Submodule Construction Using Derivatives*

Esfandiar Haghverdi and Hasan Ural
School of Information Technology and Engineering, University of Ottawa
Ottawa, Ontario, K1N 6N5, CANADA
{ehaghver,ural}@site.uottawa.ca

Abstract

Top-down design methodology is one of the widely used approaches to the design of complex concurrent systems. In this approach, the specification of a system is decomposed into a set of submodules whose concurrent behavior is equivalent to that of the system specification. The following problem is of particular importance when using this methodology: given the specification of a system and some of its submodules, derive a specification for the remaining submodules. This problem can be formulated in CCS as $A\|_L X \approx B$ where "$\|_L$" is the combination of CCS operators *composition* and *restriction*, A is the composition of all known submodules, X is the unknown submodule, B represents the entire system and L represents the actions that may be used for communication between submodules. This paper presents an algorithm for the solution of $A\|_L X \approx B$ where B is deterministic. The proposed algorithm uses the concept of derivatives. The use of the algorithm is demonstrated by several examples.

1 Introduction

Modularization has been one of the most powerful, effective and elegant solutions to a lot of problems encountered in software and system design areas. Problems like protocol conversion and top-down design methodology are but a few topics to mention. More specifically in these problems the designer is confronted with the following situation: Given a desirable behavior B for the system to be designed, find the modules $A_1, A_2, \ldots, A_n$ such that the combined behavior of these modules which is usually expressed via some appropriate composition operator be in some desirable and meaningful sense equivalent to B. An interesting subproblem of this problem, which can gainfully be used in the top-down design, may be described as follows: Assuming that some modules $A_1, \ldots, A_{n-1}$ are already known, find the module A_n such that $A_1\|A_2\|\ldots\|A_n \equiv B$ where "$\|$" and "$\equiv$" represent composition and equality respectively. A particular application of this method lies in the design of communication protocols, where specifications of protocol service and two submodules, say, sender and medium are given and a specification for the receiver is sought. This idea has

*This work is supported in part by Natural Sciences and Engineering Research Council of Canada under grant number STR0149338.

been presented and elaborated in [4]. The notions of equality and composition abound in literature which shows the multifariousness of models of concurrency. In this paper, we will consider the *composition* "|" and *restriction* "$\backslash L$" operators of CCS [5] to model the composition of modules. Our notion of equality will be *observation equivalence* "$\approx$" of Milner [5,6]. Hence, the subproblem mentioned above can be restated more rigorously as: Given modules $A_1, \ldots, A_{n-1}$ as CCS processes, find A_n such that $(A_1|\ldots|A_{n-1}|A_n) \setminus L \approx B$ which in turn can be concisely expressed as the problem of finding a solution to equations of the form $(A|X) \setminus L \approx B$ with X unknown. We present an algorithm for solving such equations subject to constraints that X is rigid and B is deterministic.

This problem has attracted considerable attention in literature, for example [3,4,8,9,10]. Larsen and Liu [3] use a different notion for equality, namely Milner's *strong bisimulation* and hence are able to treat the case when the assumption of a deterministic B is dropped. We will confine our attention to observation equivalence.

Our approach extends the work in [8] and [9]. By providing an algorithm for constructing the solution of the equation we extend the work in [8]. On the other hand, the proposed algorithm prunes the state space of X more quickly than the one given in [9]. Also for certain equations with solutions constructed by the proposed algorithm, the algorithm in [9] will not yield any solutions (see Example 2). We defer further comparison to Section 4.

The rest of the paper is organized as follows: In Section 2, we introduce the syntax and some basic definitions. Most of the definitions in Section 2 are taken from [6,7] and are introduced to provide a brief introduction to CCS. In Section 3 we present and prove correct our algorithm for the solution of $(A|X) \setminus L \approx B$. Finally we recast some of the related work in this area comparing them to our approach in Section 4 where we also discuss possible directions for future research.

2 Basic Definitions

We presuppose an infinite set $\mathcal{A}$ of *names* $a, b, c, \ldots$. Then $\overline{\mathcal{A}}$ is the set of *co-names* $\overline{a}, \overline{b}, \overline{c}, \ldots$; $\mathcal{A}$ and $\overline{\mathcal{A}}$ are disjoint and are in bijection via $(\bar{\ })$; we declare $\overline{\overline{a}} = a$. $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ is the set of labels; we shall use l to range over $\mathcal{L}$, and K, L to range over subsets of $\mathcal{L}$, and we define $\overline{L} = \{\overline{l} : l \in L\}$. Labels will identify the actions which may be performed by a process. We also introduce a distinguished *silent action* $\tau \notin \mathcal{L}$; this special action is considered to be unobservable. We set $Act = \mathcal{L} \cup \{\tau\}$, and we shall use α and β to range over Act. A *relabelling function* f is a function from $\mathcal{L}$ to $\mathcal{L}$ such that $f(\overline{l}) = \overline{f(l)}$; f is extended to Act by setting $f(\tau) = \tau$. We presuppose a countable set $\mathcal{X}$ of *process variables* $X, Y, \ldots$ and a countable set $\mathcal{K}$ of *process constants* $A, B, \ldots$. We use I as an arbitrary (possibly infinite) indexing set. The set $\mathcal{E}$ of *process expressions* $E, F, \ldots$ (also called terms) is the smallest set containing $\mathcal{X}, \mathcal{K}, E, E_i\,(i \in I)$ closed under the following operations:

$\alpha.E$, *a Prefix* $(\alpha \in Act)$,

$\sum_{i \in I} E_i$, *a Summation*,

$E_0|E_1$, *a Composition*,

$E \setminus L$, *a Restriction.*

$E[f]$, *a Relabelling* (f a relabelling function).

For any expression E, its *free* variables $fv(E)$ are those occurring unbound in E. We say E is closed if $fv(E) = \emptyset$; in this case E is called a *process*. We denote the set of processes by $\mathcal{P}$ and shall use P, Q, R to range over the processes.

The meaning of the process constructions is as follows: $\alpha.P$ is the process which performs α and then behaves as P; $\sum_{i \in I} P_i$ is the process which may behave as any of P_i; $P_0|P_1$ represents P_0 and P_1 performing concurrently, with possible communication; $P \setminus L$ behaves like P but with actions in $L \cup \overline{L}$ prohibited. We will confine ourselves with $I = \emptyset$ and $I = \{0, 1\}$. When $I = \emptyset$ we write NIL, which is called *inaction*; it is the process which can do nothing. When $I = \{0, 1\}$ we write $P_0 + P_1$.

The action relations $\xrightarrow{\alpha}$, $\alpha \in Act$ are defined to be the smallest which obey the following rules:

$$ACT : \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad SUM_j : \frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i \in I} E_i \xrightarrow{\alpha} E'_j}(j \in I)$$

$$COM_0 : \frac{E_0 \xrightarrow{\alpha} E'_0}{E_0|E_1 \xrightarrow{\alpha} E'_0|E_1} \qquad COM_1 : \frac{E_1 \xrightarrow{\alpha} E'_1}{E_0|E_1 \xrightarrow{\alpha} E_0|E'_1}$$

$$COM_2 : \frac{E_0 \xrightarrow{l} E'_0 \quad E_1 \xrightarrow{\overline{l}} E'_1}{E_0|E_1 \xrightarrow{\tau} E'_0|E'_1} \qquad REL : \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$RES : \frac{E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L}(\alpha \notin L \cup \overline{L})$$

We will use process constants to define processes using *defining equations* of the form $A \stackrel{def}{=} P$ for the constants. Note that P may contain A. We have the simple rule: $\dfrac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}$ whenever $A \stackrel{def}{=} P$.

