Submodule Construction as Equation Solving in CCS

Joachim Parrow*
Laboratory for Foundations of Computer Science
University of Edinburgh

Abstract

A method for solving CCS equations of type $(A|X)\backslash L \approx B$, where X is unknown, is presented. The method is useful in a top-down design methodology: if a system (B) and some of its submodules (A) are specified, solving such an equation amounts to constructing the missing submodules. The method works by successively transforming equations into simpler equations, in parallel with generation of a solution. It has been implemented as a semi-automatic program, which has been applied to the generation of receivers of two alternating-bit protocols.

1 Introduction

One of the most important and difficult fields in computer science is to develop methods for construction of complex systems. Most design methodologies rely on *modularization*: systems are partitioned into sets of submodules, and each submodule is given a specification. These specifications contain sufficient information for combining the modules. Thus, implementation details can be disregarded in most stages of the design.

In this paper, we will consider the particular problem of designing systems composed of several nondeterministic modules executing in parallel. As an example, consider a communication protocol, where the submodules are a sender, a receiver, and a medium. In [MB83], Merlin and Bochmann observe that when all but one of the submodules have been specified, a specification of the remaining module can be derived automatically. For example, when the sender and medium have been specified, the specification of the receiver can be deduced. One limitation in [MB83] is that specifications are expressed in terms of execution sequences. This means that they do not contain enough information to determine some aspects, e.g. deadlock potentials, of the behaviour of the system. Thus, a receiver satisfying the automatically generated specification may cause deadlocks.

Our contribution in this paper is to apply the ideas in [MB83] to a more refined specification method, namely Milner's Calculus of Communicating Systems (CCS, see e.g. [Mil80]). Specifications are in CCS called *agents*, and there is a notion of *observation equivalence*, written $\approx$, between agents. Essentially, two agents are observation equivalent if they can not be distinguished by an external observer. This equivalence is more discriminating than comparing execution sequences; in particular it is sensitive to deadlock potentials. There is also a formal syntax for combining agents: a system composed of agents $A_1, A_2, \ldots, A_n$ executing in parallel and communicating over channels L is written

$$(A_1|A_2|\cdots|A_n)\backslash L$$

In a top-down design methodology, the designer starts with an agent, call it B, representing the behaviour of the whole system to be constructed. He divides the system into n modules,

*On leave from the Swedish Institute of Computer Science, Stockholm, Sweden

communicating over channels L, and proceeds to construct for each module i an agent A_i. The criterion for a correct construction is that

$$(A_1|\cdots|A_n)\backslash L \approx B$$

Now, assume that the designer has actually constructed all modules but one, say A_n. The missing module can then be obtained as a solution for X of the CCS equation (for clarity, put $A = A_1|\cdots|A_{n-1}$):

$$(A|X)\backslash L \approx B \qquad (\dagger)$$

The theory of such equations has to some extent been studied by Shields ([Shi86b]); we defer a discussion on this and other related work to section 8.

Unfortunately, equations of type (†) may in general have several solutions, some of which are unsuitable for implementation. There are several reasons for this. Some solutions, although correct, are unnecessarily complex. For example, consider a communication protocol where the receiver always retransmits each acknowledgment at least a million times. This receiver might be formally correct, but would be highly inefficient. Other solutions are correct only in a formal sense, because of idealizations in the specification. An example of this is presented in section 7. Since we believe it impossible to give general criteria on good solutions, we feel that a semi-automatic procedure, where a designer can guide the generation of agents towards suitable solutions, is appropriate. In this way our method differs from that of Shields.

We have developed such a procedure and implemented it as a program, which accepts equations of type (†) where A and B are finite state agents, and B is deterministic. Having tested the program on nontrivial examples, we conclude that when it runs without user interaction, strange solutions might result. Operated by a designer with some idea of what a good solution will look like, more sensible solutions will be generated. The program helps the designer to find an agent which is guaranteed to be (formally) correct, or convinces him that no such agent exists.

The procedure is based on stepwise transformation of equations into simpler equations. As an example (we here assume the reader to be familiar with the fundamental concepts of CCS), assume that we want to solve

$$a.NIL\,|\,X \;\approx\; a.b.NIL + b.a.NIL$$

Here, the right hand side can do actions a and b. Since $a.NIL$ can only do a, X must supply the b action. Substituting $b.Y$ for X, we get the equation

$$a.NIL\,|\,b.Y \;\approx\; a.b.NIL + b.a.NIL$$

and by applying the expansion theorem on the left hand side we get

$$a.(NIL\,|\,b.Y) + b.(a.NIL\,|\,Y) \;\approx\; a.b.NIL + b.a.NIL$$

but by congruence properties, this is implied by

$$\begin{cases} NIL\,|\,b.Y \;\approx\; b.NIL \\ a.NIL\,|\,Y \;\approx\; a.NIL \end{cases}$$

We have now transformed the original equation into two simpler equations, and by similar reasoning we will find that $Y = NIL$ solves them both. Hence, $X = b.NIL$ is a solution.

The rest of this paper is structured as follows. In section 2 we define the syntax and semantics of the part of CCS which is related to our work. In section 3 we present the ideas of transforming equations in general, and provide a sufficient requirement for a transformation to be sound. Sections 4 and 5 contain the particular transformations needed for solving equations of type (†), and a proof that these transformations are complete in the sense that if there is a solution, then it can be generated by a sequence of transformations. In section 6 we describe an implementation in the form of a semi-automatic program. This program is applied in section 7 to nontrivial examples: generating the receiver and medium of two versions of the alternating-bit protocol. Section 8 contains ideas for extending this work, and comparisons with similar efforts.

2 Preliminaries

In this section, we establish the notation for the rest of the paper. Although all concepts will be formally defined, the reader is advised to consult some introduction to CCS such as [Mil80] for intuition.

Assume a set **Act** of *actions* where the *inverse* of action a is $\bar{a}$. The unobservable action τ has no inverse. Let **Id** be a set of *identifiers*. The set **Ag** of *agents* is the smallest set that contains **Id** and is closed under the following operators:

$a.$ (called prefixing) for $a \in \mathbf{Act}$; prefix operator.

$+$ (called nondeterministic choice); binary infix operator.

$|$ (called parallel composition); binary infix operator.

$\backslash L$ (called restriction) for $L \subseteq \mathbf{Act} - \{\tau\}$, L finite; postfix operator.

NIL agent constant.

As a shorthand for $(A|B)\backslash L$ we will write $A \parallel_L B$, or even $A \parallel B$ when L is unimportant or understood from the context. As a shorthand for $A_1 + A_2 + \cdots + A_n$ we write

$$\sum_{i=1}^{n} A_i$$

If $n = 1$, the above sum is just A_1, and if $n = 0$ it is *NIL*.

An *environment* $\mathcal{E}$ is a partial function from **Id** to **Ag**. An identifier is said to be *bound* by $\mathcal{E}$ if it is in the domain of $\mathcal{E}$, otherwise it is *free* in $\mathcal{E}$. The behaviour of an agent (i.e. the transitions that the agent can perform) is always determined with respect to an environment. Thus, we have for each $a \in \mathbf{Act}$ and environment $\mathcal{E}$ the binary transition relation $\xrightarrow{a}_{\mathcal{E}}$ on agents. These relations are defined to be the smallest relations satisfying the following clauses, which can be regarded as the operational semantics of the operators:

$$\frac{}{a.A \xrightarrow{a}_{\mathcal{E}} A} \qquad \frac{A \xrightarrow{a}_{\mathcal{E}} A'}{B + A \xrightarrow{a}_{\mathcal{E}} A'} \qquad \frac{A \xrightarrow{a}_{\mathcal{E}} A'}{A + B \xrightarrow{a}_{\mathcal{E}} A'}$$

$$\frac{A \xrightarrow{a}_{\mathcal{E}} A'}{A|B \xrightarrow{a}_{\mathcal{E}} A'|B} \qquad \frac{A \xrightarrow{a}_{\mathcal{E}} A'}{B|A \xrightarrow{a}_{\mathcal{E}} B|A'} \qquad \frac{A \xrightarrow{a}_{\mathcal{E}} A',\ B \xrightarrow{\bar{a}}_{\mathcal{E}} B'}{A|B \xrightarrow{\tau}_{\mathcal{E}} A'|B'}$$

$$\frac{A \xrightarrow{a}_{\mathcal{E}} A',\ a \notin L,\ \bar{a} \notin L}{A\backslash L \xrightarrow{a}_{\mathcal{E}} A'\backslash L} \qquad \frac{\mathcal{E}(X) \xrightarrow{a}_{\mathcal{E}} A}{X \xrightarrow{a}_{\mathcal{E}} A}$$

An agent B is a *derivative* of A (w.r.t. an environment $\mathcal{E}$) if for some $n \geq 0$ there are actions $a_1, \ldots, a_n$ such that

$$A \xrightarrow{a_1}_{\mathcal{E}} \cdots \xrightarrow{a_n}_{\mathcal{E}} B$$

If $n = 0$, this formula is interpreted as $A = B$.

