IMPLEMENTATION OF A PROTOCOL VALIDATION AND SYNTHESIS SYSTEM

By

DARREN PONG-CHOI TONG

B. Comp. Sci., University of Windsor, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT 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

May 1985

© Darren Pong-Choi Tong, 1985
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at 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
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date **April 11, 1985**
Abstract

VALISYN, an automated system for the validation and synthesis of error-free protocols has been implemented in C language. It assists designers in the detection and prevention of various kinds of potential design errors, such as state deadlocks, non-executable interactions, unspecified receptions and state ambiguities.

The technique employed is a stepwise application of a set of production rules which guarantee complete reception capability. These rules are implemented in a tracking algorithm, which prevents the formation of non-executable interactions and unspecified receptions, and which monitors the existence of state deadlocks and state ambiguities.

The implementation of VALISYN is discussed and a number of protocol validation and synthesis examples are presented to illustrate its use and features.
# Table of Contents

ABSTRACT ........................................................................................................................................... ii

TABLE OF FIGURES ........................................................................................................................... v

ACKNOWLEDGEMENTS ....................................................................................................................... vi

1. Introduction ........................................................................................................................................ 1
   1.1. Motivations and objectives ............................................................................................................. 1
   1.2. Thesis outline ................................................................................................................................. 2

2. Protocol Modelling and Potential Design Errors ........................................................................... 3
   2.1. Protocol Modelling ......................................................................................................................... 3
   2.2. Potential Design Errors ................................................................................................................... 5

3. The VALISYN — Algorithm Analysis and Implementation ............................................................ 8
   3.1. Production Rules ............................................................................................................................. 8
   3.2. Applying the Production Rules ....................................................................................................... 14
   3.3. Terminating the Synthesis Process .................................................................................................. 18
   3.4. Summary of the Algorithm ............................................................................................................. 20
   3.5. Handling of the Potential Design Errors ......................................................................................... 23
   3.6. Features of VALISYN .................................................................................................................... 24

4. Protocol Validation/Synthesis Examples ......................................................................................... 25
   4.1. Example 1. A general example ....................................................................................................... 25
   4.2. Example 2. Synthesis with unbounded channel ............................................................................. 32
   4.3. Example 3. Use of Erase Module ................................................................................................. 36
   4.4. Example 4. Detection of various kinds of potential design errors ............................................... 41
   4.5. Example 5. Independence of input order sequences .................................................................... 44

5. Conclusions ........................................................................................................................................ 47
   5.1. Thesis summary ............................................................................................................................. 47
5.2. Future work ........................................................................................................... 48

Bibliography ............................................................................................................... 49
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Simple access authorization protocol [ZWRC80].</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Example contains various Potential Design Errors.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Derivation of production Rule 1 [ZWRC80].</td>
<td>9</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Derivation of production Rule 2 [ZWRC80].</td>
<td>11</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Derivation of production Rule 3 [ZWRC80].</td>
<td>13</td>
</tr>
<tr>
<td>Figure 6</td>
<td>A synthesis design example.</td>
<td>15</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Flooring of the tree structure.</td>
<td>18</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Example with unbounded channels.</td>
<td>19</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Another example with unbounded channels.</td>
<td>20</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Removal of arc -3 from Figure 6.</td>
<td>23</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Example with unbounded channel (index not shown).</td>
<td>35</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Resulting diagram by connecting the arcs.</td>
<td>41</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank Dr. Son Vuong for his ideas and guidance, as well as for carefully reading this thesis.

I am also grateful to Dr. Sam Chanson for reading and commenting on the drafts.

Finally, I would also like to acknowledge Mr. H.Y. Wang for his contribution to designing some of the structures.
Chapter 1

Introduction

1.1. Motivations and objectives

With the growing trend towards increased reliability of computer networks capable of performing more sophisticated functions, communication protocols have grown enormously in complexity [ZWRC80, Goud84]. As a result, computer-automated tools for protocol validation and synthesis are in great demand.

Collectively, five types of formal frameworks for modelling communication protocols are available [Goud84]. They are the Finite State Machines [GoSh83, HoU179], Petri Nets (or vector Addition Systems) [BeTe82, KaMi69, Pete81], Communicating Finite State Machines [Boch78, Zafi78, BoSu80, BrZa83, GMYu83], Extended Communicating Finite State Machines [Boch80, Boch82], and Parallel Programs [Suns81].

The Communicating Finite State Machines model has successfully modelled some practical protocols, such as the Alternating-bit protocol [Boch78, GMYu83], the Binary Synchronous protocol [CGLa83], and others [Goud84].

We implemented a tool, VALISYN (VALIdator-SYNthesizer), using the Communicating Finite State Machines model and the technique developed by Zafiropulo et al. [ZWRC80], which allows the users to do protocol synthesis in an
interactive manner. In addition, it can also be used for protocol validation. In the latter application, the user can make a comparison between the protocol specification of the original copy and the error-free copy generated by the synthesis procedure.

1.2. Thesis outline

In this thesis, protocol validation and synthesis via resynthesis is discussed. This technique has been used to validate the X.75 Packet Level protocol [VuCo83b]. Before protocol validation and/or synthesis analysis can be done, a model must be set up. Chapter 2 will introduce the model employed and the existence of various kinds of potential design errors. Chapter 3 presents the algorithm and the implementation of VALISYN. The various features and usage of VALISYN are demonstrated by a number of examples described in Chapter 4. Finally, Chapter 5 concludes the thesis and outlines possible extensions that can be added to VALISYN.
Chapter 2

Protocol Modelling and Potential Design Errors

In this chapter, we discuss the model and some potential errors that may arise in designing a protocol.