Consider the process A defined below as an example:
$A \stackrel{def}{=} a.A_1 + a.A_2$
$A_1 \stackrel{def}{=} b.A$
$A_2 \stackrel{def}{=} c.A_3 + d.A$
$A_3 \stackrel{def}{=} NIL$

We will think of process constants A, A_1, A_2 and A_3 as states of process A. We use S_A to denote the set of states of A.

We will combine the composition and restriction operators in a single operator called *communication* and denoted by $E_0 \|_L E_1$ where $L \subseteq \mathcal{L}$. Thus, the rules COM_0, COM_1, COM_2, and RES will be replaced by the following rules in this paper:

$$COMM_0 : \frac{E_0 \xrightarrow{\alpha} E_0' \quad \alpha \notin L \cup \overline{L}}{E_0 \|_L E_1 \xrightarrow{\alpha} E_0' \|_L E_1} \qquad COMM_1 : \frac{E_1 \xrightarrow{\alpha} E_1' \quad \alpha \notin L \cup \overline{L}}{E_0 \|_L E_1 \xrightarrow{\alpha} E_0 \|_L E_1'}$$

$$COMM_2 : \frac{E_0 \xrightarrow{l} E_0' \quad E_1 \xrightarrow{\overline{l}} E_1' \quad l \in L \cup \overline{L}}{E_0 \|_L E_1 \xrightarrow{\tau} E_0' \|_L E_1'}$$

We define $next : \mathcal{P} \longrightarrow 2^{Act}$, which maps a given process P to the set of actions which P can perform immediately. That is, $next(P) = \{\alpha \in Act : P \xrightarrow{\alpha} P' \textit{ for some } P'\}$, for example $next(NIL) = \emptyset$ and $next(a.NIL) = \{a\}$. For a process $P \in \mathcal{P}$, we use V_P to denote the set of actions that occur in the definition of P, this set is called *alphabet of P*.

Definition 1 We define an operator ∂_α called α-derivative, $\partial_\alpha : \mathcal{E} \longrightarrow 2^{\mathcal{E}}$, $E \overset{\partial_\alpha}{\mapsto} \partial_\alpha(E)$ where $\partial_\alpha(E) = \{E' : E \xrightarrow{\alpha} E'\}$. The recursive determination of derivatives is facilitated by the following rules:

$\partial_\alpha(NIL) = \emptyset \;\; \forall \alpha \in Act$,

$\partial_\alpha(\beta.E) = \begin{cases} \{\text{E}\}, & \text{if } \alpha = \beta; \\ \emptyset, & \text{otherwise.} \end{cases}$

$\partial_\alpha(E_0 + E_1) = \partial_\alpha(E_0) \cup \partial_\alpha(E_1)$,

$\partial_\alpha(E_0 \|_L E_1) = \{E_0' \|_L E_1 : E_0' \in \partial_\alpha(E_0)\} \cup \{E_0 \|_L E_1' : E_1' \in \partial_\alpha(E_1)\}$

Note that $\{E_0' \|_L E_1 : E_0' \in \partial_\alpha(E_0)\} = \emptyset$ if $\partial_\alpha(E_0) = \emptyset$, likewise for $\{E_0 \|_L E_1' : E_1' \in \partial_\alpha(E_1)\}$.

When in $E_0 \|_L E_1$, E_0 and E_1 move with complementary actions say, $l, \overline{l}$ the expression itself will make a τ transition, that is, we have to consider $\partial_\tau(E_0 \|_L E_1)$, however to distinguish this τ action from those which occur simply by τ transitions of E_0 and/or E_1 we shall use the ∂_{τ_l} notation. The following rule defines the derivative in this case:

$$\partial_{\tau_l}(E_0 \|_L E_1) = \{E_0' \|_L E_1' : E_0' \in \partial_l(E_0) \textit{ and } E_1' \in \partial_{\overline{l}}(E_1)\}$$

for $l \in L \cup \overline{L}$ and $l \in next(E_0)$.

We extend ∂_α to strings over Act^* (* stands for Kleene star): $\partial_\epsilon(E)$ is defined to be $\{E\}$. $\partial_{\alpha.t}(E) = \bigcup_{E' \in \partial_\alpha(E)} \partial_t(E')$ for $\alpha \in Act$ and $t \in Act^*$. We will now give some standard definitions in terms of derivatives:

A process P is called finite iff $\bigcup_{s \in V_P^*} \partial_s(P)$ is finite, in other words iff there are finitely many syntactically different processes that can be derived from P.

A process P is *rigid* if $\tau \notin V_P$ and *deterministic* iff it is rigid and cardinality of $\partial_s(P)$ is either 0 or 1 for all $s \in V_P^*$.

The rest of the definitions given in this section are taken from [6]:
If $t \in Act^*$, then $\widehat{t} \in \mathcal{L}^*$ is the sequence gained by deleting all occurrences of τ from t. There are no τs in $\widehat{t}$ and we have $\widehat{(\tau^n)} = \epsilon$.

Let t be $\alpha_1 \ldots \alpha_n \in Act^*$. Then $E \xrightarrow{t} E'$ if $E \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} E'$ and $E \stackrel{t}{\Rightarrow} E'$ if

$$E(\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^* E'$$

Thus $E \xrightarrow{t} E'$ implies $E \stackrel{t}{\Rightarrow} E'$, and $E \stackrel{t}{\Rightarrow} E'$ implies $E \stackrel{\widehat{t}}{\Rightarrow} E'$.

A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over processes is a (weak) *bisimulation* if $(P, Q) \in \mathcal{S}$ implies, for all $\alpha \in Act$,

1. Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \stackrel{\widehat{\alpha}}{\Rightarrow} Q'$ and $(P', Q') \in \mathcal{S}$.

2. Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \stackrel{\widehat{\alpha}}{\Rightarrow} P'$ and $(P', Q') \in \mathcal{S}$.

P and Q are *observation equivalent* or *(weakly) bisimilar*, written $P \approx Q$, if $(P, Q) \in \mathcal{S}$ for some (weak) bisimulation $\mathcal{S}$. That is,

$$\approx = \bigcup\{\mathcal{S} : \mathcal{S} \text{ is } a\ bisimulation\}$$

3 Solution of $A\|_L X \approx B$

In this section we will present an algorithm to solve equations of the form $A\|_L X \approx B$. We assume that A and B are finite and B is deterministic and X is rigid. To simplify the procedure we further assume that B is minimal with respect to (weak) bisimulation. The set L (assumed to satisfy $L \subseteq V_A$) represents the set of actions that may be used for communication between A and X. Alphabet of X is determined by the algorithm and is minimal such that X solves the equation. However V_X must satisfy $V_B - V_A \subseteq V_X \subseteq (V_A \cup V_B) \cup \overline{L}$.

