Using Logic to Solve the Submodule Construction Problem

GREGOR V. BOCHMANN
School of Information Technology and Engineering (SITE), University of Ottawa, Canada

Note: Some very preliminary version of this paper was written in September 2001 (not published); it was completely rewritten Fall 2008 and a small subset was published in FORTE 2009 [Bochmann 2009]. This paper is a largely extended version of the 2008 paper. This work was partly supported by a research grant from the Natural Sciences and Engineering Research Council of Canada.

Abstract: Submodule construction is the problem of finding a new submodule which, together with a given submodule, provides a behavior that conforms to a given desired global behavior. A new formulation of this problem and its solution in first-order logic is presented, and it is shown how the known solutions to this problem in the context of various communication paradigms and specification formalisms can be derived. Communication paradigms are: synchronous rendezvous at several interfaces; interleaved rendezvous; input/output automata with complete or partial behavior specifications and with synchronous or interleaved communication. A new algorithm for deriving a progressive solution is also presented.

Categories and Subject Descriptors: D.2.1 [**Software engineering**]: Requirements/Specifications
Additional Key Words and Phrases: component design, equation solving, submodule construction, derivation of component behavior, state machines, labeled transition systems, input/output automata, first-order logic, discrete event control systems

---

1. Introduction

In automata theory, the notion of constructing a product machine S from two given finite state machines M_A and M_B , written $M = M_A \times M_B$, is a well-known concept (see Figure 1(a)). This notion is very important in practice since complex systems are usually constructed as a composition of smaller subsystems, and the behavior of the overall system is in many cases equal to the composition obtained by calculating the product of the behaviors of the two subsystems. Here we consider the inverse operation, called "equation solving" or "submodule construction": Given the behavior of the composed system M and one of the components M_A, what should be the behavior of the second component M_B such that the composition of these two components M_A and M_B will exhibit a behavior desired for M. That is, we are looking for the value of X which is the solution to the equation $M_A \times X = M$ (see Figure 1(b)). This problem is an analogy of integer division, which provides the solution to the equation N1 * X = N for integer values N1 and N. In integer arithmetic, there is in general no exact solution to this equation; therefore integer division provides the largest integer which multiplied with N1 is smaller than N. Similarly, in the case of equation solving for machine composition, we are looking for the most general machine X which composed with M_A satisfies some conformance relation in respect to M. In the simplest case, this conformance relation is trace inclusion.

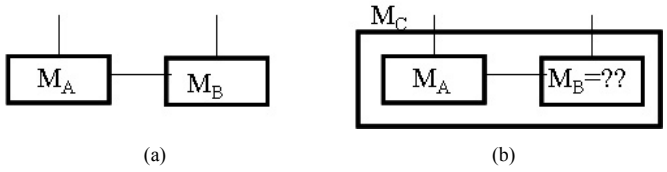


Figure 1: (a) two communicating components; (b) submodule construction problem

A first paper [Bochmann and Merlin, 1980] gives a solution to this problem for the case where the machine behavior is described in terms of labeled transition systems (LTS) which communicate with one another by interleaved rendezvous interactions (see also [Hagverdi et al. 1999] for a more formal treatment). This work was later extended to the cases where the behavior of the machines is described in CSP [Hoare, 1985] (with behavioral equivalence as conformance relation) [Parrow, 1989] and for process algebras with bisimulation equivalence as conformance relation [Larsen 1990]. In the context of state machines, solutions have also been described for finite state machines (FSM) communicating through message queues [Petrenko et al. 1998, Yevtushenko 2008], by input/output automata (IOA) [Qin 1991, Drissi 1999, Bochmann 2002, Bhaduri 2008], and by synchronous finite state machines [Kim 1997, Yevtushenko 2000]. The case of extended state machine models including state variables and assertions about input and output parameters has also been studied [Daou 2005]. The problem has also been formulated for databases using relational algebra [Bochmann 2002a].

One application of this submodule construction method was considered in the context of the design of communication protocols, where the components M_A and M_B may represent two protocol entities that communicate with one another [Bochmann and Merlin, 1980]. Later it was recognized that this method could also be useful for the design of protocol converters in communication gateways [Kelekar 1994, Tao 1997, Kumar 1997], and for the selection of test cases for testing a module in a given context [Petrenko 1996]. It is expected that it could also be used in other application domains where the re-use of components is important. If the specification of the desired system is given together with the specification of a module to be used as one component in the system, then equation solving provides the specification of a new component to be combined with the existing one.

Independently, the same problem was identified in control theory for discrete event systems [Ramadge and Wonham 1989] as the problem of finding a controller for a given system to be controlled. In this context, the specification M_A of the system to be controlled is given, as well as the specification of certain properties that the overall system, including the controller, should satisfy. If these properties are described by M, and the behavior of the controller is X, then we are looking for the behavior of X such that the equation M_A x X = M is satisfied. Solutions to this problem are also described in [Brandin 1994] using a specification formalism of labeled transition systems where a distinction of input and output is made; interactions of the system to be controlled may be *controllable* (which corresponds to output generated by the controller) or *uncontrollable* (which corresponds to input to the controller). This specification formalism seems to be

equivalent to input/output automata (IOA) [Lynch 1989] or input/output transition systems (IOTS) [Tretmans 1996].

The purpose of this paper is to show that the submodule construction (or equation solving) problem can be formulated in logic. It turns out that (a) a solution exists with a structure similar to the solutions presented in the literature, and (b) a proof of the correctness of this solution in logic is quite simple, apparently much simpler than the existing proofs of correctness for the solutions found in the literature. We show in this paper how the solutions for submodule construction in various contexts can be derived from the solution in the logic context. The proof of correctness from the logic context can therefore be used to justify the particular forms of solutions in the different contexts. These contexts differentiate themselves mainly by the nature of the communication between the different system components. We consider in this paper the following communication paradigms: (a) synchronous rendezvous at several interfaces, (b) interleaved rendezvous (that is, labeled transition systems), (c) synchronous (I/O) automata with complete or partial behavior specifications, (d) interleaving IOA with complete or partial behavior specifications. These contexts include much of the previous work mentioned above and also some not so common modeling approaches, such as synchronous rendezvous at multiple interfaces, and synchronous input/output automata with partial behavior specifications. We do not cover in this paper the context of (e) communicating finite state machines (see [Petrenko 1998, Yevtushenko 2008]), and (f) relational algebras [Bochmann 2002a] which is similar to the logic context. We mention that most of the equations derived in this paper are independent of any particular formalism that may be used for describing the dynamic behavior of the different system components, because our reasoning is based on trace semantics, which means, we are interested in the possible sequences of interactions. In addition, we consider the specification formalism of state machines, that is, regular behaviors, because they allow the definition of algorithms for the solution of the submodule construction problem. In this context, we also go beyond trace semantics and consider the absence of deadlocks and other types of blocking.

The paper is structured as follows: The next section presents the problem of equation solving in the general context of first-order logic. The main concepts and equations are established which are then referenced in the later sections. In Section 3, the submodule construction problem is introduced in the context of modular system design where the overall system is composed out of several components and the behavior of one of the components is to be found. This section deals with synchronous communication between all the components. Several simple examples are introduced in order to explain in detail the different steps of the submodule construction algorithm. Algorithms for deriving a most general solution are given for the case that the given behavior specifications are regular, including the case of prefix-closed behaviors and the elimination of deadlocks. We note, however, that the derived solution equations are valid in general for trace-based specifications. Also a new algorithm is given to find a progressive solution, that is, a

solution that has the same blocking behavior as the desired behavior M, if such a solution exists.

Section 4 shows how the synchronous modeling framework can be used to model interleaving semantics, as used by labeled transition systems (LTS). Although the solution equations and algorithms for regular behaviors with interleaving semantics look similar to the case of synchronous communication, the nature of these systems with interaction interleaving is quite different. Again, a new algorithm for finding a progressive solution is provided. Communication through inputs and outputs is considered in Section 5, and the different cases of complete and partial specifications with synchronous or interleaving communication are considered. Section 6 contains our concluding discussion.

2. EQUATION SOLVING IN THE LOGIC CONTEXT

2.1. The logic context

We use in this section set theory and first-order logic with typed variables. We consider a universe with three variables X_A, X_B, and X_C that may take values from three domains D_A, D_B and D_C, respectively. These domains may be infinite. The set of all possible value assignments to the variables is then $U = D_A \times D_B \times D_C$. We write x_A, x_B, and x_C for possible values of the variables X_A, X_B, and X_C, respectively.

