

Notes About Automatic Generation for Argument Retrieval in Asynchronous Communicating STS

DRAFT

Jean-Claude Royer

OBASCO Group, EMN - INRIA
École des Mines de Nantes, 4, rue Alfred Kastler - BP 20722, F-44307 Nantes Cedex 3
Jean-Claude.Royer@emn.fr

Abstract. In our previous work, we defined an approach for specifying components and architectures using a combination of symbolic transition systems, data types, and synchronous communications. We extend this approach with asynchronous communications. In this setting, specifications and proofs are complicated by the presence of buffers and the fact that the receipt time of a message is distinct from its execution time. This complicates the specifications and also the proofs. One approach is to specialise the behaviour of components, this is possible as soon as we have some information for example about the content of mailboxes. This note explains how to generate axioms related to asynchronous messages from this specialised behaviour.

KEYWORDS: Asynchronous Communication, Component, Architecture, Dynamic Behaviour, Unbounded or Bounded Mailbox, PVS, Specification Simplification.

1 Introduction

This short note is related to the simplification process of a component specification with asynchronous communications. The context is defined in [4]. The problem is given a S state how to retrieve the (`arg`) argument of a message. The process does not depend from a given argument, it only depends of the message, which is noted `msg`. Without specialising the STS we have to search the message in the history (trace) of the dynamic behaviour. This requires to define two auxiliary and recursive functions on the STS. An alternative way is to explicit a mailbox data structure, which is potentially infinite. The specialisation of the STS offers the opportunity to define argument retrieval with neither auxiliary function nor additional data structure. The function has to be define as recursive only if there are cycles without `msg` or in case of composed loops. In most of the case it expresses a kind of direct access to the information, direct here meaning statically known from the STS analysis. However in the general case we have to dynamically check the exact path to choose the right message. Without the specialisation we have to traverse the path for finding the message. The specialisation allows us to generate a set of cases for which the position of the right message is statically known. To choose the case for a path needs only to know a sub part of the path. This axiom generation is possible since the specialised STS expressed only loops with an equal number of receipt and execution for a given message name.

We generate a set of axioms which is expected without critical pairs, since we consider conditions we have to study feasible critical pairs [2]. Two distinct axioms have no feasible critical pairs since they have either a different starting state or the paths represented in them are not equal.

2 Examples

We have for example the component behavioural description of Figure 1. A mailbox

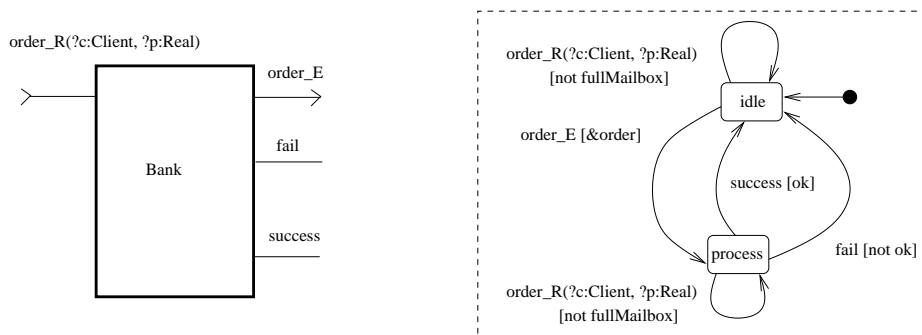


Fig. 1. The Bank Component with Asynchronous Communications

analysis of the global system has shown that: the `idle` state for `Bank` is reached with an empty buffer (\perp) or with an `order` message. From these constraints, a specialise algorithm computes a new STS for the subcomponent according to these constraints. The result for the bank component is described in Fig. 2. It expresses a simulation, in the dynamic sense [1], of the original bank dynamic behaviour.