Informally, the idea behind the approach is as follows: Suppose there exists an X that is a solution of the equation $A\|_L X \approx B$. This means that $A\|_L X$ and B are observation equivalent or in other words there is a (weak) bisimulation $\mathcal{S}$ such that $((A, X), B) \in \mathcal{S}$. Our approach then aims at constructing a process X using some operators that transform the equations into new ones while preserving the bisimulation $\mathcal{S}$. This means that we will force X to behave in such a way that $((A, X), B) \in \mathcal{S}$ as far as the behavior of X is concerned. Naturally, this will imply certain restrictions on the applicability of operators at each stage of the algorithm and the set of admissible pairs (A, B) which can coexist i.e., can form the left and right hand sides of an equation with a fixed state of X. This set will be called *context* of, denoted by $\mathcal{C}(\ )$, and will be defined formally below. Context of a process X has a very interesting property: assume we have the following chain of equations obtained by successive application of some operator, say, Op beginning from $A\|_L X \approx B$,

$$A\|_L X \approx B - Op \rightarrow A_1\|_L X \approx B_1 - Op \rightarrow A_2\|_L X_1 \approx B_1$$

Moreover, assume that $((A, X), B)) \in \mathcal{S}$ where $\mathcal{S}$ is a bisimulation. Now note that the equations given above involve two processes X and X_1 (or two states of process X) thus we have $\mathcal{C}(X) = \{(A, B), (A_1, B_1)\}$ and $\mathcal{C}(X_1) = \{(A_2, B_1)\}$. The set $\mathcal{C}(X)$ in this example contains the pairs which coexist with X in the bisimulation relation that relates (A, X) and (A_1, X) to B and B_1, respectively. In this way, contexts are related to bisimulations, hence they must satisfy certain properties which are implied by definition of bisimulation. Equations of the form $A\|_L X \approx B$ will be denoted by the following triple $(A, X; B)_L$ and the subscript L will be removed whenever there is no risk of ambiguity. For the definitions given below, let $\mathcal{R}$ be a set of triples constructed by successive application of derivatives starting with $(A, X; B)_L$.

Definition 2 An arbitrary state X_m of X occurs with one or more pairs of states of A and B in the triples in $\mathcal{R}$. The set of these pairs is represented by a set called the *context* of X_m, thus

$$\mathcal{C}(X_m) = \{(A_i, B_j) : (A_i, X_m; B_j) \in \mathcal{R}\}$$

Note that $\mathcal{C}(X_m) \subseteq S_A \times S_B$, for all $X_m \in S_X$.

Given $\mathcal{C}(X_m)$, for each $(A_i, X_m; B_j) \in \mathcal{R}$, we define the set ev as:

$$ev = next(A_i) \cup \{\alpha \in next(B_j) - V_{A_i} : \alpha \in next(B_l) \text{ and } \alpha \notin next(A_k) \text{ for all } (A_k, B_l) \in \mathcal{C}(X_m)\}$$

In Example 1, $\mathcal{C}(X) = \{(A, B), (A_1, B_1)\}$ and at node $(A, X; B)$, $ev = \{a\}$ and at the node $(A_1, X; B_1)$, $ev = \{b\}$. Note that a and c are not in the latter ev set because a is in V_{A_1} and c is not in $next(B)$.

This set determines the operators ∂_α to be applied at $(A_i, X_m; B_j)$. That is, at $(A_i, X_m; B_j)$ we apply ∂_α iff $\alpha \in ev$ of $(A_i, X_m; B_j)$.
The following definition characterizes the valid contexts.

Definition 3 Let X_m be an arbitrary state of X. $\mathcal{C}(X_m)$ is said to be *valid* iff the following conditions hold:

1. For any pair $(A_i, B_j) \in \mathcal{C}(X_m)$ and for all $\alpha \in V_{A_i} - L$:
   (a) If $\alpha \neq \tau$ and $A_i \xrightarrow{\alpha} A_k$, then $B_j \xrightarrow{\alpha} B_l$ and $(A_k, B_l) \in \mathcal{C}(X_m)$,
   (b) If $\alpha = \tau$ and $A_i \xrightarrow{\tau} A_k$, then $(A_k, B_j) \in \mathcal{C}(X_m)$;
2. For any two pairs $(A_i, B_j), (A_k, B_l) \in \mathcal{C}(X_m)$ and for all $\alpha \in V_{A_i} - L$:
   (a) If $\alpha \neq \tau$ and $A_i \xrightarrow{\alpha} A_k$, then $B_j \xrightarrow{\alpha} B_l$,
   (b) If $\alpha = \tau$ and $A_i \xrightarrow{\tau} A_k$, then $B_j \equiv B_l$ ("$\equiv$" denotes syntactical identity);
3. Let $\alpha \in V_{B_j}$ then for any pair $(A_i, B_j) \in \mathcal{C}(X_m)$, if $\alpha \in next(B_j)$ then there exist some $t \in (L \cup \{\tau\})^*$, A_i', and X_m' such that $A_i \xrightarrow{t} A_i'$, $X_m \xrightarrow{\bar{\hat{t}}} X_m'$ and $\alpha \in ev$ of $(A_i', X_m'; B_j)$.

Next, we define the application of derivatives to equations which follows naturally from the definition of derivative.

Definition 4 Let $(A_i, X_m; B_j) \in \mathcal{R}$ denote an arbitrary equation where X_m is of the form $\partial_{s_m}(X)$ for some $s_m \in V_X^*$ and $\mathcal{C}(X_m)$ be valid.

Suppose that X is deterministic* and hence $\partial_{s_m}(X)$ is a singleton if nonempty. Abbreviate $\partial_{s_m}(X) = \{X'\}$ to X'.

Then for a given ev of $(A_i, X_m; B_j)$ and $\alpha \in ev$ the following cases are possible:

1. $\alpha \in V_{A_i} - (L \cup \{\tau\})$
$(A_i, \partial_{s_m}(X); B_j) \xrightarrow{\partial_\alpha} \{(A_k, \partial_{s_m}(X); B_l)\}$, for $B_l \in \partial_\alpha(B_j)$ and for all $A_k \in \partial_\alpha(A_i)$. Note that $\partial_\alpha(B_j)$ is a singleton since B is deterministic;

2. $\alpha = \tau$
$(A_i, \partial_{s_m}(X); B_j) \xrightarrow{\partial_\tau} \{(A_k, \partial_{s_m}(X); B_j)\}$, for all $A_k \in \partial_\tau(A_i)$;

3. $\alpha \in L$
$(A_i, \partial_{s_m}(X); B_j) \xrightarrow{\partial_{\tau\alpha}} \{(A_k, \partial_{s_m\overline{\alpha}}(X); B_j)\}$, for all $A_k \in \partial_\alpha(A_i)$;

4. $\alpha \in V_{B_j} - V_{A_i}$
$(A_i, \partial_{s_m}(X); B_j) \xrightarrow{\partial_\alpha} (A_i, \partial_{s_m\alpha}(X); B_l)$, for $B_l \in \partial_\alpha(B_j)$.

We noted earlier that every state X_m of X that appears in some $(A_i, X_m; B_j) \in \mathcal{R}$ is of the form $\partial_{s_m}(X)$ for some $s_m \in V_X^*$. Hereafter, we will use the latter notation exclusively when referring to states of X.

Definition 5 Any triple in $\mathcal{R}$ from which no further processing is necessary or possible is called a *terminal node*. Let s_k and s_l be arbitrary strings in V_X^*. Each terminal node is one of the following types:

1. A triple of the form $(NIL, \partial_{s_k}(X); NIL)$;

2. All triples of the form $(A_i, \partial_{s_l}(X); B_j)$ where there exists a $(A_i', \partial_{s_k}(X); B_j') \in \mathcal{R}$ such that $s_k < s_l$ (s_k is a prefix of s_l and $s_k \neq s_l$) and $\mathcal{C}(\partial_{s_k}(X)) = \mathcal{C}(\partial_{s_l}(X))$.