We are interested in relationships between values of different variables. For instance, we may consider a relation $R \subset D_A \times D_B$ which is a subset of pairs $< x_A, x_B >$ of possible values of the variables X_A and X_B. As usual, we use predicates to characterize sets. For instance, the relation R may be characterized by a predicate $C(x_A, x_B)$ which is true exactly for those pairs $< x_A, x_B >$ that are in R. More formally, $R = \{< x_A, x_B > | C(x_A, x_B)\}$. In this paper, we use an abbreviated notation for such a set of pairs: $[C(x_A, x_B)]$ is, by definition, equal to $\{< x_A, x_B > | C(x_A, x_B)\}$.

2.2. The equation solving problem

In the following, we are interested in three relations $R_A \subset D_B \times D_C$, $R_B \subset D_A \times D_C$ and $R_C \subset D_A \times D_B$. We write $C_A(x_B, x_C)$, $C_B(x_A, x_C)$, and $C_C(x_A, x_B)$ for their respective characterizing predicates. We consider the following proposition which relates these three relations:

$$\forall < x_A, x_B, x_C > \in U : \; C_A(x_B, x_C) \wedge C_B(x_A, x_C) \Rightarrow C_C(x_A, x_B) \qquad (1)$$

The problem of equation solving is the following: We assume that C_A and C_C are given. The question is the following: What are the properties of predicate C_B that ensure that proposition (1) is satisfied?

Definition 2-1 (a solution): We say that a predicate C_B is a solution to the equation solving problem if it satisfies equation (1).

Lemma 2-1: The predicate

$$C_B^{max}(x_A, x_C) = \forall x_B \in D_B : \; C_A(x_B, x_C) \Rightarrow C_C(x_A, x_B) \qquad (2)$$

is a solution of the equation solving problem.

The proof if this lemma is trivial. We note that the right side of Equation (2) can be equivalently transformed in several steps as follows:

$$\forall\, x_B \in D_B : \quad \neg C_A(x_B, x_C) \vee C_C(x_A, x_B)$$

$$\forall\, x_B \in D_B : \quad \neg\, (\, C_A(x_B, x_C) \wedge \neg C_C(x_A, x_B)\,)$$

$$\neg\exists\, x_B \in D_B : \; C_A(x_B, x_C) \wedge \neg C_C(x_A, x_B)$$

Therefore, we have the following equivalent expression for this solution:

$$C_B^{max}(x_A, x_C) = \neg\exists\; x_B \in D_B : \; C_A(x_B, x_C) \wedge \neg C_C(x_A, x_B) \qquad (3)$$

Let us now consider any predicate $C_B'(x_A, x_C)$ that is stronger than $C_B^{max}(x_A, x_C)$, that is, $C_B'(x_A, x_C) \Rightarrow C_B^{max}(x_A, x_C)$. Using Equation (2), it is easy to see that such a predicate is also a solution to the equation solving problem. However, any weaker predicate will not be a solution. This can be shown as follows: Let us assume that $C_B^{max}(x_A, x_C) \Rightarrow C_B''(x_A, x_C)$ and $C_B^{max}(x_A, x_C) \neq C_B''(x_A, x_C)$; then there must exist a pair $<x_A, x_C>$ such that $C_B''(x_A, x_C)$ and $\neg C_B^{max}(x_A, x_C)$. Using equation (3), we see that

$$\exists\; x_B \in D_B : \; C_A(x_B, x_C) \wedge \neg C_C(x_A, x_B).$$

Therefore, for the tuple $<x_A, x_B, x_C>$ Equation (1) will not be satisfied.

To summarize, we can state the following proposition:

Proposition 2-1 (maximal solution): The predicates $C_B^{max}(x_A, x_C)$ defined by Equations (2) and (3) are equivalent and define the maximal (i.e. weakest) solution to the equation solving problem and any (stronger) predicate $C_B'(x_A, x_C)$ that satisfies $C_B'(x_A, x_C) \Rightarrow C_B^{max}(x_A, x_C)$ is also a solution.

2.3. The realized subset of $[\, C_C(x_A, x_B)\,]$

We recall that $C_C(x_A, x_B)$ defines the subset $[\, C_C(x_A, x_B)\,]$ of all pairs $<x_A, x_B> \in D_A \times D_B$ that satisfy $C_C(x_A, x_B)$. Given any solution $C_B(x_A, x_C)$ to the equation solving problem, Equation (1) ensures that

$$[\, C_C^{real}(x_A, x_B)\,] = [\, \exists\; x_C \in D_C : \; C_A(x_B, x_C) \wedge C_B(x_A, x_C)\,] \qquad (4)$$

is included in $[\, C_C(x_A, x_B)\,]$.

Definition 2-2 (realized subset of C): The set of pairs $< x_A, x_B >$ defined by Equation (4) is called the **subset** of $[C_C(x_A, x_B)]$ **realized** by the solution $C_B(x_A, x_C)$. The subset realized by the maximal solution is called the **maximally realized subset** of $[C_C(x_A, x_B)]$. We say that a solution is **complete** if the maximally realized subset is equal to $[C_C(x_A, x_B)]$.

Lemma 2-2: Any solution to the equation solving problem that satisfies the predicate

$$C_B^{incomp}(x_A, x_C) = \neg\exists\; x_B \in D_B : \; C_A(x_B, x_C) \wedge C_C(x_A, x_B)$$

has an empty realized subset.

Proof: Let us assume that there is a pair $<x_A, x_B> \in [\, C_C(x_A, x_B)\,]$ that is realized. This implies according to equation (4) that

$$\exists\ x_C \in D_C :\ \ C_A(x_B, x_C)\ \wedge\ C_B^{incom}(x_A, x_C)$$

However, this is a contradiction to the assumption of the Lemma, since $C_B(x_A, x_C)$ – together with $C_A(x_B, x_C)$ - implies $C_C(x_A, x_B)$, because $C_B^{incom}(x_A, x_C)$ is a solution. □

We conclude from Lemma 2-2 that those pairs $<x_A, x_C>$ of any solution $[C_B^{max}]$ that are in $[C_B^{incomp}]$ do not contribute to the realized pairs of $[C_C^{real}]$. We therefore may eliminate those pairs from the solution and still obtain the same realized subset $[C_C^{real}]$. We say that such a solution is **reduced.** It follows that the maximal reduced solution is defined by the predicate $C_B^{red}(x_A, x_C) = C_B^{max}(x_A, x_C)\ \wedge \neg\, C_B^{incomp}(x_A, x_C)$ and we can state the proposition:

Proposition 2-2 (reduced maximal solution): The predicate

$$C_B^{red}(x_A, x_C) = (\ \exists\ x_B \in D_B :\ \ C_A(x_B, x_C)\ \wedge\ C_C(x_A, x_B)\ )\ \wedge$$
$$(\ \neg\exists\ x_B \in D_B :\ \ C_A(x_B, x_C)\ \wedge\ \neg C_C(x_A, x_B)\ ) \qquad (5)$$

defines the reduced maximal solution to the equation solving problem.

2.4. Example

$D_A = \{a_1, a_2\};\ D_B = \{b_1, b_2, b_3\};\ D_C = \{c_1, c_2, c_3, c_4\};$

$R_A = \{<b_1, c_1>, <b_2, c_2>, <b_1, c_3>, <b_2, c_3>, <b_3, c_3>\};$

$R_C = \{<a_1, b_1>, <a_2, b_2>, <a_1, b_3>\};$

The relation corresponding to the predicate $\neg\, C_C(x_A, x_B)$ is the complement of R_C in respect to the set of all tuplets in $D_A \times D_B$ which is sometimes called the "chaos" over $D_A \times D_B$, written $Chaos_{A\times B}$. We have $Chaos_{A\times B} = \{<a_1, b_1>, <a_2, b_2>, <a_1, b_3>, <a_1, b_2>, <a_2, b_1>, <a_2, b_3>\}$, where the last three tuplets are not in R_C.

Using Formula (3), we obtain for the set of tuplets accepted by C_B^{max}

$$[\, C_B^{max}\,] = Chaos_{A\times C} \setminus \{<a_i, c_j> \mid \exists\ b_k :\ <b_k, c_j> \in R_A\ \wedge$$
$$<a_i, b_k> \in \{<a_1, b_2>, <a_2, b_1>, <a_2, b_3>\}\ \}$$
$$= Chaos_{A\times C} \setminus \{<a_1, c_2>, <a_2, c_1>, <a_1, c_3>, <a_2, c_3>\}$$
$$= \{<a_1, c_1>, <a_2, c_2>, <a_1, c_4>, <a_2, c_4>\}$$

where "\" is the set subtraction operator.

We have $[\, C_C^{real}(x_A, x_B)\,] = \{<a_1, b_1>, <a_2, b_2>\}$ which is a subset of R_C.