2.1. Protocol Modelling

The Communicating Finite State Machines (CFSM) Model, used by Bochmann [Boch78], is employed to represent the interactions between two CFSMs and the protocol itself. In this model two Communicating Finite State Machines are used to model a network, from which messages are exchanged via two uni-directional channels. Figure 1 shows a simple example using this model [ZWRC80].

In the figure, it can be seen that process A and process B start at their initial states 0. Each message or event is represented by a directed arc or edge with a label: negative for event transmission and positive for event reception. For example, process A transmits the message ACCESS REQUEST to process B, which is represented by integer -1; while process B receives the message ACCESS REQUEST, which is represented by +1.

Each directed arc connects two states. The one that occurs before the event traversal is called the departure state; the other one that is reached after the event traversal is the entry state. In Figure 1, state 1 (departure state) of process B
transmits the message $-2$ (REFUSED ACCESS) and enters state 0 (entry state).

The state pair $(0, 0)$ is said to be a pair of stable states. A stable state pair occurs whenever the channels between the two processes are empty. Thus $(1, 1)$ and $(2, 2)$ are other examples of stable state pairs.

The CFSM model is established with the following assumptions:

(1) Each process will be initialized correctly (i.e. to its zero or reset states) prior to the start of any interaction.

(2) The channels are perfect. This means there will be no loss or distortion of messages. If a process transmits a message, then the other process will receive it.
(3) The channels are First_In_First_Out (FIFO). If process A sends message 1 followed by message 2, then process B will receive message 1 before it receives message 2.

(4) There is no time constraints such as transmission and/or response delays.

One advantage of employing this model is that it has an expressive power equivalent to that of Turing machines, thus all protocols can be modelled in this way [Goud84].

2.2. Potential Design Errors

With the above assumptions made for the CFSM model, four kinds of potential design errors can be detected, namely, state deadlocks, unspecified receptions, non-executable interactions and state ambiguities. Figure 2 shows an example with these four kinds of potential design errors.

1) State Deadlocks

When the states of a stable pair have no alternatives other than remaining indefinitely in their current states, a state deadlock occurs. In other words, each state of the stable state pair does not have any transmission arc. For example, this occurs when process A transmits a -1 and at the same time process B transmits a -3. As a result both process A and process B enter state 2. The stable state pair (2, 2) will remain indefinitely in these states, because the channels are empty and neither states have a transmission arc.
Figure 2. Example contains various Potential Design Errors.

State deadlocks in general represent errors, but there are some exceptions. For example, a protocol may be designed to terminate in states with no transmission arcs when a function is finished [ZWRC80]. This kind of potential error must be detected.

2) Unspecified Receptions

These occur when feasible receptions are not specified; in other words, positive reception arcs that can be traversed are missing. For example in Figure 2, if process B transmits an arc -3, then state 0 of process A should be able to receive the reception arc +3. However, it is not specified.

Unspecified receptions are harmful in protocol designs. When they happen, unknown states are entered and thus the interactions and behavior caused by the reception arcs are unpredictable.
3) Non-executable Interactions

Non-executable interactions are overspecifications of either transmission arcs or reception arcs. In Figure 2, no interaction sequences can cause state 3 of process B to receive event +2. Hence the reception arc +2 from state 3 to state 1 in process B will be a non-executable interaction. To view it in another way, the reception arc will be executable only if process A can transmit two -2 events without a -1 event in between. However, this is impossible.

Non-executable interactions may be harmful in designs, because the designers think that these arcs will be encountered under normal interaction operations.

4) State Ambiguities

State ambiguity occurs when there are two (or more) pairs of stable states \((x_1, y_1)\) and \((x_2, y_2)\), such that \(x_1 \neq x_2\) and \(y_1 = y_2\), or \(x_1 = x_2\) and \(y_1 \neq y_2\).

In Figure 2, as process A transmits a -1, it enters state 1; while in process B, it can receive the reception arc +1 and remain in state 0. Thus we have the stable state pairs (0, 0) and (1, 0), which produce state ambiguity.

State ambiguities may or may not result in errors. However, if the designer's intention is that state 0 of process A will only coexist with state 0 of process B, then it will be an error.

The CFSM model, together with an algorithm, which will be discussed in the next chapter, enable us to detect all the above potential design errors.
Chapter 3

The VALISYN — Algorithm Analysis and Implementation

The technique we employed was developed by Zafiropulo et al. [ZWRC80]. It is made up of two parts: the Three Production Rules and a Tracking Algorithm. Both of these are automated by the data-directed design programming method [CoLu78, CGWL80]. The algorithm, based on the cause-and-effect relationships mentioned in [ZWRC80], which prevents the formation of unspecified receptions and non-executable interactions and notifies the designer in case of state ambiguities or state deadlocks.

3.1. Production Rules

The three production rules are used to generate the reception arcs for two interacting processes. They are modified from an earlier incomplete version [Zafi78]. Their necessity and sufficiency are proved in [ZWRC80].

The first production rule considers the case when there is a transmission arc after a reception arc. In Figure 3(a), process P2 transmits the message $-e$ after the reception of message $+x$. If process P1 transmits the message $-z$ and stops, then its entry state should be able to receive the message $+e$. 
Figure 3. Derivation of production Rule 1 [ZWRC80].
However, if after the transmission of \(-z\) and before the reception of \(+e\), process P1 transmits \(-y\), then the entry state of \(-y\) should be able to receive the message \(+e\), which is marked as \(+e_y\). The reception arc has the subscript \(y\) to indicate that messages \(-y\) and \(-e\) collide. Collision occurs whenever two (or more) messages in different processes are being transmitted before neither is received. Figure 3(b) shows the generalization of rule 1, which is summarized as follows:

Rule 1: If \(-e\) is appended to \(+z\) then,

- a) append \(+e\) to \(-z\);
- b) append \(+e_y\) to every negative arc sequence \(-s\) attached to \(-z\) without parsing any positive arc.

Here (a) is for the generation of reception arcs without collision, whereas (b) is used for those that collide.