We associate a regular expression with each terminal node, namely;

1. If $(A_i, \partial_{s_k}(X); B_j)$ is a terminal node of type (1) in Definition 5, the expression is $e = s_k$;

2. If $(A_i, \partial_{s_l}(X); B_j)$ is a terminal node of type (2) in Definition 5, the expression is $e = s_k(s_k')^*$ where $s_k s_k' = s_l$.

*This is natural since in a nondeterministic case, we will have more than one state of X in $\partial_{s_m}(X)$ all with the same context and hence one can easily conclude that nonexistence of a deterministic solution will entail that a nondeterministic solution cannot exist.

Definition 6 With every terminal node $(A_i, \partial_{s_k}(X); B_j) \in \mathcal{R}$ we associate a set M defined as:

$$M = \bigcup_{s' \leq s_k} \mathcal{C}(\partial_{s'}(X)).$$

we refer to this set as the M-set of this node.

Definition 7 Let $(A', \partial_{s_k}(X); B')$ be a terminal node in $\mathcal{R}$ and $M = \bigcup_{s' \leq s_k} \mathcal{C}(\partial_{s'}(X))$ be the associated M-set. M is said to be *closed* if for every element $(A_i, B_j) \in M$ (i.e., $(A_i, B_j) \in \mathcal{C}(\partial_s(X)), s \leq s_k$), the following conditions hold:

1. For all $\alpha \in V_{A_i} \cap L$, if $A_i \xrightarrow{\alpha} A_k$, then $s\overline{\alpha} \leq s_k$ or $\mathcal{C}(\partial_{s\overline{\alpha}}(X))$ is invalid;
2. For all $\alpha \in V_{B_j} - V_{A_i}$, if $B_j \xrightarrow{\alpha} B_l$, then $s\alpha \leq s_k$ or $\alpha \notin ev$ of $(A_i, \partial_s(X); B_j)$.

Note that $s\overline{\alpha} \leq s_k$ and $s\alpha \leq s_k$ imply that $(A_k, B_j) \in M$ and $(A_i, B_l) \in M$ respectively.

3.1 The Proposed Algorithm

Let $(A, X; B)_L$ be the given equation. This will be the initial node of the algorithm. At each step the algorithm finds the contexts of the states (of X) involved in the nodes being processed. These contexts will be checked for validity and termination. If the context of the node under processing is valid and the node is not a terminal node, the algorithm continues by exploring new nodes and this is achieved by making use of derivatives which are applied to nodes to generate new ones. At any given node, the set of derivatives to be applied is determined by the set ev of the node. Algorithm uses a procedure (**Proc**) to determine the contexts. Below we give the formal definition of the algorithm.

Procedure Proc
input: $(A_i, \partial_{s_k}(X); B_j)$

$$Output := \begin{cases} (A_i, \partial_{s_k}(X); B_j) \cup Y, & \\ (A_i, \partial_{s_k}(X); B_j), & \text{if } \exists (A_k, \partial_{s_k}(X); B_l) \in Y \text{such that } \partial_\alpha(B_l) = \emptyset \\ & \text{for some } \alpha \in V_{A_k} - L. \end{cases}$$

where $Y = \{(A_k, \partial_{s_k}(X); B_l) : (A_i, \partial_{s_k}(X); B_j) \xrightarrow{\partial_t} (A_k, \partial_{s_k}(X); B_l),$
for some $t \in (V_{A_i} - L)^*\}$.

Algorithm:
pre and *post*: sets of nodes,
X^i, A^i, B^i: The state of X, A and B in node i,
N: set of nodes that are explored,
e_i: the regular expression corresponding to node i,
M_i: the M-set corresponding to node i,
M: set of M-sets, Exp: set of regular expressions,
V_e: set of events occurring in the regular expression e,
$\mathcal{S}$: the smallest bisimulation relation such that $(A, X; B) \in \mathcal{S}$,
ev_n: the ev set at node n.

input: $(A, X; B)$
Output: $X, V_X, \mathcal{S}$ if a solution exists, "no solution" otherwise.
pre:=$\mathbf{Proc}((A, X; B))$;
while $pre \neq \emptyset$ do
  begin
    for every node n in pre do
      begin
        form $\mathcal{C}(X^n)$;
        if $\mathcal{C}(X^n)$ is valid then
            begin
              $N := N \cup \{n\}$;
              if n is a terminal node and $V_{e_n} \cap (V_{B^n} - V_{A^n}) \neq \emptyset$ then
                begin
                  if M_n is closed then solution e is found $e := e_n$, $\mathcal{S} := M_n$; else
                    if $\bigcup_{i \in N} M_i$ is closed then solution e is found $e := \sum_{i \in N} e_i$,
                    $\mathcal{S} := \bigcup_{i \in N} M_i$; else
                      $M := M \cup M_n$; $Exp := Exp \cup \{e_n\}$;
                end
              else
                begin
                  determine ev_n;
                  for all $\alpha \in ev_n \cap (L \cup (V_{B^n} - V_{A^n}))$ do
                  begin
                    $post := post \cup \mathbf{Proc}(\partial_\alpha(n))$;
                  end
                end
            end
        else delete all the nodes $m \in pre$ such that $\mathcal{C}(X^m) = \mathcal{C}(X^n)$;
      end
    $pre := post; post := \emptyset$;
  end { while } "no solution"

If a solution exists, the algorithm will yield a regular expression e, the desired solution is obtained using the unique (up to isomorphism) deterministic recognizer for e with X as initial state and $V_X = V_e$ (the set of events occurring in e). Let F be the nondeterministic automaton recognizing e and let F' be the deterministic automaton constructed from F by applying the well-known procedure in [2]. Every state q of F' is a set of states of F. The desired solution X is obtained as follows:

1. Two states q, q' of F' are identified if one is the subset of the other.

2. All states q of F' with no transitions are distinguished.

Example 1 The following example is adapted from [8].

$$
\begin{array}{lll}
A \stackrel{def}{=} a.A_1 & B \stackrel{def}{=} a.B_1 & L = \{b\} \\
A_1 \stackrel{def}{=} b.A & B_1 \stackrel{def}{=} c.B + a.B_2 & V_A = V_{A_1} = \{a, b\} \\
 & B_2 \stackrel{def}{=} c.B_1 & V_B = V_{B_1} = V_{B_2} = \{a, c\}
\end{array}
$$

The proposed algorithm generates the following derivation tree for this example:

$$
\begin{tikzcd}
 & \begin{array}{l}(A, X; B)\\ ev = \{a\}\end{array} \arrow[d, "\partial_a"] & & \\
 & \begin{array}{l}(A_1, X; B_1)\\ ev = \{b\}\end{array} \arrow[d, "\partial_{\tau_b}"] & & \\
 & \begin{array}{l}(A, \partial_{\overline{b}}(X); B_1)\\ ev = \{a, c\}\end{array} \arrow[dl, "\partial_c"'] \arrow[dr, "\partial_a"] & & \\
\begin{array}{l}(A, \partial_{\overline{b}c}(X); B)\\ ev\end{array} \arrow[d, "\partial_a"] & & \begin{array}{l}(A_1, \partial_{\overline{b}}(X); B_2)\\ ev = \{b, c\}\end{array} \arrow[dl, "\partial_{\tau_b}"'] \arrow[dr, "\partial_c"] & \\
\begin{array}{l}(A_1, \partial_{\overline{b}c}(X); B_1)\\ ev\end{array} & \begin{array}{l}(A, \partial_{\overline{b}\overline{b}}(X); B_2)\\ ev\end{array} & & \begin{array}{l}(A_1, \partial_{\overline{b}c}(X); B_1)\\ ev\end{array}
\end{tikzcd}
$$