We define the *experiment* relation $\stackrel{\hat{a}}{\Longrightarrow}_{\mathcal{E}}$, $a \in \mathbf{Act}$ on agents in the following way:

$$A \stackrel{\hat{a}}{\Longrightarrow}_{\mathcal{E}} B \text{ iff } \begin{cases} \text{for some } n \geq 0\text{: } A \underbrace{\xrightarrow{\tau}_{\mathcal{E}} \cdots \xrightarrow{\tau}_{\mathcal{E}}}_{n \text{ times}} B & \text{if } a = \tau \\ \text{for some } n, m \geq 0\text{: } A \underbrace{\xrightarrow{\tau}_{\mathcal{E}} \cdots \xrightarrow{\tau}_{\mathcal{E}}}_{n \text{ times}} \xrightarrow{a}_{\mathcal{E}} \underbrace{\xrightarrow{\tau}_{\mathcal{E}} \cdots \xrightarrow{\tau}_{\mathcal{E}}}_{m \text{ times}} B & \text{if } a \neq \tau \end{cases}$$

When the environment is understood from the context we will drop the index $\mathcal{E}$ in $\rightarrow$ and $\Longrightarrow$.

An agent A is *deterministic* iff all its derivatives B satisfy the following:

$$\text{for all } a\text{: } B \stackrel{\widehat{a}}{\Longrightarrow} B' \text{ and } B \stackrel{\widehat{a}}{\Longrightarrow} B'' \text{ implies } B' = B''$$

Essentially, this means that the agent can never do any τ-actions, and that for each action a there is at most one transition labelled a from any given derivative.

Let $\sqsubseteq$ be the usual order on partial functions, i.e. $f \sqsubseteq g$ iff for all x such that $f(x)$ is defined, $g(x) = f(x)$. Applied on environments, we say that $\mathcal{F}$ *extends* $\mathcal{E}$ if $\mathcal{E} \sqsubseteq \mathcal{F}$.

Let $\mathcal{E}$ be an environment. Extend the notion of (syntactic) subexpressions of agents by including $\mathcal{E}(X)$ as a subexpression of X, for all identifiers X bound by $\mathcal{E}$. A free identifier Y is *guarded* in the agent A, iff all occurrences of Y in subexpressions of A are in subexpressions of type $a.B$. An agent is *well guarded* iff all free identifiers are guarded in it. This means that the initial transitions from the agent are not affected by extending $\mathcal{E}$. An agent is *closed* iff all identifiers in all subexpressions of it are bound. This means that neither transitions from the agent, nor transitions from any derivative, are affected by extending $\mathcal{E}$.

With respect to a particular environment, a binary relation R on closed agents is a *simulation* iff ARB implies:

$$\begin{aligned}&\text{for all } a \text{ and } A' \text{ such that } A \stackrel{\widehat{a}}{\Longrightarrow} A'\\ &\quad\text{there exists a } B' \text{ such that } B \stackrel{\widehat{a}}{\Longrightarrow} B' \text{ and } A'RB'\end{aligned}$$

If both R and R^{-1} are simulations, R is said to be a *bisimulation*. Two closed agents A and B are *observation equivalent*, written $A \approx B$, iff there exists a bisimulation R such that ARB. Observation equivalence is extended to non-closed agents by $A \approx B$ iff in all extensions where A and B are closed, $A \approx B$.

We will in the following use a, b, ... to denote actions, A, B, ... to denote agents, X, Y, ... to denote identifiers, and $\mathcal{E}$, $\mathcal{F}$, ... to denote environments.

3 Tableaux

As mentioned in the introduction, our aim is to present a procedure for solving equations of type $A \parallel X \approx B$ by successive transformations of equations. We formalise this reasoning by using tableaux. A *tableau* consists of two parts: a *goal* Γ and an environment $\mathcal{E}$. The intuition behind a tableau is that it represents an intermediate stage in producing a solution: the goal says what remains to be done, and the environment records the solution produced so far. A goal is a unary predicate over environments. For example, the goal "$X \approx Y$" is true of the environments assigning observationally equivalent agents to X and Y. As another example, the goal "$X + X \approx X$" is true of all environments.

In order to solve an equation of type $A \parallel X \approx B$ we will start with an initial tableau with goal "$A \parallel X \approx B$" and an environment where A and B are closed and X is free, meaning "it remains to find an environment satisfying $A \parallel X \approx B$." The procedure then works by successively simplifying the equations and extending the environment until a tableau with goal *true* is reached. The environment of that tableau will contain the desired solution.

In the rest of this section we will make these ideas formally rigorous. We use the following goals and satisfaction relation $\models$ between environments and goals:

$$\begin{array}{ll}\mathcal{E} \models \mathit{true} & \text{always}\\ \mathcal{E} \models A \approx B & \text{for all } \mathcal{E} \text{ such that } A \approx B \text{ in } \mathcal{E}\\ \mathcal{E} \models \Gamma \wedge \Gamma' & \text{iff } \mathcal{E} \models \Gamma \text{ and } \mathcal{E} \models \Gamma'\\ \mathcal{E} \models \Gamma \supset \Gamma' & \text{iff for all } \mathcal{F} \sqsupseteq \mathcal{E}, \mathcal{F} \models \Gamma \text{ implies } \mathcal{F} \models \Gamma'\end{array}$$

Note the definition of implication, reminiscent of Kripke-semantics for intuitionistic logic. The idea is that "Γ implies Γ'" is true when any evidence of Γ, i.e. extension of the environment where Γ holds, is also evidence of Γ'. Here, this can be thought of as implicit universal quantification over free identifiers. Actually, $\supset$ will never occur in any of our tableaux, but this notion of logical implication is convenient when formulating results about tableau transformations. In particular, we will use it when formulating the soundness result.

An important property of the satisfaction relation is the *preservation property*: if an environment satisfies a goal, then all extensions of the environment also satisfy the goal.

Proposition 1 (preservation property) *If $\mathcal{E} \models \Gamma$ and $\mathcal{E} \sqsubseteq \mathcal{F}$, then $\mathcal{F} \models \Gamma$*

Proof: By induction on Γ. It is important that the atomic goals (of type $A \approx B$) possess the preservation property; our definition of observation equivalence on non-closed agents ensures this. □

We will write $\langle \Gamma, \mathcal{E} \rangle$ for a tableau with goal Γ and environment $\mathcal{E}$. A tableau $\langle \Gamma, \mathcal{E} \rangle$ is *satisfiable* if there exists an extension $\mathcal{F}$ of $\mathcal{E}$ that satisfies Γ.