The second rule is for the case when we have a message transmission followed by another message transmission. In process P2 of Figure 4(a), we have the transmission arc \(-e\) attached to the transmission arc \(-z\). Thus in process P1, after the receiving of message \(+z\), it should be able to receive the message \(+e\). Moreover, if in process P1, it transmits message \(-y\) before the reception of any message, then message \(-y\) in process P1 will be in collision with messages \(-z\) and \(-e\) in process P2. Therefore, we have \(+z_y\) appended to \(-y\), and \(+e_y\) appended to \(+z_y\).

If message \(-z\) is transmitted after the reception of \(+z_y\) but before that of message \(-e\), then message \(-e\) will collide with messages \(-y\) and \(-z\) in process P1.
Figure 4. Derivation of production Rule 2 [ZWRC80].
Thus $+e_{y,z}$ is appended to $-z$. Similarly, if process P1 transmits $-z'$, then $+e_{z'}$ is appended to it. Figure 4(b) shows the generalized Rule 2, which is summarized as follows:

Rule 2: If $-e$ is appended to $-z$ then,

a) to every $+z$ and $+x$, append $+e$ and $+e_x$ respectively;

b) to every negative arc sequence $-s'$ attached to $+z$ or $+x$, append $e_s'$ and $+e_s'$ respectively.

note: $-s$ and $-s'$ are transmission sequences.

The third rule is needed when we have a transmission arc appended to a subscripted reception arc. In Figure 5(a), transmission arc $-e$ is appended to reception arc $+w_x$. This means that the messages $-w$ in P1 and $-x$ in P2 collide. Therefore process P1 is able to receive the message $+e$ only if it has transmitted the message $-w$ and received $+x$. The event $+e$ is not subscribed because it is not in collision with any arc.

However, if P1 transmits $-y$ before the reception of $+x$, then message $-z$ will collide with both $-w$ and $-y$ while $-e$ will collide with $-y$ only. Thus we have $+x_w,y$ appended to the entry state of $-y$ and $+e_y$ appended to $+x_w,y$. The formation of arc $+e_{y,z}$ is exactly the same as that of Figure 4(a).

The generalized Rule 3 is as follows:
Figure 5. Derivation of production Rule 3 [ZWRC80].
Rule 3: If $-e$ is appended to $+v$ in $P_2$, then within the tree with root $-v$ in $P_1$:

a) append $+e$ to $+u$ and $+e_s$ to every $+u,v,s$;

b) to every negative arc sequence $-s'$ attached to $+u$ or $+u,v,s$ append $+e_s'$ or $+e_{s,s'}$, respectively.

Note:
1) $-s$ and $-s'$ represent transmission sequences.
2) "..." stands for an arbitrary message sequence.
3) part (b) of Rule 3 uses the same mechanism as that of part (b) in Rule 2.

3.2. Applying the Production Rules

Due to the fact that the production rules are based on the cause-and-effect relationships, we need two preconditions for their applications. First, each negative arc has to be uniquely defined within a process; second, prior to the start of protocol synthesis, a fictitious event ($-n$, $+n$ events in Figure 6) must occur. Furthermore, the algorithm requires designer interventions to provide semantic information, such as the entry state of a reception event.

The state diagrams are represented internally as trees. After the exchange of the fictitious event, the algorithm will request for a design action. The designer enters $A,0\rightarrow 1=-1$, which means that there is a transmission event from departure state 0 to the entry state 1 in process A and the event is marked as $-1$. However, in
Figure 6. A synthesis design example.
the tree structure of process \( A \), it is shown as \( 0.0 \rightarrow 1.0 = -1.0 \), a decimal fraction has been added to each number. The reason for adding the fractional part is that we can only have one arrival (entry) arc for each state (node) in a tree structure and a state may be entered more than once. The decimal fraction is needed to identify each state.

In general, if state \( s \) transmits an event \(-e\), the algorithm will duplicate \(-e\) by attaching \(-e.i\) to \( s.i\) and it will apply the rules to each arc individually. Since there is only one copy of state 0 (0.0), so just one arc (−1.0) is created and its entry state is 1.0.

Now we have the case where a transmission event (−1.0) follows a reception event (+n), thus rule 1 is applied. From rule 1(a), the reception arc +1.0 is appended to −n. At this point, the designer will be prompted for an entry state, "\( ? \)", to complete the interaction \( B , 0 \rightarrow ? = 1 \). As the designer enters 0, the state 0.1 is created (which is the second copy of state 0). This ends the application of rule 1 since rule 1(b) cannot be applied.

The algorithm proceeds to prompt for the next interaction from the designer, \( A , 1 \rightarrow 2 = -2 \), which turns into \( A , 1.0 \rightarrow 2.0 = -2.0 \). This is the case of a transmission event (−2.0) following another transmission event (−1.0), and hence rule 2 can be applied. The algorithm forms the reception event \( B , 0.1 \rightarrow ? = +2.0 \) and the designer enters 0 for \( ? \) to complete the interaction. The 0 is represented as 0.2 internally (the third copy of state 0).

At this point the designer enters the third transmission event \( B , 0 \rightarrow 1 = -3 \). Since there are three copies of state 0, so three events, \( B , 0.0 \rightarrow 1.0 = -3.0 \),
$B, 0.1 \rightarrow 1.1 = -3.1,$ and $B, 0.2 \rightarrow 1.2 = -3.2$ are created. Rule 2 is applied to the first event (event $-3.0$), which produces the reception events $A, 0.0 \rightarrow 1.1 = +3.0,$ $A, 1.0 \rightarrow 2.1 = 3.0_{1.0},$ and $A, 2.0 \rightarrow 0.1 = 3.0_{0.2.0}.$ The last two event come from the application of rule 2(b).