Valid contexts are:
$\mathcal{C}(X) = \{(A, B), (A_1, B_1)\}$
$\mathcal{C}(\partial_{\overline{b}}(X)) = \{(A, B_1), (A_1, B_2)\}$
$\mathcal{C}(\partial_{\overline{b}c}(X)) = \{(A, B), (A_1, B_1)\}$
$\mathcal{C}(X) = \mathcal{C}(\partial_{\overline{b}c}(X))$

There is one terminal node which yields the regular expression $e = (\overline{b}c)^*$.
$M = \{(A, B), (A_1, B_1), (A, B_1), (A_1, B_2)\}$ and the solution is:
$X \stackrel{def}{=} \overline{b}.X_1,\ X_1 \stackrel{def}{=} c.X$ and $V_X = \{\overline{b}, c\}$.

Example 2 Consider the following processes:

$$
\begin{array}{lllll}
A \stackrel{def}{=} a.A_1 & B \stackrel{def}{=} a.B_1 & L = \{b\} & \\
A_1 \stackrel{def}{=} b.A_2 & B_1 \stackrel{def}{=} a.B_2 & V_A = \{a, b\} & V_B = \{a\} \\
A_2 \stackrel{def}{=} NIL & B_2 \stackrel{def}{=} NIL & V_{A_1} = \{b\} & V_{B_1} = \{a\} \\
 & & V_{A_2} = \emptyset & V_{B_2} = \emptyset
\end{array}
$$

The proposed algorithm generates the following derivation tree for this example:

$$
\begin{tikzcd}
\begin{array}{l}(A, X; B)\\ ev = \{a\}\end{array} \arrow[d, "\partial_a"] \\
\begin{array}{l}(A_1, X; B_1)\\ ev = \{b\}\end{array} \arrow[d, "\partial_{\tau_b}"] \\
\begin{array}{l}(A_2, \partial_{\overline{b}}(X); B_1)\\ ev = \{a\}\end{array} \arrow[d, "\partial_a"] \\
\begin{array}{l}(A_2, \partial_{\overline{b}a}(X); B_2)\\ ev\end{array}
\end{tikzcd}
$$

Valid contexts are:
$\mathcal{C}(X) = \{(A, B), (A_1, B_1)\}$
$\mathcal{C}(\partial_{\overline{b}}(X)) = \{(A_2, B_1)\}$
$\mathcal{C}(\partial_{\overline{b}a}(X)) = \{(A_2, B_2)\}$

There is one terminal node which yields the regular expression $e = \overline{b}a$.
$M = \{(A, B), (A_1, B_1), (A_2, B_1), (A_2, B_2)\}$ and the solution is:
$X \stackrel{def}{=} \overline{b}.X_1,\ X_1 \stackrel{def}{=} a.X_2,\ X_2 \stackrel{def}{=} NIL$ and $V_X = \{\overline{b}, a\}$.

Note that the algorithm given in [9] will produce no solution for this example, since the set of external events is empty.

Example 3 Consider the following processes:

$$
\begin{array}{llll}
A \stackrel{def}{=} a.A_1 + a.A_2 & B \stackrel{def}{=} a.B_1 & L = \{\lambda, \beta\} & \\
A_1 \stackrel{def}{=} \beta.A_3 & B_1 \stackrel{def}{=} c.B_1 + d.B_2 & V_A = \{a, \lambda, \beta\} & V_B = \{a, c, d\} \\
A_2 \stackrel{def}{=} \lambda.A_4 & B_2 \stackrel{def}{=} NIL & V_{A_1} = \{\beta\} & V_{B_1} = \{c, d\} \\
A_3 \stackrel{def}{=} NIL & & V_{A_2} = \{\lambda\} & V_{B_2} = \emptyset \\
A_4 \stackrel{def}{=} NIL & & V_{A_3} = V_{A_4} = \emptyset &
\end{array}
$$

The proposed algorithm generates the following derivation tree for this example:

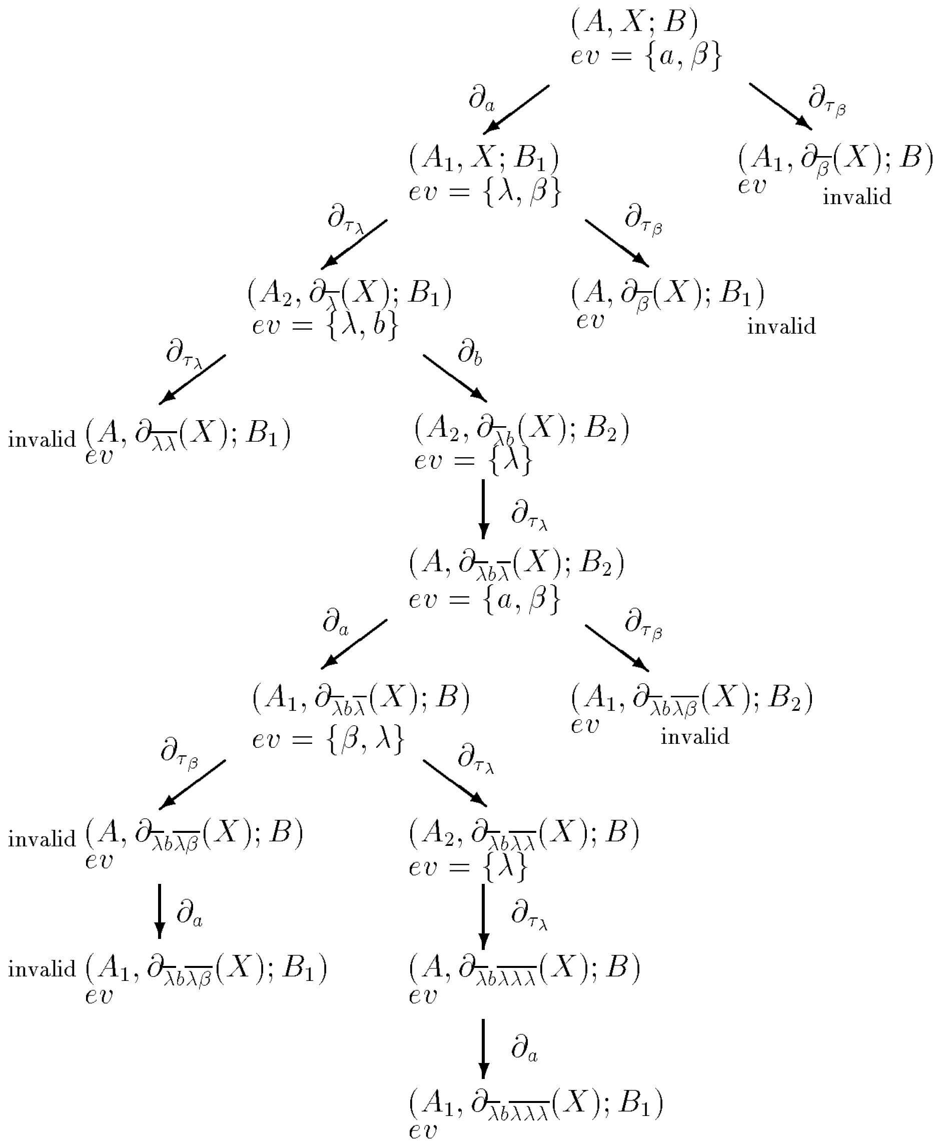
$$
\begin{tikzcd}[column sep=small]
& & & \begin{array}{l}(A,X;B)\\ ev=\{a\}\end{array} \arrow[dll, "\partial_a"'] \arrow[drr, "\partial_a"] & & & \\
& \begin{array}{l}(A_1,X;B_1)\\ ev=\{\beta\}\end{array} \arrow[d, "\partial_{\tau_\beta}"'] & & & & \begin{array}{l}(A_2,X;B_1)\\ ev=\{\lambda\}\end{array} \arrow[d, "\partial_{\tau_\lambda}"'] & \\
& \begin{array}{l}(A_3,\partial_{\overline{\beta}}(X);B_1)\\ ev=\{c,d\}\end{array} \arrow[dl, "\partial_c"'] \arrow[dr, "\partial_d"] & & & & \begin{array}{l}(A_4,\partial_{\overline{\lambda}}(X);B_1)\\ ev=\{c,d\}\end{array} \arrow[dl, "\partial_c"'] \arrow[dr, "\partial_d"] & \\
\begin{array}{l}(A_3,\partial_{\overline{\beta}c}(X);B_1)\\ ev\end{array} & & \begin{array}{l}(A_3,\partial_{\overline{\beta}d}(X);B_2)\\ ev\end{array} & & \begin{array}{l}(A_4,\partial_{\overline{\lambda}c}(X);B_1)\\ ev\end{array} & & \begin{array}{l}(A_4,\partial_{\overline{\lambda}d}(X);B_2)\\ ev\end{array}
\end{tikzcd}
$$