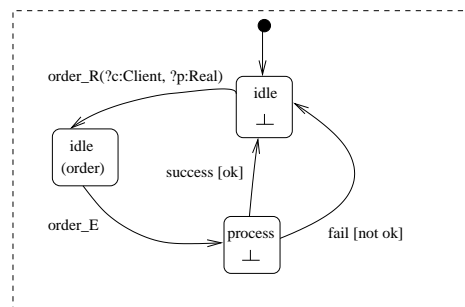


Fig. 2. A Specialised Bank Dynamic Behaviour

For the same flight reservation system the company example and its specialisation are described in Figure 3.

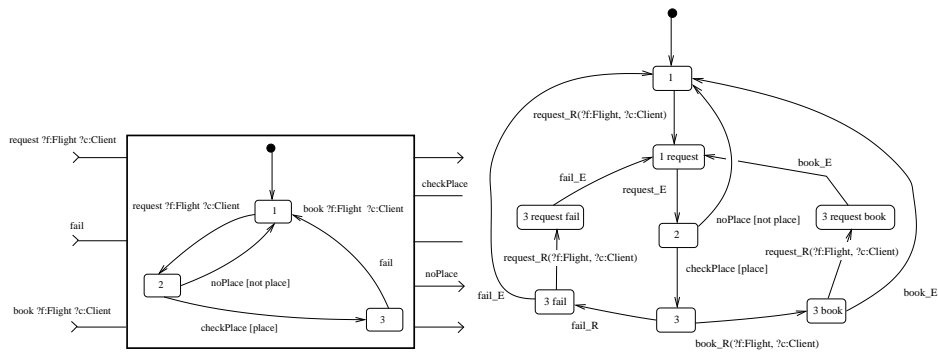


Fig. 3. The Company Example

A bus controller may be analysed as in [3] and the STS for the unit and its specialisation are depicted in Figure 4.

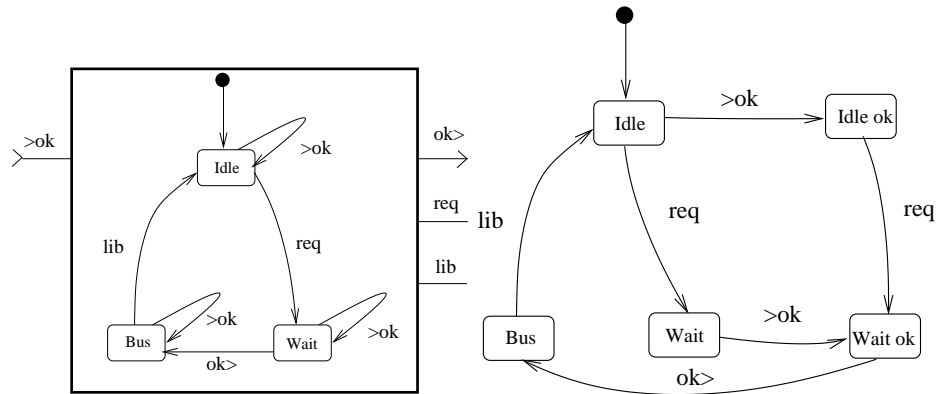


Fig. 4. The Unit Example

3 Properties

Property 3.1 *By construction, the following properties hold in such a specialised STS:*

1. *Let a trace reaching a S state:*
 - (a) *At each step the number of op_E occurrences is lesser or equal than number of op_R occurrences.*
 - (b) *Let k the number of op_E occurrences preceding S , the corresponding op_R has rank k .*

2. In each state the content of the mailbox buffer is statically known.
3. For each loop on the specialised STS and for all messages, there is a number of receipts equal to the number of executions.

The first item is a general property for traces of the STS with FIFO mailboxes. The second item says that operations acting on the mailbox buffer return constants. For instance `[not fullMailbox]` is always equal to `true`, or the `[\&op]` guard may be reduced to `true` or `false`, and so on. For the last item, if the number of receipts and the number of executions are distinct in a loop (for a given message), the source state of the cycle has a mailbox content strictly different from the target state. This is not possible with the specialised STS by construction since a state explicits the mailbox content and related guards are reduced to `true` or `false`.