In the following, let $\longrightarrow$ represent a transformation, i.e. a binary relation, on tableaux. We say that such a transformation is *safe* if whenever $\langle \Gamma, \mathcal{E} \rangle \longrightarrow \langle \Gamma', \mathcal{E}' \rangle$, then $\mathcal{E} \sqsubseteq \mathcal{E}'$ and $\mathcal{E}' \models \Gamma' \supset \Gamma$. This means that safe transformations can only add to the environment, and only strengthen the goal. Thus, if the resulting tableau is satisfiable by a particular environment, then the original tableau is also satisfiable by the same environment. Note, however, that a safe transformation might transform a satisfiable tableau into an unsatisfiable one. As an example of a safe transformation, by the expansion theorem in CCS:

$$\langle (a.A \parallel_{\{a\}} X) \approx B,\ \emptyset \rangle \longrightarrow \langle \tau.(A \parallel_{\{a\}} Y) \approx B,\ \langle X \mapsto \bar{a}.Y \rangle \rangle$$

We now prove that safe tableau transformations are indeed sound: if we start with a tableau and successively transform it until we arrive at the goal *true*, then we have derived an extension of the environment which satisfies the original goal.

Proposition 2 (Soundness) *Let $\longrightarrow^*$ be the reflexive transitive closure of $\longrightarrow$. If $\longrightarrow$ is safe, then for all tableaux $\langle \Gamma, \mathcal{E} \rangle$:*

$$\text{If } \langle \Gamma, \mathcal{E} \rangle \longrightarrow^* \langle \mathit{true}, \mathcal{F} \rangle \text{ then } \mathcal{E} \sqsubseteq \mathcal{F} \text{ and } \mathcal{F} \models \Gamma$$

Proof: If $\longrightarrow$ is safe, then $\longrightarrow^*$ is safe. This is proven by induction on the length of the transformation sequence; here the preservation property is crucial. Hence, by the definition of safe transformations we get that

$$\langle \Gamma, \mathcal{E} \rangle \longrightarrow^* \langle \mathit{true}, \mathcal{F} \rangle \text{ implies } \mathcal{E} \sqsubseteq \mathcal{F} \text{ and } \mathcal{F} \models \mathit{true} \supset \Gamma$$

But $\mathcal{F} \models \mathit{true} \supset \Gamma$ is easily seen to be true iff $\mathcal{F} \models \Gamma$ is true. □

The tableau framework is not limited to solving equations of type $A \parallel X \approx B$. Indeed, it can be applied to any equation in CCS, or even to any predicates which have the preservation property.

4 Finite Agents

We will present a tableau method for solving $A \parallel X \approx B$ where A and B are *finite state*, i.e. having only finitely many syntactically different derivatives. Thus, the initial tableau is of type $\langle A \parallel X \approx B,\ \mathcal{E} \rangle$, where $\mathcal{E}$ only contains definitions of identifiers appearing in A and B. The main idea is to guess the initial actions of X, and subsequently split the equation into several, hopefully smaller, equations.

In general, a goal will be a conjunction of equations of type $E \approx C$, where C is closed in the environment. To present the tableau transformation $\longrightarrow$ in a readable way, we first give the transformations which are sufficient when A and B are finite, i.e. do not contain any recursively defined identifiers. There are two types of transformation rules: *instantiations* extend the environment, and *consequences* strengthen the goal.

- *Instantiation*: If $\langle \Gamma, \mathcal{E} \rangle$ is a tableau with $A \parallel X \approx B$ in the goal, and X is free in $\mathcal{E}$, then

$$\left\langle X \mapsto \sum_{i=1}^{n} a_i.X_i \right\rangle$$

  can be added to $\mathcal{E}$. Here, $a_1, \ldots, a_n$ are actions and $X_1, \ldots, X_n$ are fresh (free in $\mathcal{E}$) and distinct identifiers.

- *Consequence*: $\langle \Gamma, \mathcal{E} \rangle \longrightarrow \langle \Gamma', \mathcal{E} \rangle$ if Γ' is obtained from Γ in one of the following ways:

  - *Equivalence*: An equation $E \approx B$, where E and B are observation equivalent (w.r.t. $\mathcal{E}$) can be removed. Here, removing the last conjunct of a goal means to replace the goal with *true*.

  - *Splitting*: If Γ contains an equation $E \approx B$ where E is well guarded, and the transitions from E are $E \xrightarrow{e_j}_{\mathcal{E}} E_j$ for $j \in [1, \ldots, n]$, and there are agents $B_1, \ldots, B_n$ such that:

$$\text{for all } j\text{: } B \stackrel{\widehat{e_j}}{\Longrightarrow}_{\mathcal{E}} B_j$$

    and also

$$\mathcal{E} \models \left(\sum_{j=1}^{n} e_j.B_j \right) \approx B$$

    then the equation $E \approx B$ can be replaced by the equations

$$\bigwedge_{j=1}^{n} (E_j \approx B_j)$$

The rules deserve some comments. The instantiation transformation amounts to guessing the initial actions $a_1, \ldots, a_n$ of X. In section 6 we will provide heuristics for this. The equivalence transformation will be applied sparingly, since it is computationally expensive to check observation equivalence. If, as in this section, we restrict attention to finite agents, then it is sufficient to apply the equivalence transformation to "$NIL \approx NIL$". Even if arbitrary finite state agents are considered, it suffices to apply the equivalence transformation in the final stage of a transformation sequence (cf. the proof of proposition 7).

The purpose of the splitting rule is to split an equation $E \approx B$ into a set of equations $E_j \approx B_j$, where all E_j are derivatives of E and all B_j are derivatives of B. When B is deterministic, the requirements on B_j can be simplified as demonstrated by the following proposition:

Proposition 3 *Assume the following:*

$$\begin{array}{lcl} \tilde{e} & = & \{e_1, \ldots, e_n\} \text{ a finite set of actions} \\ \tilde{e}_o & = & \tilde{e} - \{\tau\} \\ B & & \text{An agent, deterministic and closed in } \mathcal{E} \\ \tilde{b}_o & = & \{b \neq \tau : \text{for some } B',\ B \xrightarrow{b}_{\mathcal{E}} B'\} \end{array}$$

Then, the premises of the splitting rule

$$\text{for all } j\text{: } B \stackrel{\widehat{e_j}}{\Longrightarrow}_{\mathcal{E}} B_j \qquad (A1)$$

$$\mathcal{E} \models \left(\sum_{j=1}^{n} e_j.B_j\right) \approx B \qquad (A2)$$

are equivalent with

$$\text{for all } j\text{: } \begin{cases} B = B_j & \text{if } e_j = \tau \\ B \xrightarrow{e_j}_{\mathcal{E}} B_j & \text{if } e_j \neq \tau \end{cases} \qquad (B1)$$

$$\tilde{e}_o \subseteq \tilde{b}_o \text{ and, if } \tau \notin \tilde{e},\ \tilde{e}_o = \tilde{b}_o \qquad (B2)$$

Proof:

(A1 $\Rightarrow$ B1) By A1, for each j there are two cases:

1. $e_j = \tau$. Since $B \stackrel{\hat{\tau}}{\Longrightarrow} B$ is always true for any B, and B is deterministic, it follows $B = B_j$.
2. $e_j \neq \tau$. Then, by A1, $B \xrightarrow{\tau} \cdots \xrightarrow{\tau} B' \xrightarrow{e_j} B'' \xrightarrow{\tau} \cdots \xrightarrow{\tau} B_j$. Since B, and hence B'', are deterministic, it follows $B = B'$ and $B'' = B_j$, i.e. $B \xrightarrow{e_j} B_j$.

(A2 $\Rightarrow$ B2) Since B is deterministic, it follows that $\tilde{b}_o$ is the set of observable experiments possible from B, i.e. $\tilde{b}_o = \{b \neq \tau : \text{for some } B'\text{: } B \stackrel{\hat{b}}{\Longrightarrow} B'\}$. By A2, this implies $\tilde{e}_o \subseteq \tilde{b}_o$. Also, if $\tau \notin \tilde{e}$, then $\tilde{e} = \tilde{e}_o$ is the set of observable experiments possible from $\sum_j e_j.B_j$, whence by A2, $\tilde{e}_o = \tilde{b}_o$.

(B1 $\Rightarrow$ A1) Immediate.

(B1 and B2 $\Rightarrow$ A2) Let I be the identity relation on agents which are closed in $\mathcal{E}$. It is straightforward to verify that under conditions B1 and B2, the relation

$$I \cup \left\langle \sum_{j=1}^{n} e_j.B_j\ ,\ B \right\rangle$$

is a bisimulation, whence A2 follows. □

Thus, to perform a splitting of $E \approx B$ when B is deterministic, first compute (by the operational semantics in section 2) the transitions from E and B. Then, if condition B2 holds, condition B1 gives the agents B_j. If B2 does not hold, no splitting transformation is applicable.