Valid contexts are:
$\mathcal{C}(X) = \{(A,B),(A_1,B_1),(A_2,B_1)\}$
$\mathcal{C}(\partial_{\overline{\beta}}(X)) = \{(A_3,B_1)\}$
$\mathcal{C}(\partial_{\overline{\lambda}}(X)) = \{(A_4,B_1)\}$
$\mathcal{C}(\partial_{\overline{\beta}c}(X)) = \{(A_3,B_1)\}$
$\mathcal{C}(\partial_{\overline{\beta}d}(X)) = \{(A_3,B_2)\}$
$\mathcal{C}(\partial_{\overline{\lambda}c}(X)) = \{(A_4,B_1)\}$
$\mathcal{C}(\partial_{\overline{\lambda}d}(X)) = \{(A_4,B_2)\}$
There are four terminal nodes which yield the regular expression $e = \overline{\beta}c^* + \overline{\beta}d + \overline{\lambda}c^* + \overline{\lambda}d$.
$M = \{(A,B),(A_1,B_1),(A_2,B_1),(A_3,B_1),(A_4,B_1),(A_3,B_2),(A_4,B_2)\}$ and the solution is:
$X \stackrel{def}{=} \overline{\beta}.X_1 + \overline{\lambda}.X_2$
$X_1 \stackrel{def}{=} c.X_1 + d.X_3$
$X_2 \stackrel{def}{=} c.X_2 + d.X_4$
$X_3 \stackrel{def}{=} NIL$
$X_4 \stackrel{def}{=} NIL$ and $V_X = \{\overline{\beta},\overline{\lambda},c,d\}$

Example 4 The following example is adapted from [9].

$$
\begin{array}{lll}
A \stackrel{def}{=} a.A_1 + \beta.A_1 & B \stackrel{def}{=} a.B_1 & L = \{\lambda,\beta\} \\
A_1 \stackrel{def}{=} \beta.A + \lambda.A_2 & B_1 \stackrel{def}{=} b.B_2 & V_A = V_{A_1} = V_{A_2} = \{a,\lambda,\beta\} \\
A_2 \stackrel{def}{=} \lambda.A & B_2 \stackrel{def}{=} a.B & V_B = V_{B_1} = V_{B_2} = \{a,b\}
\end{array}
$$

The proposed algorithm generates the following derivation tree for this example:



Invalid nodes shown in the derivation tree generated by the algorithm will be discarded by the algorithm. They are kept in the figure to facilitate the understanding of the algorithm. Valid contexts are:

$\mathcal{C}(X) = \{(A, B), (A_1, B_1)\}$
$\mathcal{C}(\partial_{\overline{\lambda}}(X)) = \{(A_2, B_1)\}$
$\mathcal{C}(\partial_{\overline{\lambda}b}(X)) = \{(A_2, B_2)\}$
$\mathcal{C}(\partial_{\overline{\lambda}b\overline{\lambda}}(X)) = \{(A, B_2), (A_1, B)\}$
$\mathcal{C}(\partial_{\overline{\lambda}b\overline{\lambda\lambda}}(X)) = \{(A_2, B)\}$
$\mathcal{C}(\partial_{\overline{\lambda}b\overline{\lambda\lambda\lambda}}(X)) = \{(A, B), (A_1, B_1)\} = \mathcal{C}(X)$

The regular expression found by the algorithm is $e = (\overline{\lambda}b\overline{\lambda\lambda\lambda})^*$.
$M = \{(A, B), (A_1, B_1), (A_2, B_1), (A_2, B_2), (A, B_2), (A_1, B), (A_2, B)\}$ and the solution is:

$X \stackrel{def}{=} \overline{\lambda}.X_1$
$X_1 \stackrel{def}{=} b.X_2$
$X_2 \stackrel{def}{=} \overline{\lambda}.X_3$
$X_3 \stackrel{def}{=} \overline{\lambda}.X_4$
$X_4 \stackrel{def}{=} \overline{\lambda}.X$, and $V_X = \{b, \overline{\lambda}\}$.

3.2 Correctness of the Algorithm

Proposition 1 Given the equation $(A, X; B)_L$ as the initial node, let $(A', \partial_{s_k}(X); B')$ be an arbitrary terminal node generated by the algorithm and $M = \bigcup_{s' \leq s_k} \mathcal{C}(\partial_{s'}(X))$ be its associated M-set. The set $\mathcal{S}$ defined as $\mathcal{S} = \bigcup_{s' \leq s_k} \bigcup_{(A_i, B_j) \in \mathcal{C}(\partial_{s'}(X))} \{((A_i, \partial_{s'}(X)), B_j)\}$ is a bisimulation if M is closed.

Proof: Let $((A_i, \partial_{s_l}(X)), B_j)$ be an arbitrary element of $\mathcal{S}$. Note that all the contexts appearing in M and $\mathcal{S}$ are valid since the algorithm deletes all the invalid contexts. Finally $(A_i, B_j) \in M$ since $((A_i, \partial_{s_l}(X)), B_j) \in \mathcal{S}$.
For all $\alpha \in Act$,

1. $\alpha \in V_{A_i} - L$ and $\alpha \neq \tau$
   If $A_i \xrightarrow{\alpha} A_k$, then $B_j \xrightarrow{\alpha} B_l$ and $(A_k, B_l) \in \mathcal{C}(\partial_{s_l}(X))$ by validity of $\mathcal{C}(\partial_{s_l}(X))$ (Definition 3, 1(a)). Hence, $B_j \stackrel{\hat{\alpha}}{\Rightarrow} B_l$ and $((A_k, \partial_{s_l}(X)), B_l) \in \mathcal{S}$.