4 Axiom Generation for Argument Retrieval

The problem is given a S state how to retrieve the (`arg`) argument of a previously received message? The process does not depend from a given argument, it only depends of the message, which is noted `msg`. The principle is to start from S and to follow the transitions in reverse direction until the initial state or a loop is detected. A path in an STS may be easily translated into conditional axioms thus we study paths in the specialised STS. Paths reduced to operation labels may be described by rational expressions. Let us study some simple examples to understand the generating process.

4.1 A Simple Path

There is a simple path without loop from I to S . Let k the number of `msg_E` in the path, if $k = 0$ then this message is not known else the corresponding `msg_R` is the k^{th} from the initial state. This case is graphically represented in Figure 5. We assume there is only one `arg` in this message denoted by the X term. A simple axiom may be generated:

$$P_E(\text{self}) \wedge \text{cond} \Rightarrow \text{arg}(f(\dots \text{msg}_R(\text{self}, X)\dots)) = X.$$

Where E is the starting state of the corresponding `msg_R`, P_E the state predicate, $f(\dots \text{msg}_R(\text{self}, *)\dots)$ the call associated to the path from E to S and `cond` the conjunction of guards on this path.

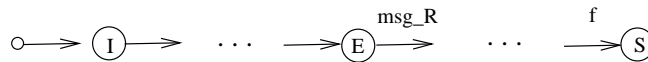


Fig. 5. A Simple Path

The simplest situation, and also the most frequent, is a `msg_R` immediately followed by its execution. In the three examples of Section 2 this case occurs.

4.2 A Path with Loop

A more general case is to consider that there is a loop on the reverse path until the initial state. The case of a loop with I and S is not considered as a loop here since we assume that the buffer of the initial state is always empty. The Figure 6 illustrates this general situation. It is general since every case with a loop may be transformed into the Figure 6 choosing the prefix, suffix and duplicating some states. In this figure *prefix* is the sequence from the initial state to the loop and similarly for *suffix*. In the sequel prefix, loop, suffix are viewed as sequences of operation calls. The pair (rp, ep) are the number of `msg_R` and the number of `msg_E` in the prefix. There is a similar notation for the suffix part. The weight w of a loop is its number of `msg_E`. We have the following numerical relations: $ep \leq rp$ and $(es + ep) \leq (rs + rp)$. A path has the general form

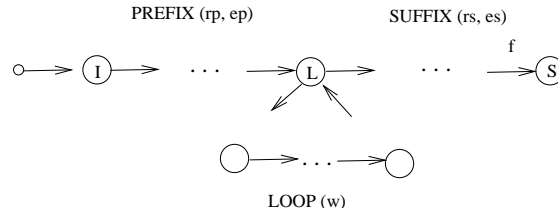


Fig. 6. The Loop Path

`prefix loop* suffix`. To retrieve the right message depends on the number of loop in the path and on the es, ep, rs, rp and w information. However we observe that there are several axioms (at least one) for the basic case and one axiom catching the general looping case. The principle is to unfold the loop a sufficient number of times.

Case: $w = 0$. The loop has neither `msg` receipt nor `msg` execution.

- If $rp - ep = 0$ then the receipt message will be found in the suffix part. We generate an axiom in the same manner as with the simple path in Section 4.1.
- If $rp - ep > 0$ and $es > (rp - ep)$ then the corresponding message will also be found in the suffix part, the same process applies.
- The last sub-case is $rp - ep > 0$ and $es \leq (rp - ep)$, the message is found in the prefix part. The axiom generation is more complex since we have to cope with the loop iterations. We generate a first axiom as in the simple path case. To handle the loop we write an axiom like:


```
P_L(self) ∧ cond =>
arg(loop(suffix(self))) = arg(suffix(self))
```

 which “removes” the loop.

Case: $w > 0$. In this case the loop may have a strict effect on the searching process.