We note that the tuplets $<a_1, c_4>$ and $<a_2, c_4>$ are in $[\, C_B^{incomp}\,]$, and therefore they are not included in the reduced maximal solution, which is $[\, C_B^{red}\,] = \{<a_1, c_1>, <a_2, c_2>\}$.

3. SUBMODULE CONSTRUCTION FOR SYNCHRONOUS SYSTEMS

3.1. Modeling systems with multiple interfaces

State machines (with finite or infinite number of states) are often used as models for reactive systems that interact with their environment. Often one considers a system model which is the composition of several state machines. Therefore a state machine is normally a component within a system, it interacts with other components of the system and possibly also with the environment of the system; or the state machine represents the interactions of the whole system with its environment.

A system component has one or more interfaces. An interface is a location where interactions with the environment of the component take place. Each interface i is associated with a domain I_i ; the elements of I_i are the possible interactions that may take place at that interface during a given time unit. We write $x_i^{(t)}$ for the interaction that takes place at interface i at time unit t. Clearly, $x_i^{(t)} \in I_i$ for all t. We write x_i for a sequence of interactions at interface i over a certain time period. We write I_i^* for the set of all finite sequences that can be formed by concatenating interactions from the domain I_i . We have $x_i \in I_i^*$. We note that abstract automata are often modeled without the notion of interfaces; only input and output interactions are distinguished. However, components in embedded systems or distributed systems are often modeled with several interfaces; each interaction is associated with one of these interfaces. The different interfaces of a given component are usually connected to different components within its environment.

We assume trace semantics for the specification of the dynamic behaviour of components or the system, that is, the dynamic behavior is defined in terms of the set of possible execution histories that could occur during the dynamic behavior. For a system with n interfaces i (i = 1, …, n), an execution history consists of a tuplet $< x_1, x_2, \dots , x_n >$ where each x_i (i = 1, … , n) is the sequence of interactions that occurred at interface i during the execution history. We therefore assume that the specification of the dynamic behavior of a system M is given in the form of a (normally infinite) set of such tuplets.

In this section we consider synchronous communication, that is, at each time instant considered, there is an interaction at each interface of the system. Therefore we assume in the following that the interaction sequences x_i at all interfaces have the same length.

The execution histories of a given behaviour can also be viewed from a language perspective. A (formal) language over an alphabet Alph is a subset of the set of sequences of elements of A. Considering the alphabet Alph = $I_1 \times I_2 \times \dots \times I_n$, each execution history is a sequence of elements of Alph, and the behaviour of the system, which is a set of such sequences, is therefore a language over Alph. We note that the concatenation of two execution histories $h = < x_1, x_2, \dots , x_n >$ and $h' = < x'_1, x'_2, \dots , x'_n >$, written h . h', is defined by the separate concatenation of the interaction sequences at the different interfaces, namely $h . h' = < x_1 . x'_1 , x_2 . x'_2 , \dots , x_n . x'_n >$.

Example: As a very simple example, we consider a system component A that has two interfaces B and C with possible interactions 0 or 1, that is, $I_B = I_C = \{0, 1\}$. This means that at each time instant, the component is involved in two interactions, each with

value 0 or 1, and the possible execution histories are of the form $h = < x_1, x_2 >$ where x_1 and x_2 are sequences of zeros and ones. We will write $\#(x_i)$ for the number of "1" in a sequence x_i. Then we may define the behaviour of component A by the predicate $C_A(x_B, x_C) = (\#(x_C) = \#(x_B) / 3)$, where "/" means integer division. This predicate must be satisfied for all execution histories; for instance if the number of "1" in x_B is 4 then the numbers of "1" in x_C must be equal to 1.

Let us assume that the system consists of a certain number of components (sub-systems) C_j (j = 1, …, m), each connected to a certain number of interfaces. As in Section 2, we assume in the following that the dynamic behaviour of the components, and also of the overall system, can be characterized by predicates that depend on the observed interaction sequences at different interfaces. For instance C_1 may be connected to interfaces i = 1, 3 and 6, and therefore its behaviour predicate will be of the form $C_1(x_1, x_3, x_6)$. Similarly the (white-box) behaviour of the overall system can be characterized by a predicate of the form $C(x_1, x_2, \ldots, x_n)$ if there are n interfaces. If the system is composed out of K components and their behaviour predicates C_k (k = 1, …, K) are given, we obtain an overall system behaviour characterized by the predicate $C(x_1, x_2, \ldots, x_n) = \wedge_k (C_k)$, that is, the logical *AND* over all predicates of the components [Adabi 1995].

Besides composition, there is another important operation for describing the behaviour of a system consisting of several components. This is the hiding of an interface that is not visible from a certain perspective. Let us consider a system configuration consisting of several components and n interfaces i (i = 1, …, n). We assume again that the dynamic behaviour of the overall system can be characterized by a predicate $C(x_1, x_2, \ldots, x_n)$. When one of the interfaces (say i) is hidden, we obtain a visible behaviour which only involves the non-hidden interfaces. We use the notation "$hide^{(syn)}_i [C(x_1, x_2, \ldots , x_n)]$" to represent the execution histories of this behaviour. As discussed by Abadi and Lamport [1995], the hiding operation can be defined as follows:

Definition 3-1 (synchronous hiding operator): The operation of hiding the interactions at the interface i from a set of execution sequences $[C(x_1, x_2, \ldots , x_n)]$ leads to the following set of execution sequences:

$$hide^{(syn)}_i [C(x_1, x_2, \ldots , x_n)] = \{ < x_1, \ldots , x_{i-1}, x_{i+1}, \ldots, x_n > \mid$$
$$\exists\, x_i \in I_i^* : C(x_1, \ldots , x_{i-1}, x_i, x_{i+1}, \ldots, x_n) \}$$

3.2. Submodule construction

We now consider a system configuration containing two components M_A and M_B as shown in Figure 2(a). Since the sequences at the three interfaces are constrained by the behaviour of the two components, we have the following predicate that characterizes the set of all possible execution histories of this system:

$$\forall < x_A, x_B, x_C > \in U : \; C_A(x_B, x_C) \wedge C_B(x_A, x_C)$$

where $U = I_A^* \times I_B^* \times I_C^*$ is the universal set of execution sequences for a system architecture as shown in Figure 2(a).

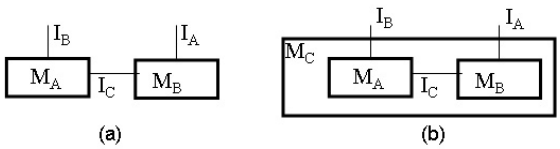


Figure 2: Two components M_A and M_B; (b) also showing the desired overall behavior M_C

Let us now assume that the system consisting of the composition of the two components M_A and M_B is supposed to behave like a system M_C characterized by the predicate $C_C(x_A, x_B)$, as shown in Figure 2(b). Then we have the following requirement:

$$\forall < x_A, x_B, x_C > \in U : \; C_A(x_B, x_C) \wedge C_B(x_A, x_C) \; = \; C_C(x_A, x_B)$$

If we suppose that the behavior defined by $C_C(x_A, x_B)$ represents a safety requirement, that is, all execution histories generated by the two components M_A and M_B must satisfy $C_C(x_A, x_B)$, then we have the requirement:

$$\forall < x_A, x_B, x_C > \in U : \; C_A(x_B, x_C) \wedge C_B(x_A, x_C) \; \Rightarrow \; C_C(x_A, x_B) \qquad (1^{syn})$$

which is identical to Equation (1) in Section 2. Please note that we assume here that the domains D_i of Section 2 are sets of interaction sequences, namely $D_i = I_i^*$ (for i = A, B and C).

The problem of equation solving introduced in Section 2 becomes, in the context of interacting components, the following "submodule construction problem": We assume a system structure as shown in Figure 2(b). If the specification of M_A is given in the form of $C_A(x_B, x_C)$, as well as the safety requirement $C_C(x_A, x_B)$ for the overall system, what is the most relaxed requirement for the dynamic behaviour of machine M_B ?