It is worthy to note that the node 0.0 in process B has 2 departure arcs, $-3.0$ and $+1.0,$ and it should be possible for the state node 1.0 in process B to receive reception arc $+1,$ after the transmission of $-3.0.$ This can be done by reapplying rule 1 to arc $-1.0.$ But a simpler method called replication is employed. The replication procedure will copy the reception tree from the departure state of the transmission arc to its entry state, without parsing any negative arcs, and all the reception arcs are subscripted to indicate collision. Two examples are $+1.0_{3.0}$ and $+2.0_{3.0}$ in process B.

On the other hand, the state node 0.1 in process A should be able to transmit the event $-1.$ The algorithm will create the transmission event automatically and placed it in a future event queue. After applying the rules to arcs $-3.1$ and $-3.2,$ the arcs in the future event queue will be examined. For example, when rule 3 is applied to the transmission event $A, 0.2 \rightarrow 1.3 = -1.2,$ the reception event $B, 0.3 \rightarrow 0.5 = +1.2$ is created. However, the algorithm, and not designer interventions, automatically determines the entry state 0.5 because it is identical to the event $B, 0.0 \rightarrow 0.1 = 1.0.$

Two reception events $a.b_{i,j}$ and $c.d_{k,l},$ with departure states $e.f$ and $g.h,$ are identical if $e = g,$ $a = c,$ $i = k$ and $j = l$ [CRZa80]. Also the entry state 0.5 is regarded as a dead node, because the subtree obtained from this node will be a duplication from another node (the one it is identical to).
Another definition, *similar*, will be considered whenever the identical test fails. Two reception events $a.b_i.j$ and $c.d_k.l$, with departure states $e.f$ and $g.h$, are *similar* if $e = g$, $a = c$, and $i = k$. If the similar test also fails, then the algorithm will request the designer to enter this piece of semantic information. Figure 6 shows a complete tree of the synthesis.

### 3.3. Terminating the Synthesis Process

After the first interaction from the designer, the future event queue will always contain some transmission events, which implies the trees will grow indefinitely. Therefore, some way must be employed to halt the extra resynthesis. The technique we used is called *flooring* [ZWRC80], which collapses the tree structure by removing

---

![Figure 7. Flooring of the tree structure.](image)
the decimal fractions in all the arcs and state nodes. Figure 7 shows the flooring of Figure 6. If the new flooring result is the same as the old one, then the algorithm will stop the resynthesis and will request the designer to enter the next transmission event. However, if the two floorings are different, the algorithm will copy the transmission arcs from the future event queue to the current event queue, and it will apply the production rules to these transmission events.

Nevertheless, this works only if the channels between the processes are bounded. The proof can be found in [ZWRC80]. For the unbounded channels, see Figure 8. Process B can transmit the message event -3 after each reception event. Assume process B does this, and process A can transmit in a very high speed such that it always receives message +3 at state 2. That is, for every message +3 received, it transmits two messages. Thus the channel from process A to process B will grow indefinitely. Figure 9 shows a trivial example which contains unbounded

![Diagram](image-url)
channels, each of process A or B can transmit whenever it wants. If their transmitting speeds are sufficiently high, then they can easily flood the channels. In order to terminate syntheses with unbounded channels, a timer is used.

3.4. Summary of the Algorithm

This section attempts to describe the algorithm in detailed [CRZa80].

1. Set up the fictitious transmission event.
2. Determine the floor of each tree.
3. Input an event transmission

   \[ p, z \rightarrow w = -e \]

   where \( p \) is the process number, \( z \) the departure state, \( w \) the entry state and \(-e\) the label of the transmission arc.
4. Duplicate the message \(-e\) such that \(-e.i\) is appended to state \( z.i \), and place them in an event queue.
5. While the event queue is not empty, remove the top element and mark it as the current arc.

5.1. Determine which rule (1, 2, or 3) to apply to the current arc.

5.1.1. Apply the appropriate rule.

5.1.1.1. Generate reception arcs as long as it is possible to do so.

5.1.1.1.1. For each generated arc, determine its entry state $y_j$ and duplicate any negative arcs attached to the entry state $y_0$ and place it in a future event queue. The entry state of a reception arc may be determined by an identical arc, a similar arc, or requesting the designer to specify.

5.2. Replicate the reception tree attached to the departure state of the current arc.

6. Determine the floor of the trees.

7. If the floor is different from the floor value in step 2, then

7.1. Replace the event queue by the future event queue and mark the future event queue empty. Save floor value of the tree. Return to step 5.

8. Save floor values of the trees.

9. Request the designer for next interaction event or quit.
During the synthesis process, it is most likely that a designer may give a piece of wrong semantic information. Therefore, an Erase Module is needed to erase errors instead of performing the whole synthesis process again.

Erase Module [CRZa80]

1. While there is a transmission or one of its duplicates is to be erased,
   1.1. Find the transmission or the duplicate.
   1.2. Erase the transmission or its duplicate and the subtree attached to the entry state of the transmission arc (or duplicate).
   1.3. Find all receptions corresponding to the transmission (or duplicate) and erase each reception and all subtrees attached to their entry state.

2. Check each transmission in both trees to determine if it has any corresponding receptions, if there are no receptions, erase the transmission.

3. Check each reception in both trees to determine if it has any corresponding transmission, if there are no transmissions, erase the reception.

The erase module would remove the subtrees from the error point, as well as all the duplicated transmissions or receptions that would slow down the synthesis process. Figure 10 shows the resulting trees after the removal of transmission event -3 in Figure 6.
3.5. Handling of the Potential Design Errors

In the generation of the required reception arcs, since the production rules are complete [ZWRC80], so it is not possible to create non-executable interactions and unspecified receptions.