- If $rp - ep = 0$ and $es = 0$ then the corresponding message will be found in the loop part. We generate the following axiom:

- $PL(\text{self}) \wedge \text{cond} \Rightarrow \text{arg}(\text{loop}(\text{suffix}(\text{self}))) = X$,
 where X is the required argument which has to be found in the loop part.
- If $rp - ep = 0$ and $es > 0$ then the corresponding message will be found in the suffix part. We generate an axiom in the same manner as with a simple path.
 - If $rp - ep > 0$, let $n = (rp - ep) \text{ div } w$, $n + 1$ is the number of axioms we need to generate. For a trace this number represents the minimal number of loops to process such that the oldest msg_R message occurs in the loop. The idea is to unfold $\{0, 1, \dots, n - 1\}$ times the loop and to generate axioms according to the simple path principle. The $n + 1^{\text{th}}$ axiom is like that: $PL(\text{self}) \wedge \text{cond} \Rightarrow \text{arg}(\text{loop}^n(\text{suffix}(\text{self}))) = X$ where X is the required argument. We argue that the receipt message appears in the $\text{loop}^n(\text{suffix}(\text{self}))$ since the n loops consume the messages coming from the prefix. If we loop one more the right message moves one loop right since there is an equal number of receipts and executions in a loop. Note that in the loop we have to find the oldest receipt which is known from the analysis of the prefix (see Fig. 9 for an example). Messages preceding the oldest in the loop are not taken into account to compute the right receipt. It needs to analyse the path compound from the n loops, the oldest receipt and the suffix.

The case where the S state occurs in the loop may be viewed as a particular case and it is possible to simplify the generation. We avoid the special cases here concentrating our process on a general process.

4.3 General Case

In the previous section we have studied the case of a path with one loop. In the general case we may have several distinct paths which are processed separately. To propose a general process we have also to describe two additional constructions: the case where loops are sequential or composed.

Loop Sequence. The sequential composition of loops is illustrated in Figure 7. The generation principle is to transform this case into two cases with a simple loop. Let $N1 = (rp - ep) \text{ div } w1$ we generate $N1$ paths with $n \in \{0..N1 - 1\}$ unfolding of the first loop. We get $N1$ new simple paths with one loop and we apply the loop case of Section 4.2. We consider a path with at least $N1$ loops of the first loop. The $L1$ state may be taken as initial state. The oldest msg message is known from the analysis of the prefix and its position will be constant if the number of the first loop is greater or equal to $N1$. We apply the analysis for a simple path with loop, but starting with $L1$ as initial state, with a prefix equal to $N1$ first loop plus MID and using the oldest receipt.

Composed Loops. A loop combination is any sequence of the two loops according to the finite state machine. Let the set of loop combinations which have a weight greater or equal than $rp - ep$ and consider cmb as the set of minimal elements to the weight. There are two cases:

1. For each combination of the loops such that its weight is lesser than $rp - ep$ we consider the associated simple path and we use the axiom generation of 4.1.

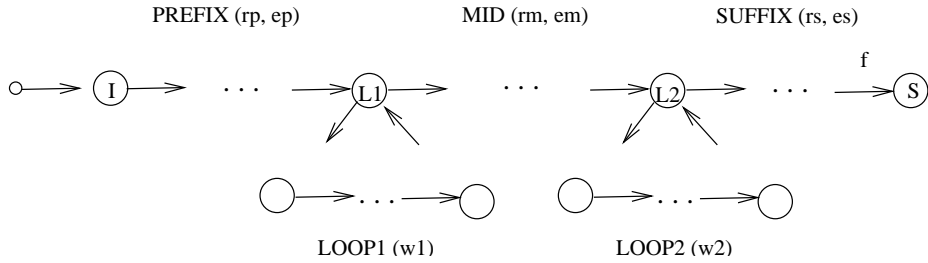


Fig. 7. Sequence of Loops

2. An element in cmb may be represented as a finite sequence alternating occurrences of the first and the second loops. It is possible to enumerate these combinations with a simple algorithm which unfolds the loops, computes the weight, compares with $rp - ep$ and backtracks to search another solution. Each of these combinations may be viewed as a sequence of l_i paths, where l_i represents the first loop with i occurrences of the inner loop. From these combinations we generate axioms starting in the $L1$ state. If the combination contains a l_i with $i > 0$ then we need to remove the extra inner loops. The recursion is used as in Section 4.2 and $w = 0$.