Since the equation above is identical to equation (1) of Section 2, we can use the solutions provided by Equation (3) or (5) to obtain the most general behaviour of M_B that satisfies (1^{syn}), or the most general reduced behaviour, respectively. We may rewrite these equations, using the hiding operator discussed in Section 3.1, and consider the set of execution sequences defined by the equations. We then obtain the following proposition:

Proposition 3-1: The maximal solution C_B^{max} and the maximal reduced solution C_B^{red} are given by the following equations (where $Chaos_{A\times C}$ is the set of all pairs $< x_A, x_C >$ in $I_A^* \times I_C^*$):

$$[\ C_B^{max}(x_A, x_C)\ ] = Chaos_{A\times C} \setminus hide^{(syn)}_B [\ C_A(x_B, x_C) \wedge \neg C_C(x_A, x_B)\ ] \qquad (3^{syn})$$

$$[\ C_B^{red}(x_A, x_C)\ ] = hide^{(syn)}_B [\ C_A(x_B, x_C) \wedge C_C(x_A, x_B)\ ] \setminus hide^{(syn)}_B [\ C_A(x_B, x_C) \wedge \neg C_C(x_A, x_B)\ ] \qquad (5^{syn})$$

Example: Similar to the example of Section 3.1, we assume that at all three interfaces, there are two possible interactions: $I_A = I_B = I_C = \{0, 1\}$. We use $C_A(x_B, x_C)$ as defined in Section 3.1 and $C_C(x_A, x_B) = (\ \#(x_A) = \#(x_B) / 6\ )$. To find the condition for the missing component B, we can use Formula (2) of Section 2.3 which becomes:

$$C_B^{max}(x_A, x_C) = \forall\, x_B \in I_B^* :\ (\ \#(x_C) = \#(x_B) / 3\ ) \Rightarrow (\ \#(x_A) = \#(x_B) / 6\ )$$

which implies $C_B^{max}(x_A, x_C) = (\ \#(x_A) = \#(x_C) / 2\ )$.

3.3. Prefix closure

A language (or set of execution histories) is prefix-closed if for each sequence included in the language, all its prefixes are also included in that language. This is an important concept since the set of execution sequences of a system starting from its initial state has this property. In fact, the execution history that led the system from the initial state to its current state is a prefix for all execution sequences that may be pursued from the current state. We note, however, that sometimes only "complete" traces are considered as valid execution histories, which are traces that lead to some valid final state; in this case the set of "complete" execution histories is not prefix-closed.

Let us assume that the system has performed synchronous interactions over t time units and the execution history $h = < x_1, x_2, \ldots, x_n >$ has been observed, as explained in Section 3.1. Now we ask the question: What could be the interactions $x_i^{(t+1)}$ at the interfaces (i = 1, …, n) during the next time unit (t+1). These interactions must satisfy the conditions $C_k(\ldots)$ of all components k (k = 1, …, N) that are involved in the composition. Since h is a valid trace, the $x_1, x_2, \ldots x_n$ satisfy all conditions $C_k(\ldots)$. The next interactions $x_1^{(t+1)}, x_2^{(t+1)}, \ldots, x_n^{(t+1)}$ must be chosen such that the extended execution history $h' = < x_1 . x_1^{(t+1)}, x_2 . x_2^{(t+1)}, \ldots, x_n . x_n^{(t+1)} >$ also satisfies all conditions $C_k(\ldots)$. We note that this means that there is a kind of global rendezvous between all the components to agree on the interactions $x_i^{(t+1)}$ at all interfaces such that they are valid for all components.

We note that the definition of this kind of synchronous rendezvous that occurs simultaneously at several interfaces is a concept that would be difficult to implement in a distributed environment. It is not clear whether this is a concept of practical importance, however, we think that it exhibits a theoretical simplicity which makes it interesting. We note that the issues discussed for this kind of synchronous communication carry over to the more practical communication paradigms discussed in the subsequent sections.

3.4. The case of regular behaviors

The behaviour of a component is often modeled by a state machine. If the state machine model has a finite number of states, the set of possible execution histories is a language which can be defined by a regular expression. Therefore such behaviors are called regular. We note that state machines are just a convenient way of defining certain types of behaviors. It is clear that the set of execution sequences defined by a state machine could also be described by a characterizing predicate, as we assumed in the preceeding sections.

One distinguishes different types of state machines depending on the nature of the interactions with the environment, such as accepting automata, finite state machines where an input is followed by an output within a single transition, and Input/Output Automata (IOA) [Lynch 1989] or Input-Output Transition Systems (IOTS) [Tretmans 1996] where inputs and outputs are realized through separate transitions. For the synchronous rendezvous communication considered within this section, we consider accepting automata as a model of the component behaviour.

An accepting automata has a finite number of states and transitions between these states which are labelled by an element of the interaction alphabet Alph (as defined in Section 3.1) which is a tuplet of interactions taking place at the different interfaces. A subset of the states are usually considered accepting states, which means that an execution history that leads to one of these states is accepted by the automaton, that is, it represents valid behavior. In the following, we often deal with prefix-closed behaviors which can be modeled as automata for which all states are accepting.

An example behaviour definition is shown in Figure 3(a). It represents the behaviour of the module M_C of Figure 2(b) and each transition is labelled with the two interactions that take place at the two interfaces I_A and I_B . For instance, in the initial state, there is only one possible transition with interactions a_1 at I_A and b_1 at I_B . In state 2, two transitions are enabled, both having interaction a_2 at interface I_A .

One advantage of this modeling approach is the fact that operations on behaviour definitions can be performed by simple algorithms (see for instance [Aho et al. 1986]). We consider in particular:

- Completing the behaviour model, that is, there should be a transition for all combinations of interactions in each state. This model can be obtained by adding a (non-acccepting) *Fail* state and a new transition to this state from each state of the automaton for those combinations of interactions for which there is no transitions in the original model. As an example, Figure 3(c) shows a completed version of the behavior of Figure 3(a).
- Finding an equivalent deterministic model, assuming that the given automaton is non-deterministic. A state s in the equivalent deterministic model represents the set of states in which the original model could be after having participated in the execution sequence that leads to the state s (see [Aho et al. 1986] for more

details). Unfortunately, this operation has a worst-case algorithmic complexity that is exponential in the number of states in the original model.

- Complement of a behaviour. This corresponds to all execution histories that are invalid for the given behaviour, e.g. the complement of $[C_A(x_B, x_C)]$ is $(I_B^* \times I_C^*) \setminus [C_A(x_B, x_C)]$. For a deterministic completely defined automaton, it is sufficient to exchange accepting and non-accepting states. If the given automaton is non-deterministic, it must first be converted into an equivalent deterministic one (see above).
- Product of two behaviors, e.g. $C_A(x_B, x_C)$ and $C_B(x_A, x_C)$, where the interactions at the shared interfaces, e.g. I_C , must be the same. This corresponds to the logic expression $C_A(x_B, x_C) \wedge C_B(x_A, x_C)$. The algorithm for constructing the product is very simple: The states of the product are of the form $<s_A, s_B>$ and the product has a transition from $<s_A, s_B>$ to $<s'_A, s'_B>$ labelled (x_A, x_B, x_C) iff $C_A(x_B, x_C)$ has a transition labelled (x_B, x_C) from s_A to s'_A and $C_B(x_A, x_C)$ has a transition labelled (x_A, x_C) from s_B to s'_B . A product state is accepting if the states of both behaviours are accepting.
- Hiding an interface (as defined in Section 3.1): This operation can be performed by deleting the interaction of that interface in the labels of all transitions. Note that in general a non-deterministic automaton is obtained. In the case of interleaving semantics (see Section 4), when there is a non-null interaction on at most one interface at a given time, one usually replaces the non-null interactions at the hidden interface by a symbol, e.g. "i", representing an internal event.

Using these operations, the equations (3^{syn}) and (5^{syn}) can be evaluated algorithmically. We discuss in the following an algorithm to determine the reduced maximal solution using equation (5^{syn}). We assume that both behaviors, $C_A(x_B, x_C)$ and $C_C(x_A, x_B)$ are prefix-closed and given in the form of two automata. The following algorithm is proposed:

Algorithm 3-1 (to find the reduced maximal solution):

Step 1: Build the completed model of $C_C(x_A, x_B)$ by introducing a (non-accepting) *Fail* state and the transitions leading to it.

Step 2: Construct the product automaton $C_A(x_B, x_C)$ x $C_C(x_A, x_B)$. In this automaton, each transition will be labelled with a tuplet $<x_A, x_B, x_C>$, and each state has the form $<s_A, s_C>$ where s_A is a state of $C_A(x_B, x_C)$ and s_C is a state of $C_C(x_A, x_B)$.

Step 3: For this product automaton, hide the interactions at the interface I_B .

Step 4: Find an deterministic automaton equivalent to the one obtained in Step 3. In this resulting automaton, each state represents a set of states of the product automaton, that is, a set of pairs $<s_A, s_C>$.

Step 5: For the automaton obtained in Step 4, designate as non-accepting each state which includes a state pair $<s_A, s_C>$ for which s_C = *Fail*. (The accepting status of the other states are not changed). The resulting automaton, which we call B^I, represents the reduced maximal solution $C_B^{red}(x_A, x_C)$.

To understand the significance of the solution B^1, it is important to note the following points:

- The interaction sequences allowed by B^1 are exactly those that are compatible with A. Other sequences could not be executed by the component B jointly with A because A would block them.
- The state reached by B^1 after a given execution sequence represents exactly the set of state pairs in which the components A and C could be when they would execute jointly communicating over the interface I_B (without the presence of B) generating at the interfaces I_A and I_C the given execution sequence. (Note that the interface I_B is not visible by B).