A simple example might be illuminating at this point: assume that we want to solve

$$(a.b.NIL|X)\backslash\{b\} \approx a.c.NIL$$

In the following, we write tableaux as boxes with goals to the left and environments to the right. For this particular example, we write $\|$ for $\|_{\{b\}}$. Hence, the original tableau is:

$a.b.NIL \parallel X \approx a.c.NIL$	$\emptyset$

Only an instantiation transformation is applicable here. By the heuristics (to be described in section 6), X should have an initial $\bar{b}$ action to match the potential b transition in $a.b.NIL$. Instantiating X to $\bar{b}.Y$ gives

$a.b.NIL \parallel X \approx a.c.NIL$	$X \mapsto \bar{b}.Y$

The left hand side of the equation is now well guarded. Both sides of the equation can initially do an a transition: for the left hand side the result is "$b.NIL \parallel X$", and for the right hand side the

result is "$c.NIL$". The premises of the splitting rule are satisfied, and as a result of the splitting we get

$b.NIL \parallel X \approx c.NIL$	$X \mapsto \bar{b}.Y$

The left hand side is still well guarded, and can now only do a τ transition, resulting in "$NIL \parallel Y$". By another splitting:

$NIL \parallel Y \approx c.NIL$	$X \mapsto \bar{b}.Y$

Of course, it is now easy to see that $Y = c.NIL$ will solve this goal. Following our method strictly, it is time to supply the initial actions of Y. By the heuristics, Y should have an initial c to conform with the initial c transition in the right hand side:

$NIL \parallel Y \approx c.NIL$	$X \mapsto \bar{b}.Y$ $Y \mapsto c.Z$

By a splitting we get

$NIL \parallel Z \approx NIL$	$X \mapsto \bar{b}.Y$ $Y \mapsto c.Z$

After instantiating Z to NIL, and applying a final splitting transformation we get

$NIL \approx NIL$	$X \mapsto \bar{b}.Y$ $Y \mapsto c.Z$ $Z \mapsto NIL$

The equivalence transformation removes the last equation; we get the goal *true* and the environment contains the desired solution, which can be written $\bar{b}.c.NIL$.

There seems to be a fair amount of tedium in applying the tableau method even to simple problems. Our point is that this tedium can be automated. Indeed, the program described in section 6 will do the above steps automatically.

To demonstrate the soundness of this tableau method (i.e. that when the goal *true* is reached, the environment will make the original goal true), we prove that $\longrightarrow$ is safe, and appeal to proposition 2.

Proposition 4 $\longrightarrow$ *is safe.*

Proof: For the instantiation and equivalence transformations this is obvious. For the splitting transformation, if $E \xrightarrow{e_j} E_j$ are the transitions from E and E is well guarded, then $E \approx \sum_j e_j.E_j$. Also, by congruence properties of $\approx$,

$$\bigwedge_{j=1}^{n}(E_j \approx B_j) \quad \text{implies} \quad \left(\sum_{j=1}^{n} e_j.E_j\right) \approx \left(\sum_{j=1}^{n} e_j.B_j\right)$$

Thus, from $\bigwedge_j E_j \approx B_j$ and the premises of the splitting rule we infer $E \approx B$, i.e. the goal is made stronger by replacing $E \approx B$ with $\bigwedge_j E_j \approx B_j$. □

For finite agents, $\longrightarrow$ is complete in the following sense: starting with any satisfiable equation $A \parallel X \approx B$ where A and B are finite, the goal *true* can eventually be reached:

Proposition 5 (Completeness) *Let Γ be a satisfiable goal $A \parallel X \approx B$ where A and B are finite agents. Then, there exists an environment $\mathcal{F}$ such that*

$$\langle \Gamma, \emptyset \rangle \longrightarrow^* \langle true, \mathcal{F} \rangle$$

Proof: See the proof of the related proposition 7. □

5 Finite State Agents

Obviously, with the tableau method in the previous section, it is impossible to generate environments with recursively defined identifiers. The following extension of the instantiation transformation will amend this situation:

- *Instantiation by identification*: If $\langle \Gamma, \mathcal{E} \rangle$ is a tableau with $A \parallel X \approx B$ in the goal, X is free in $\mathcal{E}$, and Y is an identifier bound by $\mathcal{E}$, then

$$\langle X \mapsto Y \rangle$$

  can be added to $\mathcal{E}$.

Thus, it is possible to "identify" a free identifier with a bound identifier. Clearly, the extended tableau transformation is still safe. Instead of adding $\langle X \mapsto Y \rangle$ to the environment, we can uniformly substitute X by Y in the tableau — this has the same effect in terms of the $\rightarrow_{\mathcal{E}}$ relations, and hence does not affect observation equivalence w.r.t. $\mathcal{E}$.

A simple example will illustrate how the tableau method is used. Let $A \mapsto a.A$ and $B \mapsto a.B + b.B$ be an environment and assume we want to find an X satisfying $A|X \approx B$. The initial tableau is (for convenience, we do not show $A \mapsto a.A$ and $B \mapsto a.B + b.B$; these are tacitly present in the environment):

$A\|X \approx B$	$\emptyset$

According to the heuristics, X should have an initial b action to conform with the initial b in B. By instantiating X to $b.Y$ we get

$A\|X \approx B$	$X \mapsto b.Y$

Now the left hand side is well guarded. Both sides of the equation can do a and b transitions; after a splitting we get

$A\|X \approx B$ $A\|Y \approx B$	$X \mapsto b.Y$

The first equation in the goal is identical with the original equation. In the second equation Y is free, and should be instantiated. The heuristics suggest that Y should be identified with X, since they are in the same equations. Identifying Y with X yields:

$A\|X \approx B$ $A\|X \approx B$	$X \mapsto b.X$

There are now no free identifiers in the tableau. The goal contains two (identical) equations, these are true in the environment and can be removed. Hence, "$X \mapsto b.X$" is the desired solution.

In the rest of this section we will prove that the tableau transformation is complete. The following lemma is crucial. It says that if an equation is in a form suitable for a splitting transformation, i.e. the left hand side is well guarded, then it can always be subjected to a splitting transformation which preserves satisfiability.

Lemma 6 *Let $\langle \Gamma, \mathcal{E} \rangle$ be a satisfiable tableau, i.e. there exists an extension $\mathcal{F}$ of $\mathcal{E}$ such that $\mathcal{F} \models \Gamma$. If an equation in Γ is $E \approx B$, where E is well guarded and B is closed in $\mathcal{E}$, and the initial transitions from E are $E \xrightarrow{e_j} E_j$ for $j \in [1, \ldots, n]$, then there exist agents $B_1, \ldots, B_n$ such that*

$$\text{for all } j\text{: } B \stackrel{\widehat{e_j}}{\Longrightarrow}_{\mathcal{E}} B_j$$

and

$$\mathcal{E} \models \left(\sum_{j=1}^{n} e_j.B_j\right) \approx B$$

and

$$\mathcal{F} \models \bigwedge_{j=1}^{n} (E_j \approx B_j)$$

Proof: In all extensions of $\mathcal{E}$ we have that $E \stackrel{\widehat{e_j}}{\Longrightarrow} E_j$; in particular this must hold in $\mathcal{F}$. By the premises, $E \approx B$ is true in $\mathcal{F}$. Thus, for all $j = 1, \ldots, n$ the following diagram (where we write $\Updownarrow$ for $\approx$ in environment $\mathcal{F}$)

$$\begin{array}{ll} E & \stackrel{\widehat{e_j}}{\Longrightarrow}_{\mathcal{F}} \quad E_j \\ \Updownarrow & \\ B & \end{array}$$

can be completed with agents B_j to a commuting diagram

$$\begin{array}{lll} E & \stackrel{\widehat{e_j}}{\Longrightarrow}_{\mathcal{F}} & E_j \\ \Updownarrow & & \Updownarrow \\ B & \stackrel{\widehat{e_j}}{\Longrightarrow}_{\mathcal{F}} & B_j \end{array}$$