We do not have completely detailed this case. There are particular cases when the weights are equal to zero but they may be handled using recursion.

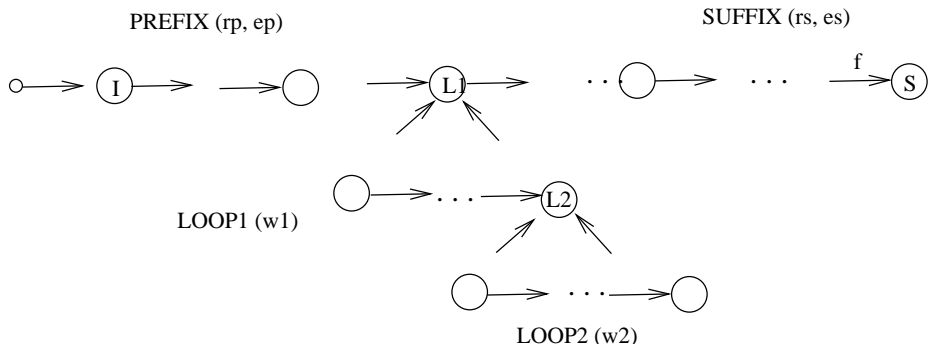


Fig. 8. Composition of Loops

This provides an algorithm (`callSequence`) which computes from a given S state and a $msg.arg$ operation.argument name a finite set of axioms to retrieve the argument of the message visible from S . The extracting process is complex but it is automatic. The argument retrieval function has to be defined with recursion if there are some loops without occurrences of the `msg` message. This also appears with composed loops, but sometimes it is not mandatory.

5 Application Examples

For the Bank example the only interesting and simple case is to know the parameters of the order message in state `process` with an empty buffer. This is a simple path without loop, the generated axiom for the price is:

price: AXIOM $\text{idle?}(\text{self}) \supset \text{price}(\text{order_E}(\text{order_R}(\text{self}, c, p))) = p$

A more interesting example is the `client` of the request receipt in state 2 of the Company. This is a simple path with one loop on state 1 `request`. The prefix is `request_R`, the loop has weight equal to 1, and the suffix is `request_E`. The numbers are: $rp = 1, ep = 0, rs = 0, es = 1, w = 1$, thus $n = 1$, we have two axioms.

c1: AXIOM $I?(\text{self}) \supset \text{client}(\text{request_E}(\text{request_R}(\text{self}, f, c))) = c$

c2: AXIOM $I\text{request?}(\text{self}) \supset \text{client}(\text{request_E}(\text{checkPlace}(\text{book_R}(\text{request_R}(\text{book_E}(\text{request_E}(\text{self})), f, c), fl, c1)))) = c$

An example to illustrate the oldest receipt in a loop is Figure 9. In this example we need one loop to consume the receipt of the prefix. The oldest receipt in the loop is obviously the second transition, we have to discard the first execution when computing the right message.

a1: AXIOM $I?(\text{self}) \Rightarrow \text{arg}(\text{a}>(\text{>a}(\text{self}, X))) = X$

% n=1

a2: AXIOM $L1?(\text{self}) \Rightarrow \text{arg}(\text{a}>(\text{>a}(\text{a}>(\text{self}), Y))) = Y$

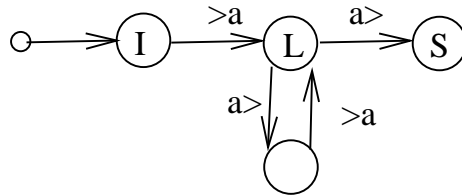


Fig. 9. An Example with a Loop

A concrete example for the sequence of loops is described in the Figure 10. In this example $N1 = 1$ and the second $n = 1$ too.