2. $\alpha \in V_{A_i} - L$ and $\alpha = \tau$
   If $A_i \xrightarrow{\tau} A_k$ then $(A_k, B_j) \in \mathcal{C}(\partial_{s_l}(X))$ by validity of $\mathcal{C}(\partial_{s_l}(X))$ (Definition 3, 1(b)). Hence, $B_j \stackrel{\epsilon}{\Rightarrow} B_j$ $(\hat{\tau} = \epsilon)$ and $((A_k, \partial_{s_l}(X)), B_j) \in \mathcal{S}$.

3. $\beta \in V_{A_i} \cap L$
   If $A_i \xrightarrow{\beta} A_k$ and $\partial_{s_l}(X) \xrightarrow{\overline{\beta}} \partial_{s_l\overline{\beta}}(X)$ and $\mathcal{C}(\partial_{s_l\overline{\beta}}(X))$ is valid, then by Definition 4 (3), $(A_k, B_j) \in \mathcal{C}(\partial_{s_l\overline{\beta}}(X))$. Moreover, $(A_k, B_j) \in M$ since M is closed. Hence, $B_j \stackrel{\epsilon}{\Rightarrow} B_j$ $(\hat{\tau} = \epsilon)$. Finally, $(A_k, B_j) \in M$ and $(A_k, B_j) \in \mathcal{C}(\partial_{s_l\overline{\beta}}(X))$ imply that $((A_k, \partial_{s_l\overline{\beta}}(X)), B_j) \in \mathcal{S}$.

4. Transitions of type $\partial_{s_l}(X) \xrightarrow{\alpha} \partial_{s_l\alpha}(X)$ for $\alpha \in V_{B_j} - V_{A_i}$ occur only when B makes a transition with α. Hence, next item also covers this case.

5. $\alpha \in V_{B_j} - V_{A_i}$
   If $B_j \xrightarrow{\alpha} B_l$, then by Definition 3 (3), there exists some $t \in (L \cup \{\tau\})^*$ such that $A_i \xrightarrow{t} A'_i$ and $\alpha \in ev$ of $(A'_i, \partial_{s_l\overline{t}}(X); B_j)$. Moreover, by (2) and (3) above, $(A'_i, B_j) \in M$. Assume $\mathcal{C}(\partial_{s_l\overline{t}\alpha}(X))$ is valid, then by Definition 4 (4), $(A'_i, B_l) \in \mathcal{C}(\partial_{s_l\overline{t}\alpha}(X))$. Since M is closed, we have $(A'_i, B_l) \in M$. Therefore, $(A_i, \partial_{s_l}(X)) \stackrel{\hat{\alpha}}{\Rightarrow} (A'_i, \partial_{s_l\overline{t}\alpha}(X))$ and $((A'_i, \partial_{s_l\overline{t}\alpha}(X), B_l) \in \mathcal{S}$. Note that there exists at least one $t \in (L \cup \{\tau\})^*$ such that

$\mathcal{C}(\partial_{s_l\widehat{t\alpha}}(X))$ is valid, since otherwise $((A_i, \partial_{s_l}(X))), B_l)$ will lead to nodes which are all invalid and hence cannot belong to M.

6. $\alpha \in V_{A_i} \cap V_{B_j}$
If $B_j \xrightarrow{\alpha} B_l$, then by Definition 3 (3), there exists a $t \in (L \cup \{\tau\})^*$ such that $A_i \xrightarrow{t} A_i'$ and $\alpha \in next(A_i')$. By (2) and (3) above, $(A_i', B_j) \in M$ and $((A_i', \partial_{s_l\hat{t}}(X), B_j) \in \mathcal{S}$. Let $A_i' \xrightarrow{\alpha} A_k$, then $(A_i, \partial_{s_l}(X)) \stackrel{\hat{\alpha}}{\Rightarrow} (A_k, \partial_{s_l\hat{t}}(X))$ and by (1) above, $((A_k, \partial_{s_l\hat{t}}(X), B_l) \in \mathcal{S}$.

Hence, by definition of bisimulation, $\mathcal{S}$ is a bisimulation and the proof is complete. □

Corollary 1 Let $(A', \partial_{s_k}(X); B')$ and $(A'', \partial_{s_l}(X); B'')$ be two terminal nodes generated by the algorithm. Consider the set $M = M_1 \cup M_2$ where M_1 and M_2 are the associated M-sets of $(A', \partial_{s_k}(X); B')$ and $(A'', \partial_{s_l}(X); B'')$, respectively. If M is closed, then $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ where $\mathcal{S}_1 = \bigcup_{s' \leq s_k} \bigcup_{(A_i,B_j)\in\mathcal{C}(\partial_{s'}(X))} \{((A_i, \partial_{s'}(X)), B_j)\}$ and $\mathcal{S}_2 = \bigcup_{s' \leq s_l} \bigcup_{(A_i,B_j)\in\mathcal{C}(\partial_{s'}(X))}$ $\{((A_i, \partial_{s'}(X)), B_j)\}$ is a bisimulation.

The sets M and $\mathcal{S}$ in Corollary 1 can be defined for finite number of sets.

Theorem 1 Consider the equation $A \|_L X \approx B$. The proposed algorithm terminates and finds a solution iff this equation has a solution.

Proof: Assume $A \|_L X \approx B$ has a solution P, then there exists a bisimulation $\mathcal{S}$ such that $((A, P), B) \in \mathcal{S}$, hence $\mathcal{C}(P)$ is valid. For all $\alpha \in Act$,

1. Whenever $(A, P) \xrightarrow{\alpha} (A', P')$ then, for some B', $B \stackrel{\hat{\alpha}}{\Rightarrow} B'$ and $((A', P'), B') \in \mathcal{S}$.

2. Whenever $B \xrightarrow{\alpha} B'$ then, for some (A', P'), $(A, P) \stackrel{\hat{\alpha}}{\Rightarrow} (A', P')$ and $((A', P'), B') \in \mathcal{S}$.

Validity of $\mathcal{C}(P)$ implies that $\mathcal{C}(X)$ is valid. By (1) and (2) above, application of derivatives to $(A, X; B)$ will result in newly generated nodes $(A', \partial_s(X); B')$ with $\mathcal{C}(\partial_s(X))$ valid, therefore the set of valid nodes generated by the algorithm starting with $(A, X; B)$ is not empty. By repetitive application of derivatives, the algorithm will finally terminate and explore all the terminal nodes, since number of contexts is finite (indeed the maximum number of possible valid contexts is $2^{(S_A \times S_B)} - 1$). Moreover, there are n $(n \geq 1)$ terminal nodes, the union of whose M-sets is closed since otherwise either there is no bisimulation or number of contexts is not finite contrary to the assumptions. By Proposition 1, a bisimulation $\mathcal{S}'$ will be obtained which includes $((A, X), B)$, hence a solution is found. This completes the sufficiency part.