This implies that

$$\mathcal{F} \models \bigwedge_{j=1}^{n} (E_j \approx B_j)$$

Since B is closed in $\mathcal{E}$, $B \stackrel{\widehat{e_j}}{\Longrightarrow}_{\mathcal{F}} B_j$ implies $B \stackrel{\widehat{e_j}}{\Longrightarrow}_{\mathcal{E}} B_j$. By congruence properties of $\approx$ with respect to guarded sum, we get (w.r.t. $\mathcal{F}$):

$$\sum_{j=1}^{n} e_j.B_j \approx \sum_{j=1}^{n} e_j.E_j \approx E \approx B$$

However, since B and all B_j are closed under $\mathcal{E}$, this implies that w.r.t. $\mathcal{E}$:

$$\sum_{j=1}^{n} e_j.B_j \approx B$$

□

For the completeness result we make the following definitions: an agent is in *parallel form* iff it is in the form $A \parallel X$ where X is an identifier. An equation $E \approx B$ is in parallel form iff its left hand side E is in parallel form, and its right hand side B is closed. The completeness result is that if a satisfiable goal consists of a finite number of equations in parallel form, then the goal *true* can be derived:

Proposition 7 (Completeness) *Let $\langle \Gamma, \mathcal{E} \rangle$ be a satisfiable tableau where Γ consists of finitely many equations in parallel form, and all agents in the equations are finite state. Then, there exists an environment $\mathcal{F}$ such that*

$$\langle \Gamma, \mathcal{E} \rangle \longrightarrow^* \langle \mathit{true}, \mathcal{F} \rangle$$

Proof: The following proof outlines the algorithm behind our implementation of the tableau method described in section 6.

Since all involved agents are finite state, and the goal is satisfiable, it must be satisfiable by a finite environment[1]. Call this environment $\mathcal{F}$. Such an environment can be defined by a *finite* number, call it n, of applications of the instantiation transformation.

Apply induction on n = the number of necessary applications of the instantiation transformation. For $n = 0$, no instantiations are necessary. Hence, $\mathcal{F} = \mathcal{E}$ and Γ is true of $\mathcal{E}$, whence all equations can be removed by the equivalence transformation.

For the inductive step, $n \geq 1$, i.e. at least one instantiation is necessary. We assume the proposition for $n - 1$, and will prove it for n. Perform the following procedure:

1. Distinguish between marked and unmarked equations in Γ. Originally, all equations are unmarked.

2. For each unmarked equation, such that the identifier to the right of $\|$ is bound by $\mathcal{E}$, do the following steps:

   (a) Apply a splitting transformation to the equation. This is always possible, since if an identifier is bound, it is well guarded (this follows from the form of the instantiation transformations), and hence lemma 6 applies. The result of the splitting will be a finite set of equations in parallel form.

   (b) Mark each resulting equation which has been treated before by this procedure (this requires remembering all equations treated by the procedure).

3. Perform the steps under 2 repeatedly until there are no more unmarked equations with a bound identifier to the right of $\|$. This will eventually happen: since all involved agents are finite state, only a finite number of different equations can be generated with splitting transformations.

The resulting goal still is satisfiable by $\mathcal{F}$, and still consists of a finite number of equations in parallel form. There are now two different cases:

1. There are no unmarked equations left at all. There may still be free identifiers in the goal, but their instantiation will not matter for the truth of Γ, since they will never be exercised when determining possible transitions. Hence, if Γ is true of one extension of $\mathcal{E}$, it is true of all extensions. But Γ is true of $\mathcal{F}$, thus it is true of all extensions, hence also of $\mathcal{E}$. This contradicts the assumption that at least one more instantiation is necessary.

2. There is at least one unmarked equation left. This equation must have a free identifier to the right of $\|$. Thus, it is possible to apply an instantiation transformation to the goal. By choosing the transformation in accordance with $\mathcal{F}$, the goal can now be satisfied by the rest of $\mathcal{F}$, i.e. by $n - 1$ applications of instantiation. By induction, the proposition follows. □

Finally, it can be proven that if $A \parallel X \approx B$ has a solution, then it has a solution bounded in size by the sizes of A and B. Hence, it is decidable whether it has a solution or not.

6 An Implementation

Our program for solving equations with the tableau method works in the following way: first, the user enters the equation $A \parallel_L X \approx B$ that he wants to solve. A and B must be finite state, and B must be deterministic. Also, the expected sort of the solution must be given (alternatively, the program will compute an expected sort). Thereafter, a semi automatic procedure will start

[1] A finite set of recursive equations with guarded sum is sufficient to express any finite state agent; see eg. [Mil84].

as outlined in the proof of proposition 7: by treating all equations with splitting transformations until no new equations are generated.

The program does all splitting transformations automatically. When a free identifier X is to be instantiated, the program permits the user to identify X with a previously bound identifier, or guess the initial actions of X. When the program discovers that the goal is unsatisfiable (e.g. by finding it impossible to perform a splitting transformation) it backtracks to the preceding instantiation.

The practical use of such a program would be small, were it not for the fact that good heuristics for the instantiations can be presented. At each instantiation the user decides whether to follow or disregard the heuristics, and can thus avoid solutions which would not be suitable for implementation. The user can explore different possibilities and backtrack at will. He can even run the program in an automatic mode, where all instantiations are made according to the heuristics.

The most important information presented to the user when a free identifier X is to be instantiated are the sets of *admissible* actions and *useful* actions. An action a is *inadmissible* for X if there is an equation $A \parallel X \approx B$ such that if X were instantiated with an initial action a, then the equation would be unsatisfiable. Inadmissibility is in general hard to check, but it can be approximated by k-inadmissibility as follows: for an agent A, let $Tr_k(A)$ be the set of traces (transition sequences where τ transitions have been deleted) of length $\leq k$. An action a is k-inadmissible for X if $Tr_k(A \parallel a.NIL) \not\subseteq Tr_k(B)$. This means that if X were instantiated with an initial action a, then there would be traces (of length k) of $A \parallel X$ which are not traces of B. For every k, k-inadmissibility implies inadmissibility. When determining which actions are admissible, the program tries each action for k-inadmissibility up to some predetermined maximum value of k. The higher maximum value of k, the more accurate the admissibility test, and the more computation time is spent in determining admissibility. The user can interactively modify this maximum value.

In the equation $A \parallel_L X \approx B$, an action a is *useful* for X, if there is a transition of a derivative of $A \parallel_L X$ which depends on the fact that X can do an initial a transition. The useful actions can be computed as follows: say that a is *covered* by L if $a \in L$ or $\bar{a} \in L$. Then, all actions not covered by the restriction L are useful for X. Furthermore, an action a is useful for X if A can perform a transition sequence, not containing any actions covered by L, but containing $\bar{a}$ — in this case, an initial a in X can result in a synchronisation with this $\bar{a}$ in A.

As an example of these concepts, consider the equation (from section 4)

$$a.b.NIL \parallel_{\{b\}} X \approx a.c.NIL$$

Here, the action $\bar{b}$ is useful for X (it can result in a synchronisation with b). Also, $\bar{b}$ is admissible. The action c is useful, but not admissible — in fact, it is even 1-inadmissible.

Instead of guessing the initial moves of X, the user can decide to identify X with a bound identifier. At each choice, the program supplies the user with a list of adequate bound identifiers. An identifier Y is *adequate* for X if the equations containing X constitute a subset of the equations containing Y, and the admissible and useful actions of X agree with the initial actions of Y. The intuition is that this is a strong indication that X could successfully be identified with Y: a solution for Y will always also be a solution for X.

A simple example will illustrate the program. Assume that we seek an agent X, which in parallel with a buffer of capacity one yields a buffer of capacity two. A buffer of capacity one on channels a and b is defined by

$$A \mapsto a.b.A$$

and a buffer of capacity two on channels a and c is

$$\begin{cases} B \mapsto a.B' \\ B' \mapsto c.B + a.c.B' \end{cases}$$

The user also has to supply the restriction L for solving $A \parallel_L X \approx B$, in this case the restriction is $\{b\}$. The program infers a sort for the solution, in this case it is $\{\bar{b}, c\}$, and the user acknowledges this.