Proposition 3-2: Algorithm 3-1 finds the maximal reduced solution as defined by Equation (5^{syn}).

Proof: It is clear that Steps 1 through 4 lead to an automaton that represents the behaviour of the first line of Equation (5^{syn}). Note that a state of the obtained deterministic automaton is accepting if one of the corresponding state pairs $<s_A, s_C>$ is accepting, that is, both s_A and s_C are accepting. To construct an automaton that represents the behaviour of the second line of Equation (5^{syn}), one would proceed through the same steps, except that after Step 1, a complement must be performed (which leads to an exchange of accepting and non-accepting states), and after Step 4, another complement must be performed. For this purpose, one would normally first complete the obtained automaton (by introducing a new fail state, say *F1*), and then exchange accepting and non-accepting states. Because the accepting and non-accepting states are exchanged twice, this would lead to the same automaton which was obtained by Steps 1 through 4 except that the states that have a corresponding state pair $<s_A, Fail>$ are non-accepting. Since the acceptance conditions for the first line and the second line must be satisfied, Step 5 is introduced. Note that the transitions to the new fail state *F1* have no impact on the result because they are in conflict with the behaviour of $C_A(x_B, x_C)$. □

As an example, we consider the behaviour $C_C(x_A, x_B)$ defined by the transition diagram of Figure 3(a) and the behaviour $C_A(x_B, x_C)$ shown in Figure 3(b). The following figures show the results of the different steps during the derivation of the behaviour of $C_B^{red}(x_A, x_C)$ according to Formula (5^{syn}) and Algorithm 3-1. The completion of $C_C(x_A, x_B)$ is shown in Figure 3(c) – that is, there is a transition for each interaction tuplet from each state, with the addition of a non-accepting *Fail* state - and the complement is shown in Figure 3(d) – obtained by interchanging accepting and non-accepting states. Figure 3(e) shows the product of this complement with $C_A(x_B, x_C)$, and Figure 3(f) shows it after hiding the interactions at the I_B interface. Finally, Figure 3(g) shows the reduced maximal solution C_B^{red}, obtained from Figure 3(f) after determinization and exchange of accepting and non-accepting states. We note that the maximal solution C_B^{max} is similar, it includes additional transitions indicated in the figure as arrows that do not lead to any state; these transitions will never be executed in the context of the architecture of Figure 2(b). To

check the correctness of the solution, we show in Figure 3(h) the composition of the given component A with the solution of Figure 3(g); which after hiding the interactions at the interface I_C becomes as shown in Figure 3(i). As mentioned earlier, the additional transitions in C_B^{max} do not contribute to this joint behaviour. One notes that the states (1, (1,1)) and (1, (2,2 or 1.1)), as well as (2, (2,2)) and (2, (2,2 or 1,1)), are equivalent; therefore this composition is equivalent to the behaviour of C_C, as defined in Figure 3(a), which means that the solution is complete (and it is also progressive – see below). This is a quite trivial example; we note that the behaviour of Figure 3(j) is also a progressive solution to the equation; it is larger than C_B^{red} , but not maximal – its advantage is its simplicity. **Notation:** In a product, a state name (1,1) means that both components are in state 1; (F,*) means that the first component is in the *Fail* state and the other component in any state; in the determinized product after hiding, the state name ((2,2),(1,1)) means that the two components are either in the states (2,2) or in (1,1). The non-accepting states are indicated by a dashed rectangle.

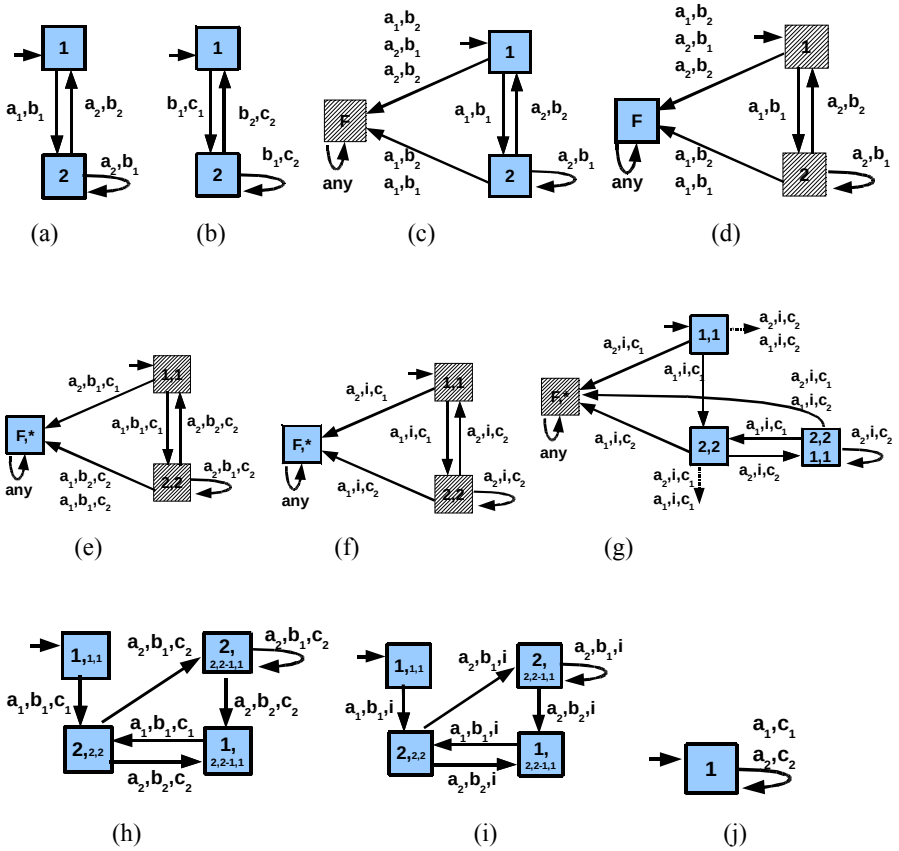


Figure 3: Simple example of submodule construction for synchronous machines. The alphabets at the interfaces are: $I_A=\{a_1, a_2\}$, $I_B=\{b_1, b_2\}$, $I_C=\{c_1, c_2\}$: (a) behaviour $C_C(x_A, x_B)$; (b) behaviour $C_A(x_B, x_C)$; (c) completed behaviour description for C_C (x_A, x_B); (d) shows the complement of (c); (e) shows the product of (d) with (b); (f) behaviour of (e) with hidden interactions at interface I_B; (g) behaviour of C_B^{red} obtained from (f) after determinization and exchanging accepting states; (h) composition of C_B^{red} with $C_A(x_B, x_C)$; (i) behaviour (h) with interface I_C hidden; (j) another solution of the submodule construction problem.

A more interesting example is shown in Figure 4. Figures 4(a) and (b) show the behaviour of C and A, respectively. In Figures 4(a), (c) and (d), the non-accepting states (including *Fail*) are not shown; instead the labels of transitions leading into such states are written next to their starting states. Figure 4(c) shows the product C x A, and the solution (its determinization after hiding the b_x interactions) is shown in Figure 4(d). Finally, the combined behaviour of A with the solution is shown in Figure 4(e). By looking at this figure, we see that the realized subset of C (as defined in Section 2.4) is not equal to C, which implies that the solution is not complete. In fact, the transition (a_2,b_1) from state 2 in Figure 4(a) is never executed by the behaviour of Figure 4(e); the transition (a_1,b_2) from state 1 is not executable from the initial state of Figure 4(e); and the transition (a_1,b_1) from state 2 can only be executed three times in a row. However, there are no deadlocks.