In addition, each time when we have the event pairs \((-e, +e), (+e_y, +y_e),\) or \((+e_{...y}, +y_{...e})\), their entry states \((i, j)\) is a stable state pair. For each stable state pair, if the algorithm cannot find any transmission event departing from them, it will report state deadlock. Furthermore, if two stable pairs \((i, j)\) and \((l, k)\) exist, either \(i = l\) or \(j = k\) will indicate state ambiguity.

Lastly, the algorithm is designed with the resynthesis technique; the final protocol design is independent of the order of the input transmission events. Hence with different event sequences, for example, \((-1, -2, -3)\) and \((-3, -1, -2)\), the designer would get the same final result.
3.6. Features of VALISYN

The package, VALISYN, is written in C language and runs on VAX\textsuperscript{\dagger} 11/750 under 4.2bsd UNIX\textsuperscript{*}. Two modes, interactive and non-interactive, are available for the validation and synthesis of communication protocols.

There is a similar package, written in APL, adopted from the University of Waterloo, Canada, now running on Amdahl 470/V8 under MTS (Michigan Terminal System).

In comparing these two packages, VALISYN is preferable over the other package because of the following features:

- VALISYN is written in C language rather than APL, so it is more portable.
- VALISYN has both interactive and non-interactive modes, while the package written in APL can only be used interactively.
- An erase procedure is available in VALISYN but not in the other package; this procedure is very helpful in protocol syntheses.

Apparently, VALISYN is slower than the other package by about 30\%. However, VALISYN runs on a minicomputer, VAX 11/750, while the other package runs on a large mainframe computer, Amdahl 470/V8. Thus the difference in execution time is not significant.

\textsuperscript{\dagger}VAX is a trademark of Digital Equipment Corporation.
\textsuperscript{*}UNIX is a trademark of AT&T Bell Laboratories.
Chapter 4

Protocol Validation/Synthesis Examples

This chapter will illustrate the use of VALISYN by means of examples.

The package can be used both interactively or non-interactively. For interactive applications, the package requires the designer to enter semantic information when necessary. An Erase Module is also available to erase errors. In non-interactive applications, the designer initially enters all the transmission and reception arcs of the two state diagrams, then the algorithm will produce the final result and any potential design errors.

Examples 1 to 3 demonstrate interactive applications of the package, while the other two illustrate non-interactive usages.

4.1. Example 1. A general example.

The following is a sample session to illustrate the use of VALISYN, as well as the kind and meaning of information obtained from it.

```
package

Debugging information? [y\n] y

Level 2 debugging information? [y\n] y
```
Interactive or Non-interactive? [i|n] i

Enter transmission event
1,0→1=1

1 0.0 -1.0 1.0

Rule 1 is called
2, 0 → ? = 1 enter ? OR h -- HELP 0
2 0.0 1.0 0.1

Rule 1B is called
Replicate is called

Display stable pairs? [y|n] y

Stable States

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Enter transmission event
1,1→2=2

1 1.0 -2.0 2.0

Rule 2 is called
2, 0 → ? = 2 enter ? OR h -- HELP 0
2 0.1 2.0 0.2

Rule 2B is called
Replicate is called

Display stable pairs? [y|n] y

Stable States

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
2 1

Enter transmission event
2,0→1=3

2 0.0 -3.0 1.0

Rule 2 is called
1, 0→?=3 enter ? OR h -- HELP
1
1 0.0 3.0 1.1

Rule 2B is called
1, 1→?=3(1) enter ? OR h -- HELP
2
1 1.0 3.0 2.1 1.0

Rule 2B is called
Replicate is called
2, 1→?=1(3) enter ? OR h -- HELP
2
2 1.0 1.0 2.0 3.0

Replicate is called
2, 2→?=2(3) enter ? OR h -- HELP
3
2 2.0 2.0 3.0 3.0

Replicate is called
2 0.1 -3.1 1.1

1, 1→?=3 enter ? OR h -- HELP
2
1 1.0 3.1 2.2

Rule 1B is called
Similar -- 1 2.0 3.1 0.2 2.0

Rule 1B is called
Replicate is called
2, 1→?=2(3) enter ? OR h -- HELP

Example 1 continue
0
2 1.1 2.0 0.3 3.1

Replicate is called

2 0.2 -3.2 1.2

Rule 1 is called
1, 2 → ? = 3 enter ? OR h -- HELP
0
1 2.0 3.2 0.3

Rule 1B is called
Replicate is called

1 1.1 -2.1 2.3

Rule 1 is called
2, 1 → ? = 2 enter ? OR h -- HELP
0
2 1.0 2.1 0.4

Rule 1B is called
Replicate is called

1 0.1 -1.1 1.2

Rule 3 is called
2, 3 → ? = 1 enter ? OR h -- HELP
2
2 3.0 1.1 2.1

Rule 3B is called
Replicate is called

1 0.2 -1.2 1.3

Rule 3 is called
Identical -- 2 0.3 1.2 0.5

Rule 3B is called
Replicate is called

2 0.3 -3.3 1.3

Rule 3 is called
Identical -- 1 0.2 3.3 1.4

Example 1 continue
Rule 3B is called
Similar -- 1 1.9 3.3 2.4 1.2

Rule 3B is called
Replicate is called
Similar -- 2 1.9 1.2 2.2 3.3

Replicate is called
1 0.3 -1.3 1.5

Rule 1 is called
2, 1 → ? = 1 enter ? OR h -- HELP
2 2 1.2 1.3 2.3

Rule 1B is called
Replicate is called
2 0.4 -3.4 1.4

Rule 1 is called
Identical -- 1 2.9 3.4 0.4

Rule 1B is called
Replicate is called
1 1.2 -2.2 2.5

Rule 2 is called
2, 2 → ? = 2 enter ? OR h -- HELP
3 2 2.1 2.2 3.1

Rule 2B is called
Replicate is called
1 1.5 -2.9 2.6

Rule 2 is called
Identical -- 2 2.9 2.9 3.2

Rule 2B is called
Replicate is called

Display stable pairs? [y/n] y

Example 1 continue
Stable States

Process 1          Process 2
  0   1   2   3
  0   1   1   0   1
  1   1   1   0
  2   1   1   1