Now the tableau method begins. We will here display the choice points as tableaux: to the left are equations containing the free identifier under consideration (note that the complete tableau contains more equations that are not immediately relevant for this identifier), in the middle is the solution generated so far, and to the right are the recommendations of the program. The first choice point is:

$A \parallel X \approx B$		No adequate bound identifiers Admissible and useful actions: $\{\bar{b}\}$

The user is presented with a menu of various alternatives; in this case he chooses to instantiate X according to the recommendation (the only initial action is $\bar{b}$) The next choice point is:

$A \parallel X_1 \approx B'$	$X \mapsto \bar{b}.X_1$	No adequate bound identifiers Admissible and useful actions: $\{c\}$

Again, following the recommendation leads to the next choice point:

$A \parallel X_2 \approx B$ $b.A \parallel X_2 \approx B'$	$X \mapsto \bar{b}.X_1$ $X_1 \mapsto c.X_2$	Adequate bound identifiers: X Admissible and useful actions: $\{\bar{b}\}$

Now, following the recommendation to identify X_2 with X, the program discovers that there are no more free identifiers, and proceeds to check the environment against the goal[2]. In this case, the environment satisfies the goal, and the program reports the solution to the user:

$$\begin{cases} X \mapsto \bar{b}.X_1 \\ X_1 \mapsto c.X \end{cases}$$

The solution can be written $X \mapsto \bar{b}.c.X$, i.e. it defines as expected a buffer of capacity one.

When run in the automatic mode, the program resolves the instantiations according to the *maximal* strategy:

1. If there is at least one adequate bound identifier, then identify with one of them.
2. If there are no adequate bound identifiers, then instantiate with the set of all admissible and useful actions.

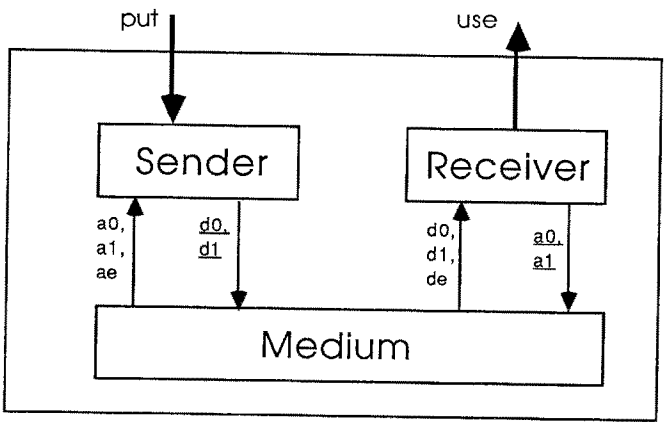
The strategy is called "maximal" because the solutions will in general be agents that have maximal freedom: if more transitions are added, then they would either cause inadmissible behaviour, or would never be exercised. Maximality might, or might not, be a sensible criterion for good solutions. For most small examples, such as those presented in this paper (excepting section 7), the strategy produces the expected solutions automatically. It should be noted that the maximal strategy is not complete for satisfiable goals: it sometimes results in a diverging sequence of choices.

7 The Alternating-Bit Protocol

In this section we study the effects of applying the the maximal strategy of the program to a nontrivial example: the alternating-bit protocol.

The purpose of the alternating-bit protocol (originally presented in [BSW69]) is to provide reliable data transmission over an imperfect medium. Figure 1 shows the general structure of the protocol. It consists of three modules: a sender, a medium and a receiver. There are several versions of this protocol; we will begin by studying the protocol as presented in [MB83]. There,

[2]Actually, it checks against the original goal rather than the current goal; this has turned out to be more efficient in practice.



Notation:

put / *use* submitting/receiving a message to/from protocol
$\underline{d_i}$ / d_i transmitting/receiving a message to/from medium
$\underline{a_i}$ / a_i transmitting/receiving an acknowledgment to/from medium
d_e / a_e receiving corrupt message/acknowledgment from medium

Figure 1: The structure of the modules in the alternating-bit protocol.

the medium can corrupt but not lose messages. A message is delivered to the sender through the primitive *put*, and accepted from the receiver through the primitive *use*. The service of the protocol is that of a perfect one place buffer, i.e. *put* and *use* alternate:

$$Service \mapsto put.use.Service$$

Figure 2 depicts state transition diagrams for the modules in the protocol. We will in this section consistently use such diagrams to represent agents; the transformation between diagrams and a system of recursive agent identifier definitions is trivial.

The protocol works as follows. The sender adds a one bit sequence number to an incoming message (starting with 0 for the first message) and transmits it to the medium. We will not explicitly represent message contents, but the sequence numbers are important for the synchronisation properties of the protocol. Thus, we use $\underline{d_0}$ to represent transmission of messages with sequence number 0, and $\underline{d_1}$ for messages with sequence number 1. Following a transmission, the sender awaits an acknowledgment (actions a_0 and a_1) with the same sequence number. After reception of the correct acknowledgment, the procedure is repeated: a new message can be accepted for transmission. This time the sequence number is inverted. If the sender receives an acknowledgment with wrong sequence number, or a corrupt acknowledgment (action a_e), then it retransmits the last message.

Our model of the sender differs from that in [MB83] in one respect: in the states where acknowledgments are not expected, the sender may accept and discard spurious acknowledgments.

The receiver acknowledges all messages (d_0, d_1) by transmitting an acknowledgment with the same sequence number as the message $(\underline{a_0}, \underline{a_1})$. If the sequence number differs from the preceding one, then the message is not a retransmission, and is delivered to the user through the primitive *use*. If a corrupt message arrives (d_e), then the last acknowledgment is retransmitted.

The medium can contain at most one message or acknowledgment at a time, i.e. it is half duplex. Thus, following an action $\underline{d_i}$ (the inverse of $\underline{d_i}$), it either delivers the message through $\overline{d_i}$

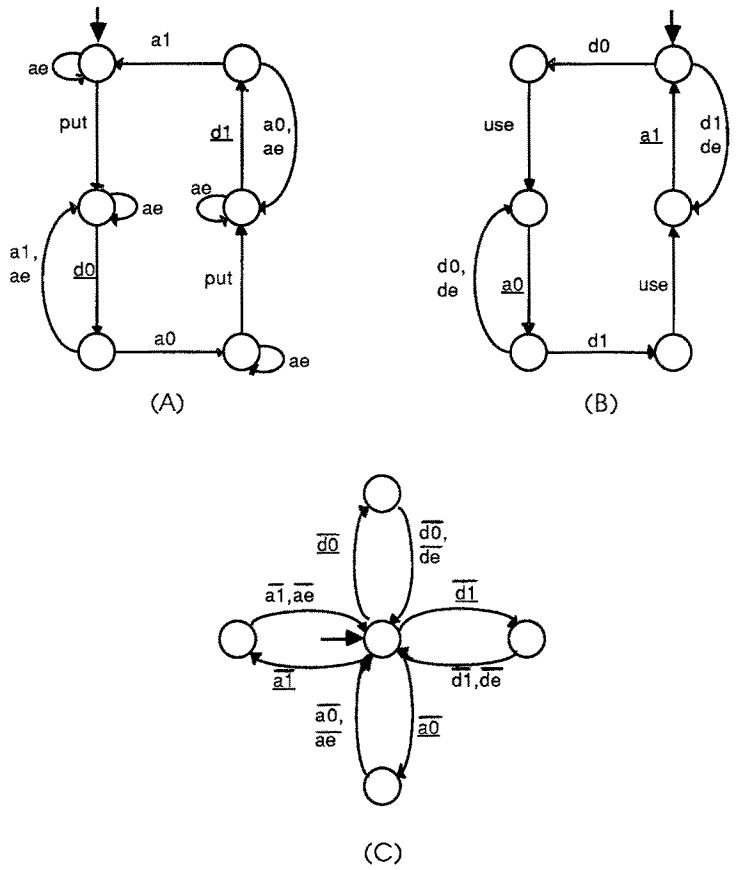


Figure 2: The modules in the alternating bit protocol. (A) The Sender. (B) The Receiver. (C) The Medium.