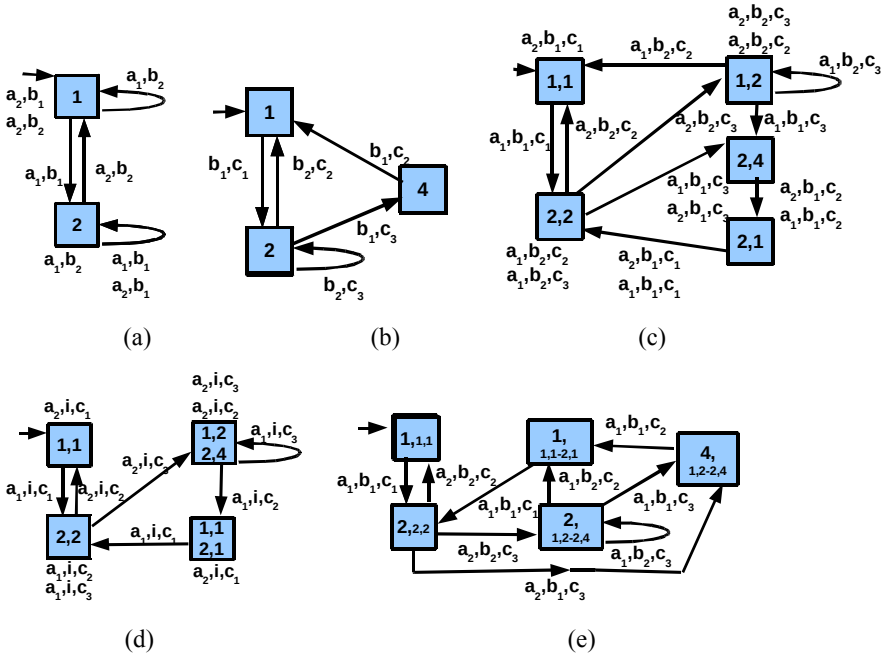


Figure 4: Another example with synchronous communication; (a) behaviour of C; (b) behaviour of A; (c) product C x A; (d) solution; (e) product A x solution (see explanations in the text)

3.5. Prefix-closed solutions

It is important to note that the complement of a prefix-closed language is not prefix-closed. Therefore the solution obtained according to Formulas (3^{syn}) or (5^{syn}) are, in general, languages that are not prefix-closed, even if the specifications of $C_C(x_A, x_B)$ and $C_A(x_B, x_C)$ are prefix-closed, as demonstrated by the example shown in Figure 5.

This example is a modification from Figure 3. The behaviour of C is the same while a new state 5 has been added to A, as shown in Figure 5(a) (the new state and transitions are shown in bold). Figure 5(b) shows the product C x A, and the solution is shown in Figure 5(c). We see that all transitions from the initial state of the solution lead C into the

Fail state. Therefore the prefix-closed solution is the empty sequence (B blocks in the initial state). However, there exist solution execution histories with longer length. For example, when B has executed the sequence (a_1, c_1), (a_2, c_2), the component A will be in state 1 or 2 and the behaviour at the interfaces I_A and I_B corresponds to the state 1 or 2 of C; this is a solution sequence (since C is in an accepting state), however, it is not prefix-closed since the prefix (a_1, c_1) is not a solution.

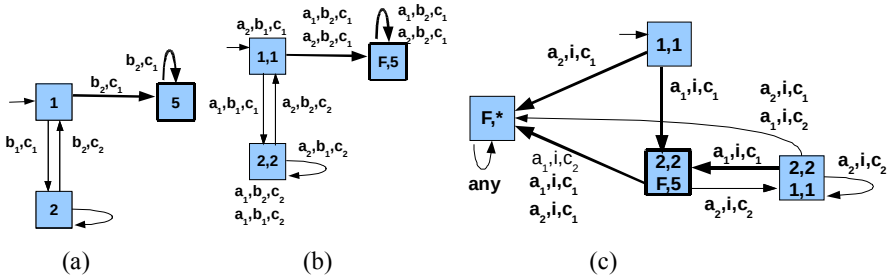


Figure 5: Submodule construction with a solution that is not prefix-closed; (a) behaviour of A; (b) product C x A; (c) solution

In order to obtain a prefix-closed solution to the submodule construction problem, the following Step 6 can be applied to B^1 which was obtained by Steps 1 through 5 of Algorithm 3-1:

Algorithm 3-2 (for prefix-closed state pruning):

Step 6: Identify all states in B^1 that include, in the set of their corresponding state pairs, a pair (s_A, *Fail*) for some s_A. Prune all transitions leading to these states, using the following algorithm for pruning transitions. We call the resulting solution B^2 .

Algorithm 3-3 (for pruning transitions - rendezvous communication):

Input: a set of transitions of the solution state machine B (to be eliminated).

Step (a): Delete the set of transitions from the state machine B. (Note: for input-output communication, this procedure is more complicated – see Section 5.2.1)

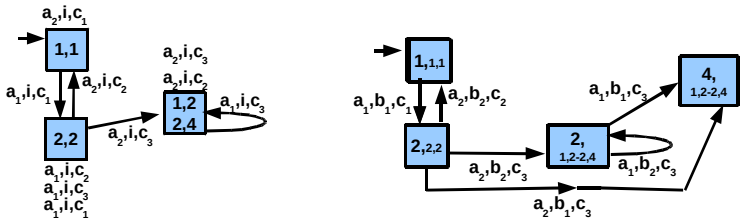
Step (b): Delete any state from the state machine B that is not any more accessible from the initial state. Also delete any outgoing transitions from these states.

Proposition 3-3: Algorithm 3-1 followed by Algorithm 3-2 finds the maximal reduced prefix-closed solution for the submodule construction problem for synchronous rendezvous communication.

Proof: In the case of a prefix-closed solution, all states reached by one of the execution sequences must be accepting. Since the states of B^1 that are associated with a state pair (s_A, *Fail*) are non-accepting, any execution sequence that leads to such a state must be eliminated. This is what Algorithm 3-2 does. The pruning Algorithm 3-3 is straightforward in this case of synchronous communication; it eliminates the transitions leading to the non-accepting states in Step (a), while Step (b) simply eliminates the states that have become non-accessible, which does not change the behavior. □

3.6. Deadlocks and progressiveness

We note that the behaviors obtained for the component B according to the algorithms above often include deadlocks or dead ends with loops. Such parts of the behavior should normally be eliminated. However, this usually leads to a reduction of the realized subset of the global behavior C. As an example, Figure 6(a) shows the reduced maximal solution for the case that component A is defined as in Figure 4(b) and the desired global behavior C is as shown in Figure 4(a), except that the transition (a_1, b_1) from state 2 is deleted. The solution shows a dead end at the state (1,2 or 2,4), which leads to a deadlock in the product machine A x B, as shown in Figure 6(b). Therefore one would delete the state (1,2 or 2,4) together with the transition leading to it. This would result in a realized global behavior C as shown in Figure 6(b) without the two states on the right. This behavior has no deadlock and is a subset of the desired behavior shown in Figure 4(a).



(a) (b)

Figure 6: Example of deadlocks: (a) solution with dead-end loop; (b) deadlock in the product of A with the solution (a) – see explanations in text.

Algorithm 3-4 (for eliminating deadlocks in the realized global behavior):

The input to this algorithm is the prefix-closed solution B^2 and the state machine of component A.

Step 7: Set $B^3 := B^2$

Step 8: Construct the product machine P = A x B^3. This machine represents the realized global behavior. Note that each state of P is a pair (a state s_A, a set of state pairs $<s'_A, s'_C>$), where the state s_A must be equal to one of the states s'_A ; and each transition of P is labelled by a tuple (i_A, i_B, i_C).

Step 9: "Prune" each state s in P that deadlocks, that is, has no outgoing transition. By "pruning a state in P" we mean to make the necessary changes to B^3 such that this state cannot be reached any more (since we can only prune states or transitions in B, but not directly in the product machine P). Therefore, in order to "prune" a state s = (s_A, $\{<s'_A, s'_C>\}$) in P, we prune all those transitions of B^3 that participate in a joint transition of A x B^3 into this state s. We use Algorithm 3-3 for this purpose.

Step 10: If the initial state has been deleted, then there is no solution. Otherwise, if some transitions were deleted, go back to Step 8. If no transitions were deleted in the last step then B^3 is the largest solution without deadlock.

Proposition 3-4: Algorithm 3-1 followed by Algorithms 3-2 and 3-4 find the maximal reduced prefix-closed solution that does not lead to any deadlock when executed jointly with the given component A.

Proof: Algorithm 3-4 is in some sense "experimental": The product behaviour of the given component A with the solution is formed, and if a deadlock state of this product is detected, the transitions of the solution by which the product could enter this state are eliminated. This process is repeated until no deadlock is encountered. The resulting solution is maximal since only those transitions of the solution are eliminated that actually lead to a deadlock state of the product. □

We note that one may also eliminate deadlocks in two phases, first eliminating deadlocks in the solution B^2 , and then eliminate the deadlocks that may still show up in the realized global behavior P, using the algorithm above.

We see from the above examples that it is very common that the desired behavior C is only partially realized with the given component A. In the case of synchronous communication, a minimum solution always exists which is a component B that blocks in its initial state. In other situations, all defined execution histories of C are realized, but this does not necessarily mean that no deadlock may occur in the realized behavior.