Enter transmission event
end

Process 1 has the following transmission arcs
  0 → 1 = 1
  1 → 2 = 2

Process 1 has the following reception arcs
  2 → 0 = 3
  0 → 1 = 3
  1 → 2 = 3

Process 1 has the following reception arcs with collision
  2 → 0 = 3(2)
  1 → 2 = 3(1)

Process 2 has the following transmission arcs
  0 → 1 = 3

Process 2 has the following reception arcs
  0 → 0 = 1
  0 → 0 = 2
  1 → 0 = 2
  1 → 2 = 1
  3 → 2 = 1
  2 → 3 = 2

Process 2 has the following reception arcs with collision
  1 → 0 = 2(3)
  1 → 2 = 1(3)
  2 → 3 = 2(3)

*** Protocol Error --- Deadlock states
(2, 1)   (2, 2)   (2, 3)

Total CPU time : 1560000 sec.
Total System Call : 1200000 sec.

Example 1 continue
This example tries to show the details of how the trees are built. The following points are worth noting:

- The "debugging information" tells the designer the rule or replication being applied.

- The "level 2 debugging information" will print out each arc created. For example, 
  (2 1.1 2.0 0.3 3.1) means that in tree 2, the reception arc +2.0 with subscript 3.1, which departs from state node 1.1 to entry state node 0.3, is created.

- The transmission event is entered as 1,0→1==1 rather than 1,0→1==−1, because it is only necessary for the designer to enter the transmission arcs, and the reception arcs will be generated automatically.

- When the algorithm requests the designer to enter the entry state, e.g. 1, 1 → ? == 3(1), the "1" in parentheses is the subscript of the reception arc.

- The stable states is a two-dimensional matrix, a "1" in element (i, j) implies state[row(i)] and state[column(j)] are a pair of stable states. It is useful to show the state ambiguities in the design.

- "Similar" indicates the reception arc is similar to another one within the tree, and the state is decided automatically.

- "Identical" indicates the reception arc is identical to another arc, which means a dead node is formed.

- The resulting state diagrams for this example can be obtained by connecting all the arcs (which is the same as Figure 7).
4.2. Example 2. Synthesis with unbounded channels.

This sample session shows how VALISYN reacts when dealing with unbounded channels.

package

Debugging information? [y/n]
  n

Level 2 debugging information? [y/n]
  n

Interactive or Non-interactive? [i/n]
  i

Enter transmission event
  1,0→0=1

  2, 0 → ? = 1  enter ? OR h -- HELP

  0

Display stable pairs? [y/n]
  n

Enter transmission event
  1,0→0=2

  2, 0 → ? = 2  enter ? OR h -- HELP

  0

Display stable pairs? [y/n]
  n

Enter transmission event
  2,0→0=1

  1, 0 → ? = 1  enter ? OR h -- HELP

  0

  1, 0 → ? = 1(1)  enter ? OR h -- HELP

  0

  1, 0 → ? = 1(2)  enter ? OR h -- HELP

  0
Display stable pairs?  [y|n]  
n
Enter transmission event
2,0→0=2

1, 0 → ? = 2  enter ? OR h -- HELP
0

1, 0 → ? = 2(1)  enter ? OR h -- HELP
0

1, 0 → ? = 2(2)  enter ? OR h -- HELP
0

Total CPU time : 13 90000 sec.
Total System Call : 1 460000 sec.

*** Warning -- the protocol may be unbounded

Display stable pairs?  [y|n]  
y

Stable States

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Flooring of arcs?  [y|n]  
y

Process 1 has the following transmission arcs
0 0 = 1
0 0 = 2

Process 1 has the following reception arcs
0 0 = 1
0 0 = 2

Process 1 has the following reception arcs with collision
0 0 = 1(1)
0 0 = 1(2)
0 0 = 2(1)
0 0 = 2(2)

Example 2 continue
Process 2 has the following transmission arcs
0 → 0 = 1
0 → 0 = 2

Process 2 has the following reception arcs
0 → 0 = 1
0 → 0 = 2

Process 2 has the following reception arcs with collision
0 → 0 = 1(1)
0 → 0 = 1(2)
0 → 0 = 2(1)
0 → 0 = 2(2)

Carry on with the synthesis? [y/n] y

Total CPU time : 30 340000 sec.
Total System Call : 2 400000 sec.

*** Warning -- the protocol may be unbounded

Display stable pairs? [y/n] y

Stable States

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Flooring of arcs? [y/n] y

Process 1 has the following transmission arcs
0 → 0 = 1
0 → 0 = 2

Process 1 has the following reception arcs
0 → 0 = 1
0 → 0 = 2

Process 1 has the following reception arcs with collision
0 → 0 = 1(1)
0 → 0 = 1(2)
0 → 0 = 2(1)

Example 2 continue
$0 \rightarrow 0 = 2(2)$

*Process* 2 has the following transmission arcs

- $0 \rightarrow 0 = 1$
- $0 \rightarrow 0 = 2$

*Process* 2 has the following reception arcs

- $0 \rightarrow 0 = 1$
- $0 \rightarrow 0 = 2$

*Process* 2 has the following reception arcs with collision

- $0 \rightarrow 0 = 1(1)$
- $0 \rightarrow 0 = 1(2)$
- $0 \rightarrow 0 = 2(1)$
- $0 \rightarrow 0 = 2(2)$

Carry on with the synthesis? [y/n]

n

Total CPU time : 36 890000 sec.
Total System Call : 2 880000 sec.