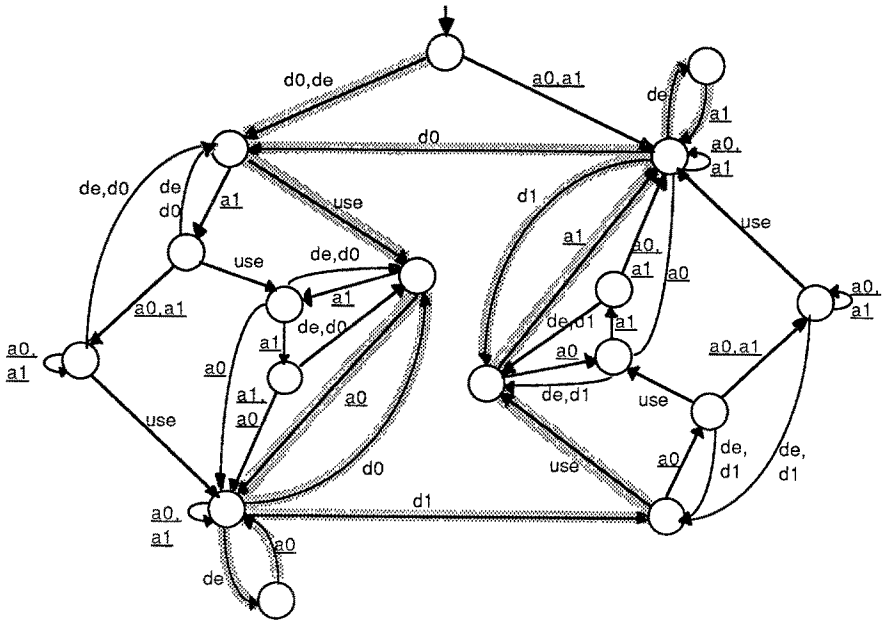


Figure 3: A most general receiver.

or delivers a corrupt message through $\overline{d_e}$. Similarly, acknowledgments may be corrupted.

The program for equation solving can be used to generated any one unknown module of the protocol. As an example, we have generated the receiver by solving the equation

$$(Sender|Medium) \parallel X \approx Service$$

Here, $\parallel$ means parallel composition and restriction over the internal actions ($\underline{d_i}$ and $\underline{a_i}$ for $i = 0$ and 1; d_i and a_i for $i = 0$, 1, and e). When applying the maximal strategy to solve the equation, the result is the rather surprising receiver in figure 3.

This receiver is a most general receiver in the sense that from any state, additional actions will either never be exercised or will lead to inadmissible behaviour of the protocol. It is clear that it is much more general than the expected solution in figure 2. For example, in the initial state, the receiver may begin by transmitting any sequence of acknowledgments. Of course, in a real implementation this would be ridiculous. Nevertheless, the receiver satisfies the formal problem. Indeed, with the medium being half duplex and of capacity one element, these extra acknowledgments are harmless: when the receiver has not accepted any message and it transmits an acknowledgment, then no message has been sent, and hence the sender is in a state where it discards the incoming acknowledgments. There are other similar paradoxical aspects of the behaviour of this receiver. Note, however, that the expected solution is contained as a subgraph (highlighted transitions). We take this example as a good illustration of our point: a completely automatic procedure for generating submodule behaviours is not always desirable.

A variation on the alternating-bit protocol is to use a full-duplex medium, with capacity one element in each direction, and ability to lose messages. For simplicity, we assume that messages are either lost or delivered intact (this is a realistic assumption; there might be an error detection mechanism that discards all corrupt messages). The medium can be modelled as the parallel composition of two independent simplex media, where τ actions correspond to message loss. The sender and receiver modules are modified by deleting all transitions dealing with corrupt messages (d_e and a_e), and by adding timeout transitions to the sender (τ transitions leading from states where the sender waits for acknowledgments to states where it can do retransmissions). These modules are shown in figure 4.

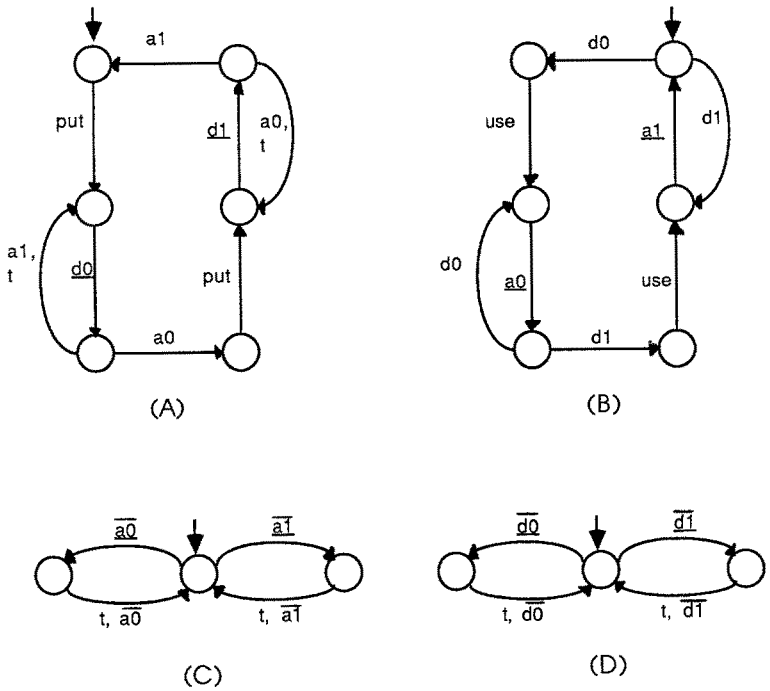


Figure 4: Second version of the modules in the alternating-bit protocol. (A) The Sender. (B) The Receiver. (C, D) Simplex media. The Medium in the protocol consists of these two media in parallel.

Again, using the maximal strategy to solve the equation where the receiver is unknown, yields a most general solution as shown in figure 5. This solution does not depart very much from the expected solution. The initial state is unreachable from the other states (in the initial state there is no useful d_1 action), and in all states it is harmless to retransmit the last acknowledgment or accept a duplicate of the last accepted message.

In a similar way, and with similar results, the sender of the protocol can be generated when the receiver is known. It is even possible to generate a medium if both sender and receiver are known. A most general medium for the protocol in figure 4 is depicted in figure 7. Naturally, it is unlikely that the medium is unknown in a real protocol design project. Instead, this result indicates the worst possible conditions under which the protocol will work. As can be seen in figure 7, the medium may not only lose messages, but also generate spurious messages in certain situations, without harming the protocol.

8 Conclusions and Comparisons with Related Work

We have in this paper indicated one way to give meaning to CCS equations of type $(A|X)\backslash L \approx B$, and presented a method for solving such equations. The method is based on a general tableau framework. Within this framework, we have formulated the transformations necessary for deriving solutions. These transformations form a basis for an implementation, which has been applied to generating the receiver of different versions of the alternating-bit protocol.

Our experience is that a completely automatic procedure for solving equations is not always desirable. Typically, an equation $(A|X)\backslash L \approx B$ has many solutions. Even if it has a unique most general solution (i.e. a solution which simulates every other solution), it is not certain that this solution is suitable for implementation. Thus, when generating a solution, some criteria for what

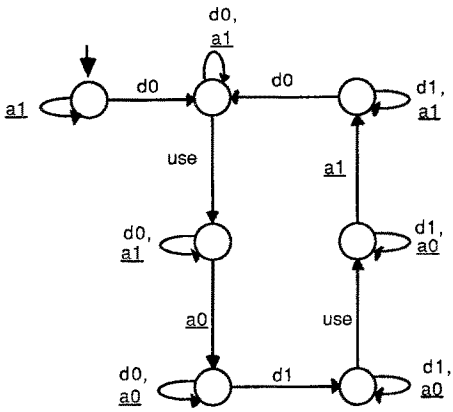


Figure 5: A most general receiver of the second version of the protocol.