$P_I(\text{self}) \Rightarrow \text{arg}(f(\text{a}>(\text{g}(\text{>a}(\text{self}, X)))) = X$

$P_L1(\text{self}) \Rightarrow \text{arg}(f(\text{a}>(\text{g}(\text{>a}(\text{a}>(\text{self}), Y)))) = Y$

$P_L2(\text{self}) \Rightarrow \text{arg}(f(\text{a}>(\text{a}>(\text{>a}(\text{self}, Z)))) = Z$

An example with composed loop is given in Figure 11. In this example $rp - ep = 1$ and the set *cmb* of minimal combination is simply $\{l_0, l_1\}$. We have to generate a simple path axiom starting from *I* and +++

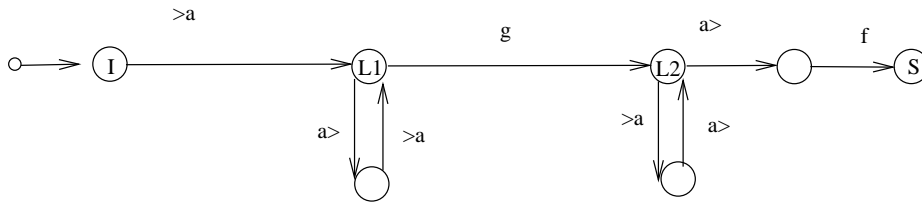


Fig. 10. A Loop Sequence Example

```

% simple path
P_I(self) => arg(a>(>a(self, X))) = X
% one loop1
P_L1(self) => arg(a>(>a(a>(self), Y)))) = Y
% one loop1+loop2
P_L1(self) => arg(a>(>a(a>(>a(a>(self), X)), Y))) = Y
% more than one loop2 inside loop1
P_L1(self) => arg(a>(>a(a>(>a(a>(>a(a>(self), Z)), X)), Y)) = arg(a>(>a(a>(>a

```

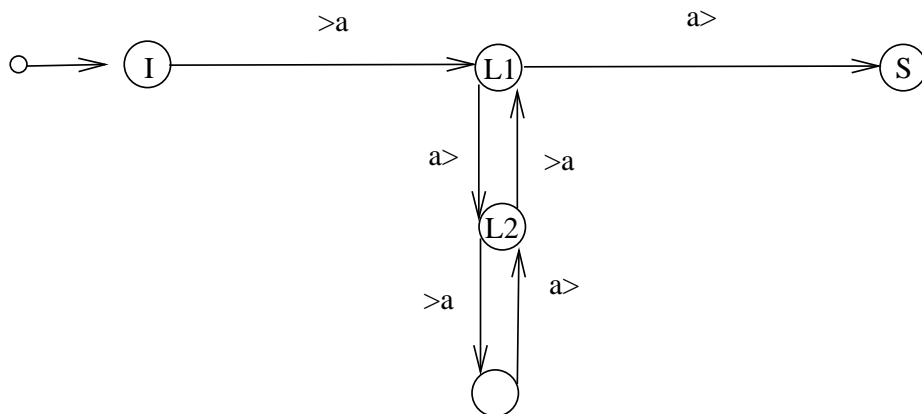


Fig. 11. A Composed Loop Example

In this case there is a general simplification which is to define one axiom starting from the $L2$ state.

References

1. André Arnold. *Finite Transition Systems*. International Series in Computer Science. Prentice-Hall, 1994.
2. Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, volume B of *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. Elsevier, 1990. Jan Van Leeuwen, Editor.

3. Olivier Maréchal, Pascal Poizat, and Jean-Claude Royer. Checking Asynchronously Communicating Components Using Symbolic Transition Systems. In Z. Tari R. Meersman and al., editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer Verlag, 2004.
4. Jacques Noyé, Sébastien Pavel, and Jean-Claude Royer. A PVS Experiment with Asynchronous Communicating Components. In *17th Workshop on Algebraic Development Techniques*, Barcelona, Spain, 2004. www.emn.fr/x-info/jroyer/rrWADT04.pdf.gz.