There is a more subtle property of non-blocking based on refusal semantics [Hoare 1985], sometimes called progressiveness. We recall that the product machine P, determined in Step 8 above, represents the realized global behavior. We discussed above how one can eliminate blocking situations (deadlocks) in this behavior. We note, however, that a realized behavior that is complete (realizes all traces of C) and has no deadlock does not necessarily have the same blocking properties as the specification C. This is due to possible non-determinism. If we want to compare P with the state machine of C, we first have to hide the interactions at the I_C interface, and this may lead to non-determinism. Assuming for instance that the behavior of C is defined by Figure 3(a); it could be that the joint behavior of A and the solution would have two alternative branches for getting into a state corresponding to state 2 of Figure 3(a); and going through one of these branches, the self-loop transition (a_2, b_1) of Figure 3(a) would not be possible. This would represent a refusal of the next transition (a_2, b_1) which is however foreseen by the behavior C.

Definition (progressive solution): A solution to the submodule construction problem is **progressive** if the realized behavior, in any reached state, can perform all transitions that are specified by the desired behavior C in its corresponding state. (Note that C is assumed to be deterministic).

We propose the following algorithm for obtaining a (complete) progressive solution, if it exists.

Algorithm 3-5 (progressive solution): The algorithm uses the four steps (7) through (10) of Algorithm 3-4 for deadlock removal, but Step (9) is replaced by the following. (Note: See explanation in Step 8 of Algorithm 3-4 concerning the form of the states of P and the labels of its transitions)

Step 9: For each state $s = (s_A, \{<s'_A, s'_C>\})$ in P do the following:

For each pair $<s'_A, s'_C>$, and for each outgoing transition from s'_C in C determine the label (i_A, i_B) of that transition and check that P has a transition from the state s which is labeled (i_A, i_B, i_C) for some i_C . If this condition is not satisfied, "prune" this state as explained in Step 9 of Algorithm 3-4.

We note that it is sometimes useful to distinguish between required traces and optional traces in the behavior specification of a component [Drissi 1999]. In the above, we have assumed that all traces of C are required. The notion of progressiveness can only be applied to required traces.

4. SUBMODULE CONSTRUCTION FOR INTERLEAVING SEMANTICS

4.1. Modeling interleaving semantics

In this modeling framework, we also have rendezvous interactions at interfaces, but interleaving semantics is assumed, which means that at most one interaction (on a single interface) may occur during each time unit. We use in the following the same modelling framework used for synchronous machines, but introduce the following changes:

(a) We allow an interface to have the value *null* during a given time unit, which means that no interaction takes place at this interface during this time unit. We write $x_i^{(t)} \in (I_i \cup \{null\})$.

(b) In a system of several components with n interfaces, a possible execution history $< x_1, x_2, \dots , x_n >$ must satisfy the following constraint IC, called **interleaving constraint**:

$$IC(x_1, x_2, \dots , x_n) = \text{for all } t : x_i^{(t)} \in I_i \text{ implies } x_j^{(t)} = null \text{ for all } j \neq i.$$

Any execution history $h = < x_1, x_2, \dots , x_n >$ that satisfies the interleaving constraint defines a linear (time) order for the (non-null) interactions at the interfaces. We write seq(h) for this sequence and call it the execution sequence corresponding to h. For execution sequences over n interfaces, as above, we have $seq(h) \in (I_1 \cup I_2 \cup \dots \cup I_n)^*$. Normally, the semantics of labelled transition systems is described in terms of these (finite or infinite) execution sequences and the possibilities of blocking after finite sequences. We will continue using the model of separate interaction sequences x_i at the different interfaces, as introduced for synchronous communication; we thus obtain a uniform framework for treating systems with both types of communication, synchronous and interleaving.

Definition 4-1 **(equivalence of execution histories):** Since only the execution sequences count for the semantics of labelled transition systems, we say that two execution sequences h_1 and h_2 are equivalent, written $h_1 \cong h_2$, if they define the same execution sequence, that is, $seq(h_1) = seq(h_2)$.

This corresponds to the so-called stuttering equivalence between execution sequences that contain at certain time units null-interactions at all interfaces. Clearly, we assume that any predicate defining the behaviour of a given system component has the

same value for equivalent execution histories; the value should only depend on the corresponding execution sequence.

The notion of equivalence between execution histories leads to a slightly modified definition of the hiding operator as follows:

Definition 4-2 (interleaving hiding operator): The operation of hiding the interactions at the interface i from a set of execution sequences $[C(x_1, x_2, \dots, x_n)]$ leads to the following set of execution sequences:

$$\begin{aligned} \text{hide}^{(LTS)}{}_i\,[C(x_1, x_2, \dots, x_n)] = \{ &< x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n > \mid \\ & IC(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) \\ \wedge\ & \exists < x_1', \dots, x_{i-1}', x_i', x_{i+1}', \dots, x_n' > : \ (IC(x_1', \dots, x_{i-1}', x_i', x_{i+1}', \dots, x_n') \\ & \wedge\ < x_1', \dots, x_{i-1}', x_i', x_{i+1}', \dots, x_n' > \in [C(x_1, x_2, \dots, x_n)] \ ) \\ & \wedge\ < x_1', \dots, x_{i-1}', x_{i+1}', \dots, x_n' > \cong < x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n > \ \} \end{aligned}$$

This definition is more complex than Definition 3-1 for synchronous communication because equivalent execution sequences are not always of the same length (e.g. when one or both include time instants where the interactions at all interfaces are null). Therefore an execution sequence $<x_1, \dots, x_n>$ of the hidden behaviour may be shorter than any of the original sequences that include the interface i. The sequences $<x_1', \dots, x_n'>$ represent those sequences of different length to which the sequence $< x_1, \dots, x_n >$ may correspond.

4.2. Submodule construction

Due to the interleaving constraints and the equivalence between execution histories, we have the following modified equations. Equation (1^{syn}) becomes:

$$\forall < x_A, x_B, x_C > \in U :$$
$$IC(x_A, x_B, x_C) \wedge C_A(x_B, x_C) \wedge C_B(x_A, x_C) \Rightarrow C_C(x_A, x_B) \qquad (1^{LTS})$$

Equation (2) becomes:

$$\begin{aligned} C_B{}^{max}(x_A, x_C) = IC(x_A, x_C) \ \wedge \forall < x_A', x_B', x_C' > \in U : & \\ IC(x_A', x_B', x_C') \ \wedge \ C_A(x_B', x_C') \Rightarrow C_C(x_A', x_B') & \\ \wedge \ <x_A', x_C'> \cong <x_A, x_C> \qquad (2^{LTS}) & \end{aligned}$$

This definition of $C_B{}^{max}$ says that an execution history at the interfaces I_A and I_C is an allowed behavior for component M_B if for all global execution histories $< x_A', x_B', x_C' >$ that have an equivalent behavior for M_B, the satisfaction of C_A leads to the satisfaction of C_C . This modification to Equation (2) is introduced because the specification of the behavior for M_B can only restrain the possible execution sequences of the component, but has no impact on which of the equivalent execution histories would be realized in collaboration with the other system components and the environment.

Using a similar derivation as for Equations (3) in Section 2, it is easy to see that Equation (2LTS) is equivalent to

$$\begin{aligned} C_B{}^{max}(x_A, x_C) = IC(x_A, x_C) \ \wedge \ \neg\exists < x_A', x_B', x_C' > \in U : & \\ IC(x_A', x_B', x_C') \ \wedge C_A(x_B', x_C') \ \wedge \neg C_C(x_A', x_B') \ \wedge <x_A', x_C'> \cong <x_A, x_C> & \end{aligned}$$

Using the definition of the hiding operator given above, this leads to the following proposition:

Proposition 4-1: The maximal solution C_B^{max} and the maximal reduced solution C_B^{red} for the submodule construction problem with interleaved communication are given by the following equations:

$$[\ C_B^{max}(x_A, x_C)\ ] = [\ IC(x_A, x_C)\ ] \cap \neg\, hide^{(as)}_B [\ C_A(x_B, x_C) \wedge \neg C_C(x_A, x_B)\ ] \quad (3^{LTS})$$

$$[\ C_B^{red}(x_A, x_C)\ ] = hide^{(LTS)}_B [\ C_A(x_B, x_C) \wedge C_C(x_A, x_B)\ ] \setminus hide^{(LTS)}_B [\ C_A(x_B, x_C) \wedge \neg C_C(x_A, x_B)\ ] \quad (5^{LTS})$$

Equation (5^{LTS}) was presented (using a different notation) by Bochmann and Merlin [1980], which was the first paper on submodule construction to our knowledge. We note that this formula is the same as (5^{syn}), except that a different hiding operator is used.