In Figure 11, each state has two self-transmitting loops, thus the channels can be easily flooded. When the timer fires, the following will occur:

- give a warning to the designer for unbounded channels.

\[ 
\begin{array}{c}\text{PROCESS A} \\ \begin{array}{c} \text{0} \\ \pm 1 \\ \pm 2 \end{array} \end{array} \quad \begin{array}{c}\text{PROCESS B} \\ \begin{array}{c} \text{0} \\ \pm 1 \\ \pm 2 \end{array} \end{array} \]

Figure 11. Example with unbounded channels (index not shown).
- output the stable states.
- output all the transmission and reception arcs.
- ask the designer whether to continue or not.

4.3. Example 3. Use of Erase Module.

This sample session illustrate the application of the Erase Module of VALISYN.

```
package

Debugging information? [y\n]
  n

Level 2 debugging information? [y\n]
  n

Interactive or Non-interactive? [i\n]
  i

Enter transmission event
  1,0\rightarrow 1\Rightarrow 1

  2, 0 \rightarrow ? = 1 enter ? OR h -- HELP
  1

Display stable pairs? [y\n]
  y

Stable States

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Enter transmission event
  2,1\rightarrow 2\Rightarrow 3
```
1, 1 → ? = 3  enter ? OR h -- HELP
2

Display stable pairs? [y\|n]
y

Stable States

\begin{array}{ll}
\text{Process 1} & \text{Process 2} \\
0 & 1 2 \\
0 & 1 0 0 \\
1 & 0 1 0 \\
2 & 0 0 1 \\
\end{array}

Enter transmission event
2,0→3=4

1, 2 → ? = 4  enter ? OR h -- HELP
3

Display stable pairs? [y\|n]
y

Stable States

\begin{array}{ll}
\text{Process 1} & \text{Process 2} \\
0 & 1 2 3 \\
0 & 1 0 0 0 \\
1 & 0 1 0 0 \\
2 & 0 0 1 0 \\
3 & 0 0 0 1 \\
\end{array}

Enter transmission event
erase

Enter the transmission arc to be ERASED
2,1→2=3

Do you really want to ERASE the arc? [y\|n]
y

Trans. arc 2, 1 → 2 = 3 has been removed

Display stable pairs? [y\|n]
y

Stable States
Process 1    Process 2
    0 1
    0 1 0
    1 0 1

Flooring of arcs? [y/n]  
y

Process 1 has the following transmission arcs  
0 → 1 = 1

Process 1 has the following reception arcs

Process 1 has the following reception arcs with collision

Process 2 has the following transmission arcs

Process 2 has the following reception arcs  
0 → 1 = 1

Process 2 has the following reception arcs with collision

Enter transmission event  
2,1→2→3

1, 1 → ? = 3  enter ? OR h -- HELP
2

Display stable pairs?  [y/n]  
n

Enter transmission event  
2,2→3→4

1, 2 → ? = 4  enter ? OR h -- HELP
3

Display stable pairs?  [y/n]  
n

Enter transmission event  
2,1→0→2

1, 1 → ? = 2  enter ? OR h -- HELP
0

Example 3 continue
Display stable pairs? [y/n] 
n
Enter transmission event 
1,3→1=6

2, 3 → ? = 6  enter ? OR h -- HELP 
1

Display stable pairs? [y/n] 
n
Enter transmission event 
1,2→0=5

2, 2 → ? = 5  enter ? OR h -- HELP 
0

2, 3 → ? = 5(4)  enter ? OR h -- HELP 
0

1, 0 → ? = 4(5)  enter ? OR h -- HELP 
0

2, 0 → ? = 1(4)  enter ? OR h -- HELP 
1

1, 1 → ? = 4(1)  enter ? OR h -- HELP 
1

Display stable pairs? [y/n] 
y

Stable States

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 0 0 0</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>0 1 0 0 0</td>
</tr>
<tr>
<td>2 0 0 1 0</td>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>3 0 0 0 1</td>
<td>0 0 0 1 0</td>
</tr>
</tbody>
</table>

Enter transmission event 
end

Process 1 has the following transmission arcs 
2 → 0 = 5
0 → 1 = 1
3 → 1 = 6

Process 1 has the following reception arcs
1 → 0 = 2
1 → 2 = 3
2 → 3 = 4

Process 1 has the following reception arcs with collision
0 → 0 = 4(5)
1 → 1 = 4(1)

Process 2 has the following transmission arcs
1 → 0 = 2
1 → 2 = 3
2 → 3 = 4

Process 2 has the following reception arcs
2 → 0 = 5
0 → 1 = 1
3 → 1 = 6

Process 2 has the following reception arcs with collision
3 → 0 = 5(4)
0 → 1 = 1(4)

Total CPU time : 2 140000 sec.
Total System Call : 0 980000 sec.
The Erase Module will erase the whole subtree from the error point. This example shows that when transmission event 2 is erased after the input sequence (1, 2, 3), event 3 is also removed. Figure 12 shows the final state diagrams.

4.4. Example 4. Detection of various kinds of potential design errors.

In the sample session, VALISYN is used as a validator to detect various kinds of potential errors.

```
package

Debugging information? [y/n]
y
Level 2 debugging information? [y/n]

Interactive or Non-interactive? [i/n]

Have default CPU limit 50 sec.? [y/n]
```
Format for inputting the arc

E.g. Trans. arc -- 1 2 3
i.e. from Dept. state 1 to Entry state 2 with Trans. arc 3
Note -- the FIRST Trans. arc in Process 1 will be executed first

Recept. arc -- 1 2 3 4
i.e. from Dept. state 1 to Entry state 2 with Recept. arc 3
and collision arc 4
Enter a ZERO for collision arc if no or unknown collision occurs

