    5    6    7    8    9    10   11   |12   13|
                                                        |14     |
           15   16   17   18   19                              |
      20   (P1-1)[ 2]                                          |
           (P1+1)[ 5]                                         |
```

Figure A.4. A reachability tree that goes beyond the right margin.

```
------------------- level ------------------
|0   |1   |2   |3   |4   |5   |6   |7   |8   |9   |10

 0    1    2     (P1+1)[UR]
             (P2+1)[UR]
           3    4    5    6    7    8    9    10   11

------------------- level ------------------
|11  |12  |13  |14  |15  |16  |17  |18  |19  |20  |21


 12   13
 14

------------------- level ------------------
|0   |1   |2   |3   |4   |5   |6   |7   |8   |9   |10

           15   16   17   18   19
      20   (P1-1)[ 2]
           (P1+1)[ 5]
```

Figure A.5. A reachability tree printed in different scopes of levels.

To represent the different kind of nodes in the reachability tree, the following notations are being used:

1. Unspecified node

   An unspecified node indicates an unspecified reception. It is represented by the notation:

   (Pi+m)[UR]

   where i is the process number, m is the message type, and (Pi+m) represents the transition that initiates the unspecified reception. For example, the "(P1+1)[UR]" node shown in Figure A.3 represents that "process 1 receives message 1 leading to an unspecified reception".

2. Repeated nodes

   A repeated node is represented by the notation:

   (Pi+m)[ n]

   where n is the node number of the duplicated node, and (Pi+m) has the same meaning as in the unspecified node representation. Thus, the node "(P1-1)[ 2]" represents that "process 1 transmits message 1 resulting in a duplicated node having the same global state value as node 2".

3. Unbounded nodes

   An unbounded node appears when the number of messages in a channel exceeds a prespecified channel bound. It is represented by the notation:

   (Pi-m)[UC]

   where (Pi-m) has the same meaning as in the unspecified node representation. Thus, the node "(P1-2)[UC]" represents that "process 1 transmits message 2" resulting in the channel bound being exceeded".

4. Others

   Other than the above, all the nodes in the reachability tree are represented by sequentially ordered node numbers. The global state value of these nodes can be found in its corresponding entry within the node table, as will be described shortly.

**B. Representation of the node table**

It is assumed the user has a general understanding of the following terms: [4]

configuration $<S;T> = <s_1,....,s_N ;t_1,....,t_M>$

where N is the total number of processes and M is the total number of message types.

local state $s_i$ — is the current state of process i.

message counter $t_k$ — is the number of type-k messages.

As noticed from Figure A.2, the node table consists of all the nodes in the reachability tree. They are ordered according to their node numbers, from node 0 (root node) to the last node. Each node entry in the node table is composed of three parts:

<parent node number><transition><global state>

For example, node 1 in Figure A.2:

0(P1-1)1010000

represents the transition "process 1 transmits message 1" initiated at node 0 resulting in a global state 1010000.

The configuration of the global states is printed right above the node table. In this case, the configuration <s1,s2;t1,t2;t3,t4,t5> is used. To avoid confusions, the local states and message types are limited to a maximum number of 35. The numbers from 10 to 35 are printed as A to Z, respectively. Thus, a global state having s1=11 and s2=2 will be printed with s1,s2=B2 rather than s1,s2=112 which is ambiguous. The message type associated with each message counter is also printed as in Figure A.2.

When the R-state approach is used, there will be an extra set of state pointers — the r-state pointers — included in the configuration (i.e.<S;R;T>). Each message channel, in this case, is associated to a r-state pointer. A r-state pointer indicates the next message to be received at the corresponding channel by pointing to

---

[4] To avoid considerable overlappings, the terms mentioned are not explained in this manual. However, users are urged to refer to Chapter 3 of the thesis for complete explanations.

the state where the message was sent.

As a final note, if the symbol $\omega$ appears in a message counter, the message counter are detected to be unbounded and consequently the corresponding channel as well.

**C. The error summary**

The error summary is self-explanatory, yet attention is required to the stable-state table. If there exists a column in the stable-state table such that two entries containing the same state, then state ambiguity occurs. For example, the pair (0,0) and (0,3) in the stable-state table shown in Figure A.2 indicates that state 0 in process 1 can coexist stably with either state 0 or state 3 in process 2. The above observation can be extended to the muti-process case.

— THE END —