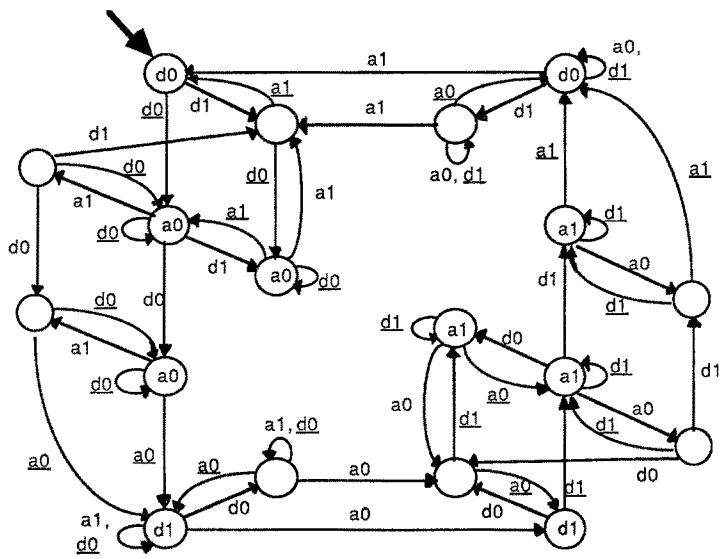


Figure 6: A most general medium of the second version of the protocol. To improve readability we have omitted the bars denoting inverted actions. In contrast with the previous solutions we here also indicate the non useful transitions. From each unlabelled state, actions which are not already labels of (useful) transitions are labels of non useful transitions. The same convention holds for labelled states, with the exception that the label of the state is not a label of a non useful transition. Since such transitions will never be exercised, their target states are unimportant.

constitutes a good solution must be used. Obviously, such criteria are dependent on the particular equation to be solved. With our method, 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.

One interesting way to extend this work is to consider a larger class of equations. Already, the method is sufficiently powerful to treat several equations simultaneously. Hence, it can be used to solve problems as "find an X such that $A_1 \parallel X \approx B_1$ and also $A_2 \parallel X \approx B_2$". Similarly, the method could handle nonlinear equations (e.g. "$X \parallel X \approx B$") or even systems of nonlinear equations (e.g. "find X and Y such that $X \parallel X \parallel Y \approx B$ and $X \parallel Y \parallel Y \approx C$"). However, for the nonlinear case our methods for determining admissible and useful actions would not apply.

Another way to extend the scope of this method is to consider other operators and other types of equivalences. For example, in TCSP ([BHR84]) there are other types of parallel operators, other types of nondeterministic choices, and a different equivalence relation. Also, the testing equivalences from [NH84] could be used in this context. We conjecture that the tableau method would work well also in these systems. A congruence property of guarded sum would be sufficient for a sound splitting transformation. For a completeness result, a counterpart of lemma 6 is needed.

It would be exciting to extend our method to include communication with value passing. The tableau transformation is easily extended by including events with value parameters and parametrised identifiers in the instantiations. The difficulty would be to provide good heuristics for choosing the value expressions in the output events.

Since there is a vast literature on generating modules of complex systems, we will here only comment on some approaches related to our method. To our knowledge, the only work on solving CCS equations is [Shi86b] and [Shi86a]. There, equations of type $(A|X)\backslash L \approx B$ are called "interface equations". For the case that B is deterministic, and under some requirements (not very restrictive) on the sorts of A and B, necessary and sufficient conditions for the existence of solutions of such equations are given. In the case that there exist solutions, an explicit construction of a solution is presented. This construction, and also the requirements for existence of a solution, are formulated in terms of the state spaces of A and B. There is, however, no indication that this method can be used interactively and guided towards solutions which are suitable for implementation.

We have already mentioned the work in [MB83]. There, a similar problem is considered with finite automata instead of agents, and trace equivalence instead of observation equivalence. Also, the definition of parallel composition is slightly different in that the simultaneous execution of two actions does not always result in an unobservable action. Within this formalism, the authors derive a solution in terms of the "complement" operation on automata (the complement of an automaton A accepts the complement of the language accepted by A). They apply this method to generate the receiver of the alternating-bit protocol, and remark that the most general solution is not always the best one. Their suggested remedy is to start by generating a most general solution, and proceed by deleting states and transitions which are unnecessary (i.e. can be deleted while preserving trace equivalence of the system). Also, they remark that trace equivalence is not sufficient to demonstrate properties like deadlock freedom.

The recent [BG86] goes one step further. There, the authors present a method to automatically partition an overall system behaviour B into submodules $A_1, \ldots, A_n$. These submodules, when composed in parallel, yields a behaviour which is trace equivalent with B. The idea is to partition the set of actions in B into different locations, and generate one module A_i for each location. The method assumes that the modules communicate over unbounded perfect channels.

In [ZWR*80], a semi automatic procedure is given on how to complete partly specified modules into a system which will be guaranteed to be free of certain unwanted properties such as deadlocks. In [GY84], an algorithm is presented for generating one module of a protocol when a second module is given. However, in neither of these is there any formal specification of the expected service of the combined system. Algorithms for synthesis of concurrent programs from service specifications in

temporal logic are presented in [EC82] and [MW84]. A new direction is taken in [APP86]. There, specifications are formulated in knowledge logic (where assertions can be of type "module A knows the contents of message m").

Our method is based on transformation of tableaux. The main inspiration for this is [MW80], where (sequential) functional programs are generated in a similar way by transforming predicate logic formulas. Later, this idea was extended to synthesis of asynchronously communicating networks ([JMW86]). The approach is to first generate one single module, defined as a functional program, and subsequently transform this module into several modules working in parallel. This transformation is specifically aimed at generating dataflow networks.

Acknowledgments

I am grateful to Bengt Jonsson, Robin Milner, Colin Stirling, and David Walker for a critical reading of the manuscript. This work was carried out under a grant from the British Science and Engineering Research Council

References

[APP86] Foto Afrati, Christos Papadimitriou, and George Papageorgiou. The synthesis of communication protocols. In *Proceedings of the fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 263–271, 1986.

[BG86] Gregor von Bochmann and Reinhard Gotzhein. Deriving protocol specifications from service specifications. In *Proceedings of the ACM SIGCOM Symposium*, pages 148–156, 1986.

[BHR84] S. Brookes, C.A.R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BSW69] K Bartlett, R Scantlebury, and P Wilkinson. A note on reliable full-duplex transmissions over half duplex lines. *Communications of the ACM*, 2(5):260–261, 1969.

[EC82] E. Emerson and E. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[GY84] Mohamed Gouda and Yao-Tin Yu. Synthesis of communicating finite-state machines with guaranteed progress. *IEEE Transactions on Communications*, COM-32(7):779–788, 1984.

[JMW86] Bengt Jonsson, Zohar Manna, and Richard Waldinger. Towards deductive synthesis of dataflow networks. In *Proceedings of Symposium on Logic in Computer Science*, pages 26–37, 1986.

[MB83] Philip Merlin and Gregor von Bochmann. On the construction of submodule specifications and communication protocols. *ACM Transactions on Programming Languages and Systems*, 5(1):1–25, 1983.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes of Computer Science*, Springer Verlag, 1980.

[Mil84] Robin Milner. A complete inference system for a class of regular behaviours. *J. of Computer System Science*, 28:439–466, 1984.

[MW80] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1), 1980.

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[NH84] R de Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Shi86a] M W Shields. *Extending the Interface Equation.* Technical Report SE/079/3, Electronic Engineering Laboratories, University of Kent at Canterbury, August 1986.

[Shi86b] M W Shields. *Solving the Interface Equation.* Technical Report SE/079/2, Electronic Engineering Laboratories, University of Kent at Canterbury, July 1986.

[ZWR*80] P Zafiropulo, C H West, H Rudin, D D Cowan, and D Brand. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications*, COM-28(4):651–661, 1980.