Enter the Trans. Arcs for Process 1 -- end with a NULL line
0 1 1
1 2 2

Enter the Recept. Arcs for Process 1 -- end with a NULL line
1 2 3 0
2 0 3 0

Enter the Trans. Arcs for Process 2 -- end with a NULL line
0 1 3

Enter the Recept. Arcs for Process 2 -- end with a NULL line
0 0 1 0
0 0 2 0
1 0 2 0
1 2 1 0
2 3 2 0
3 2 1 0
3 1 2 0

Stable States

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td></td>
</tr>
<tr>
<td>2 1 1 1</td>
<td></td>
</tr>
<tr>
<td>3 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Process 1 has the following transmission arcs
0 \rightarrow 1 = 1
1 \rightarrow 2 = 2

Example 4 continue
Process 1 has the following reception arcs
2 → 0 = 3
1 → 2 = 3
0 → 3 = 3

Process 1 has the following reception arcs with collision
2 → 0 = 3(2)
1 → 2 = 3(1)

Process 2 has the following transmission arcs
0 → 1 = 3

Process 2 has the following reception arcs
0 → 0 = 1
0 → 0 = 2
1 → 2 = 1
3 → 2 = 1
2 → 3 = 2

Process 2 has the following reception arcs with collision
1 → 0 = 2(3)
1 → 2 = 1(3)
2 → 3 = 2(3)

*** Warning -- the following specified arcs are not executable
2, 3 → 1 = 2

*** Error -- the following Recept. arcs are not specified
1, 0 → 3 = 3

*** PROTOCOL ERROR --- Deadlock states

Total CPU time : 1 80000 sec.
Total System Call : 0 630000 sec.

In this example, the package is used non-interactively, and the transmission and reception arcs of Figure 2 are entered as input data.
The final output indicates that there are non-executable arc, unspecified reception, state deadlock and state ambiguities (from the stable state matrix). All these serve to illustrate the potential design errors described in Chapter 2.

4.5. Example 5. Independence of input order sequences.

With different input order sequences, VALISYN can produce the same final result.

```
package

Debugging information? [y/n]

n

Level 2 debugging information? [y/n]

n

Interactive or Non-interactive? [i/n]

n

Have default CPU limit 50 sec.? [y/n]

y

Format for inputting the arc

E.g. Trans. arc -- 1 2 3
    i.e. from Dept. state 1 to Entry state 2 with Trans. arc 3
    Note -- the FIRST Trans. arc in Process 1 will be executed first

    Recept. arc -- 1 2 3 4
    i.e. from Dept. state 1 to Entry state 2 with Recept. arc 3 and collision arc 4
    Enter a ZERO for collision arc if no collision occurs

Enter the Trans. Arcs for Process 1 -- end with a NULL line
0 1 1
2 0 5
3 1 6
```
Enter the Recept. Arcs for Process 1 -- end with a NULL line
0 0 4 0
1 1 4 0
1 0 2 0
1 2 3 0
2 3 4 0

Enter the Trans. Arcs for Process 2 -- end with a NULL line
1 0 2
1 2 3
2 3 4

Enter the Recept. Arcs for Process 2 -- end with a NULL line
0 1 1 0
2 0 5 0
3 1 6 0
3 0 5 0

Stable States

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>2 0 1 0</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>3 0 0 1</td>
<td>0 0 0 1</td>
</tr>
</tbody>
</table>

Process 1 has the following transmission arcs
2 → 0 = 5
0 → 1 = 1
3 → 1 = 6

Process 1 has the following reception arcs
1 → 0 = 2
1 → 2 = 3
2 → 3 = 4

Process 1 has the following reception arcs with collision
0 → 0 = 4(5)
1 → 1 = 4(1)

Process 2 has the following transmission arcs
1 → 0 = 2
1 → 2 = 3
2 → 3 = 4

*Process 2 has the following reception arcs*
2 → 0 = 5
0 → 1 = 1
3 → 1 = 6

*Process 2 has the following reception arcs with collision*
3 → 0 = 5(4)
0 → 1 = 1(4)

*Total CPU time* : 1 260000 sec.
*Total System Call* : 0 680000 sec.

This last example uses Figure 12 in Example 3 as input. In spite of different input order sequences, (1, 3, 4, 2, 6, 5) in Example 3 and (1, 2, 3, 5, 4, 6) in this example, the final result is the same as that of Example 3.
Chapter 5

Conclusions

5.1. Thesis summary

A package for protocol validation/synthesis has been implemented in C language. The tool performs protocol validation by resynthesizing error-free versions of the given protocols. The algorithm, based on a stepwise application of a set of production rules, is able to detect various kinds of potential design errors, such as state deadlocks, non-executable interactions, unspecified receptions and state ambiguities.

However, when the algorithm is applied to protocols with unbounded channels, especially self-loop transmissions, a limitation arises, namely, too many dead nodes are being generated. This limitation can be corrected by altering the design, such as setting a bound on the level of resynthesis, or employing a timer. But both of these can at times prematurely terminate the generation of reception events. This problem becomes more serious when the protocol is large and complex. Some decomposition schemes would be helpful in breaking the large protocol into smaller, more manageable components [VuCo82a].

Despite this limitation, the tool has proven useful in validating and synthesizing various practical protocols, such as X.75 [VuCo83b].
5.2. Future work

The algorithm can be extended to cope with non-ideal channels, priority channels, and interactions among N-processes. In addition, it would be desirable to have the ability to save the work and resume it some time later, but this feature may add an excessive amount of space overhead.

Finally, it is worth noting that the algorithm applied in VALISYN is essentially equivalent to the reachability analysis method [VuCo82b, VuCo83a]. A similar tool has been developed, called VALIRA, which can handle the same kinds of errors [Hui85]. The VALIRA can handle N-process protocols, non-FIFO and priority channels as well as the FIFO channels. Nevertheless, VALISYN can be used as a synthesis tool and it gives the designers more insight into the errors. Both tools, VALISYN and VALIRA, can be used in a complementary manner.
Bibliography