For necessity, assume that the algorithm finds a solution. Then there are n $(n \geq 1)$ terminal nodes union of whose M-sets is closed. By Proposition 1 the set $\mathcal{S}$ defined using these M-sets (Corollary 1) is a bisimulation and $((A, X), B) \in \mathcal{S}$, thus $A \|_L X \approx B$ has a solution. □

4 Conclusion and Related Work

In this paper we have studied the solution of CCS equations of the form $A\|_L X \approx B$ and presented and proven correct an algorithm which finds the smallest bisimulation $\mathcal{S}$ such that $((A, X), B) \in \mathcal{S}$, hence a smallest solution is constructed provided the given equation has a solution. Note, however that this does not imply the state minimality of X and a further minimization may reduce the number of states. The main ideas are the formulation of a derivative operator and its use to transform equations toward a solution and controlling the behavior of X to keep $((A, X), B)$ in a bisimulation. The key observation is the use of subsets of $S_A \times S_B$ formulated as contexts to represent the information content of states of X. A similar idea is used in [9], however, in our study the notion of context arises naturally as a context of a state of X i.e., $\mathcal{C}(\partial_s(X))$ represents the set of pairs $(A, B) \in S_A \times S_B$ that coexist with $\partial_s(X)$ in forming a bisimulation. Hence, $\mathcal{C}(\partial_s(X))$ completely determines the behavior of $\partial_s(X)$. The transformational approach to the solution of $A\|_L X \approx B$ advocated here provides an algorithm as a basis for the implementation of a program which will make the solution of large equations encountered in practice possible.

One possible extension of this work is to consider larger classes of equations; that is equations in which X appears in an arbitrary environment (CCS context) or system of simultaneous equations like, $A\|X \approx B$ and $A'\|X \approx B'$. Another direction for extending this work is to consider different parallel operators and equivalence relations, for example parallel operator of TCSP and failure equivalence [1].

The problem of submodule construction is treated in [4] for communicating finite state machines and in [8,9,10] for CCS processes. Merlin and Bochmann [4] consider finite state machines instead of processes and trace equivalence instead of observation equivalence. Another difference is that according to the definition of parallel composition, the simultaneous execution of two actions does not always result in an unobservable (τ) action. Authors derive a solution in terms of the "complement" operation of an automaton. The method is applied to generate the receiver of the Alternating Bit Protocol. One limitation noted by authors is that an automatically generated specification for a submodule may cause deadlocks which stems from the fact that trace equivalence is insensitive to deadlock potentials.

In [10], equations of type $(A|X) \setminus L \approx B$ are called "interface equations" and necessary and sufficient conditions for the existence of solutions of such equations are given for the case where B is *weakly determinate* and subject to some requirements on the sorts of A and B. An algorithm for constructing a solution is presented. This construction and the requirements for existence of a solution are formulated in terms of the state spaces of A and B.

Among the related work cited above, [8] and [9] are closest to our study since they have treated the same problem using a similar definition for parallel operator and observation equivalence as the notion for equality and under the same assumptions on A, B, and X. Hence, we undertake a more detailed comparison with their work.

In [8], Parrow studies equations of type $(A|X) \setminus L \approx B$ and presents a method for solving such equations where B is deterministic. The method is based on a general tableau framework where a solution is derived via a sequence of transformations. Hence, tableau method formalizes the successive transformational approach to the solution of the equation. These transformations form a basis for the implementation of a semi-automatic program. This program attempts to find a most general solution (a solution which simulates every other solution). The program performs some of the transformations automatically, but a user can effect critical transformations in order to guide the program towards a suitable solution which may be less general but smaller in size.

In the algorithm proposed in this paper, we use a transformational approach as well. However, we formalize our approach using derivative operators which provide a much simpler framework to work with and reason about than that provided by tableau method and their application is less cumbersome than tableau transformations. Moreover, we do not need heuristics in guiding the algorithm since the possibilities of unacceptable solutions are prohibited by the rules encoded in the definition of context validity and determination of ev sets.

The work by Qin and Lewis [9] extends the work by Shields [10]. In [9], authors consider equations of the form $A|||X \approx B$, called "weak factorization" equations and use a finite state machine model consistent with CCS and $|||$ is a derived CCS composition operator similar to our $||$ operator and $\approx$ represents observational equivalence. An algorithm is presented and proven correct to find the most general solution X to $A|||X \approx B$ if solutions exist, where B is deterministic and X is rigid. The key observation remarked by authors is the use of set of pairs of states of A and B to represent a state of X also called box which corresponds to our context. However, an invalid context is more restrictive that a bad state of [9] i.e., a bad state is always an invalid context, whereas not all invalid contexts are bad states. This issue gives rise to a substantial difference between two algorithms, namely the algorithm presented in [9] generates states iteratively, beginning from initial state and when a bad state is encountered a backtracking is necessary to remove incoming links to this bad state and subsequently the effected states are removed. In our algorithm however, an invalid context is detected once generated and is removed immediately. For example, in Example 3 treated here, $\mathcal{C}(\partial_{\overline{\lambda}b\lambda\overline{\beta}}(X))$ corresponds to state X_6 in the same example treated in [9] where it is removed after a few iterations back from state X_8 (a bad state). Moreover, an extra state is generated and subsequently removed (X_9). It is also shown that (see Example 2) the proposed algorithm in our paper produces solutions to a class of equations for which the algorithm in [9] yields no solutions.

Another difference is pointed out in what follows. At step 1 and 2 of the algorithm in [9], all state pairs in $S_A \times S_B$ and their I sets are generated. Generation of I sets corresponds to running Procedure **Proc** introduced here. This marks another important difference between the two algorithms, that is, our algorithm operates incrementally by generating contexts when necessary and terminates as soon as closedness condition of M-sets is met, though there may be other nodes to be processed. Hence, our algorithm will produce the smallest bisimulation containing $((A, X), B)$.

References

[1] Brooks, S., Hoare, C.A.R., Roscoe, W. (1984) A Theory of Communicating Sequential Processes. *Journal of ACM*, **31**, 560–599.
[2] Hopcroft, J.E. and Ullman, J.D. (1979) *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley.
[3] Larsen K. and Liu X. (1990) Equation Solving Using Modal Transition Systems. *in Proc. LICS.*
[4] Merlin, P. and Bochmann, G.V. (1983) On the Construction of Submodule Specification and Communication Protocols. *ACM Trans. on Prog. Lang. and Sys.*, **5**, 1–25.
[5] Milner, R. (1980) *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92, Springer–Verlag, Berlin.
[6] Milner, R. (1989) *Communication and Concurrency.* Prentice–Hall.
[7] Milner, R. (1990) Operational and Algebraic Semantics of Concurrent Processes. In Van Leeuwen, J. (ed.), *Handbook of Theoretical Computer Science*, Volume B. Elsevier Science Publishers B.V.
[8] Parrow, J. (1989) Submodule Construction as Equation Solving in CCS. *Theoretical Computer Science*, **68**, 175–202.
[9] Qin, H. and Lewis, P. (1989) Factorization of Finite State Machines under Observation Equivalence. In *Proceedings of International Workshop on Automatic Verification Methods for Finite State Systems.* Lecture Notes in Computer Science 407, Springer–Verlag, Berlin.
[10] Shields, M.W. (1989) Implicit System Specification and the Interface Equation. *The Computer Journal*, **32**, 399–412.

Full Title: Submodule Construction Using Derivatives

Authors: Esfandiar Haghverdi and Hasan Ural
School of Information Technology and Engineering, University of Ottawa
Ottawa, Ont. K1N 6N5, CANADA.

e-mail addresses: $\{ehaghver, ural\}@site.uottawa.ca$

Correspondence Author:
Prof. Hasan Ural
School of Information Technology and Engineering
University of Ottawa
150 Louis Pasteur Priv.
Ottawa, Ontario, K1N 6N5
CANADA

e-mail: ural@site.uottawa.ca
Tel: (613) 562 - 5800 Ext: 6684
Fax: (613) 562 - 5185

Keywords : Top-down design, Automated synthesis, Submodule construction.