Like in the case of synchronous machines, these solutions may be evaluated algorithmically when the specifications of C_A and C_C are given in the form of regular languages (finite state machines – labelled transition systems). Since Equation (5^{LTS}) has the same form as Equation (5^{syn}), Algorithm 3-1 can be used for obtaining a reduced maximal solution, however, the different hiding operator $hide^{(LTS)}$ must be used in Step 3, and the algorithm for obtaining the product of two machine behaviors is different (see [Aho 1986]). For instance, the transitions of the product machine C x A are not labeled with triplets (i_A, i_B, i_C), but by single interactions i_A, i_B, or i_C.. The issues of prefix-closure, deadlocks, and progressiveness are the same as for synchronous systems.

Algorithms 3-2 and 3-3 for obtaining a prefix-closed solution can also be applied for specifications with interleaving semantics. We note, however, that in contrast to the case of synchronous communication, we may find a solution state machine for which the initial state is not accepting. In this case, there is no prefix-closed solution, not even a minimal all-blocking one. This situation could occur when component A starts with an interaction i_B that is not allowed in the initial state of C.

Algorithm 3-4 for finding a deadlock-free solution can be used in the context of interleaved communications, however in Step 9, the sentence "we prune all those transitions of B^3 that participate in a joint transition of A x B^3 into this state s" must be replaced by "for all transition paths of A x B^3 that lead into the state s, we prune in B^3 the transition that is the last transition of B^3 involved on that path". Note that in interleaving semantics, some transitions of A x B^3 may only involve component A.

The Algorithm 3-5 for finding a progressive solution must also be adapted since, for interleaving semantics, an interaction required by the behavior C_C may not be enabled immediately when the last interaction of C_C just occurred. This is so because some interactions at the interface I_C (between the two components) may be required before the next interaction of C_C can occur, while that interface is not visible from the perspective of C_C . We propose the following algorithm for finding a progressive solution.

Algorithm 4-1 (progressive solution – interleaving communication): The algorithm uses the four steps (7) through (10) of Algorithm 3-4 for deadlock removal, but Step (9)

is replaced by the following. (Note: See explanation in Step 8 of Algorithm 3-4 concerning the form of the states of P)

> Step 9: For each state s = (s_A, {<s'_A, s'_C >}) in P do the following:
>
> For each pair <s'_A, s'_C >, and for each outgoing transition from s'_C in C determine the label i_A or i_B of that transition and verify that there is a sequence of transitions from state s in P with interactions at interface I_C and leading to a state s' in P from which a transition labeled i_A or i_B , respectively, is enabled. If for some state s this condition is not satisfied, "prune" this state as explained in Step 9 of Algorithm 3-4.

We note that this algorithm does not check for livelocks (infinite loops) involving only interactions at the interface I_C, which may prevent any progress at the other interfaces. The problem of progressiveness is discussed in [Buffalov 2003] in a more general context.

4.3. Example

An example is shown in Figure 7. State diagrams representing the behaviour of C and A are shown in Figures 7(a) and (b). The behaviour of the product C x A is shown in Figure 7(c). Figure 7(d) shows the reduced maximal solution which is obtained from Figure 7(c) after the following two steps: (a) hiding the interactions b_1 and b_2 at the interface I_B , and (b) transforming the resulting non-deterministic machine into an equivalent deterministic one. We note that the states (2,1) and (3,1) become invalid states after hiding the b-interactions, since they have a spontaneous transition (previously labelled b_1 or b_2) which leads C into the *Fail* state. Such states, together with transitions leading to them, must be pruned (see Step 5 in Algorithm 3-1).

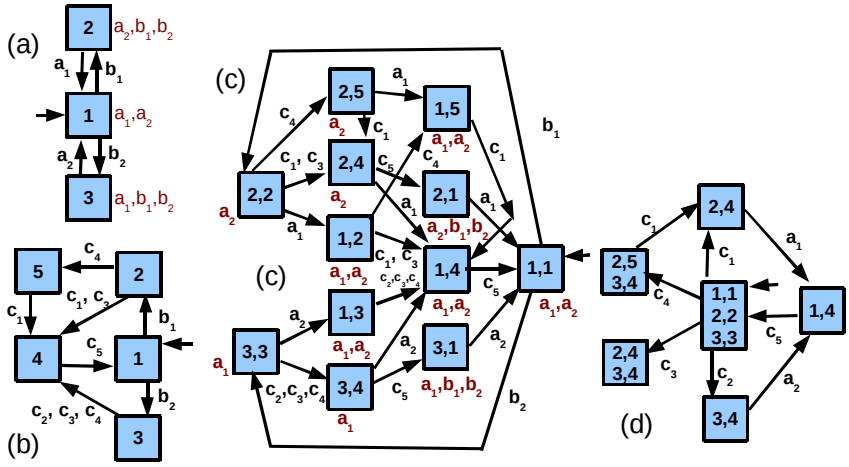


Figure 7: (a), (b) State diagrams representing the behavior of C and A; (c) behavior of the composition C x A; (d) the same after hiding interactions b1 and b2 and determination.

As discussed in Section 3.5 for synchronous systems, the obtained solution may contain deadlocks, or may lead to deadlocks in the realized global behavior. The same problems occur with interleaving semantics; and the same remediation is proposed: pruning certain transitions of the behaviour of the solution (assuming that the specifications of C and A cannot be changed). If we do this in two phases, we first

eliminate the deadlocked states in the solution, namely (2,4 or 3,4) in Figure 7(d) by eliminating the transition c_3 leading to it. In the second phase, we check for deadlocks in the realized global behaviour. In our example, the execution sequence (b_2, c_4) leads to a deadlock. Following Algorithm 3-4 for eliminating deadlocks, we would delete the transition c_4 from the initial state of the solution and therefore also the state (2,5 or 3,4) with its outgoing transition. This leads to the maximal deadlock-free solution.

.

5. EQUATION SOLVING FOR BEHAVIOR SPECIFICATIONS WITH INPUTS AND OUTPUTS

5.1. Component specifications based on assumptions and guarantees

In Sections 3 and 4, we considered rendezvous communication between the different system components. In that case, an interaction can only occur on a given interface when all components connected to that interface are ready, and the execution of that interaction involves the execution of a corresponding transition in each component. In the case of automata with inputs and outputs (without message queuing), the execution of an interaction on a given interface also involves simultaneous transitions in each of the connected components, however, the distinction of input and output implies that the interaction is an output produced by one component, and it is input to all other connected components. The outputting component alone may select the interaction to be executed. We do not consider queued interactions in this paper. In order to avoid competition for output at a given interface between several components, we assume that each interface represents output for exactly one component and input for all other connected components. An example is shown in Figure 8(a).

In the case of synchronous communication, an output is produced at each interface during each time unit. In the case of interleaved communication (which is the case for traditional IOA [Lynch et al. 1989] and IOTS [Tretmans 1996]), the outputting components (and the environment) may compete for triggering the next interaction by their respective output within the interleaving semantics. It appears that input and output interactions correspond, in the context of discrete event systems for controller design [Ramage et al. 1989] to what is called “uncontrollable” and “controllable” interactions, respectively.

Since a given component has no control over which input will occur at its interfaces, there are two schools of specification. Often so-called “completed” specifications are considered; they contain, for each state, transitions for all possible inputs. A more realistic approach is to use so-called “partial” specifications which, for certain states, may not include transitions for all possible inputs; it is assumed that such “non-specified” inputs will not occur when the component is in the state in question. Partial specification is a paradigm where the specification of a component contains two parts: (1) the assumptions about the inputs received from the component’s environment (what are the

valid inputs ?), and (2) the guarantees about the outputs that will be produced by the component (what outputs may be produced by the component ?).

Using this paradigm (see for instance [Abadi et al. 1995, Misra et al. 1991]), the specification of the requirements for a given component M includes two parts: (1) The assumption C_M^{Ass} about the behavior of the environment that the implementation of the component may assume, and (2) the guarantees C_M^{Guar} that the implementation of the component must ensure. The specification C_M of the component then has the form

$$C_M = C_M^{Ass} \Rightarrow C_M^{Guar}$$

which means that the guarantees are provided if the assumptions hold.

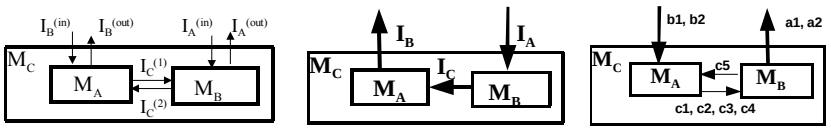


Figure 8: (a) two communicating system components with separate input and output interfaces; (b) and (c) – two different input-output relationships

Since two interacting components, as shown in Figure 8(a), may make assumptions about one another, one needs a method for deriving the global behavior of the composition without circular reasoning. In the case of prefix-closed behaviors, this can be done by using induction over the length of the execution histories (or sequences) [Adabi 1995]. Let us assume that for an execution history up to time t, the assumptions of all components have been satisfied. Then the guarantees for each component will indicate the possible outputs that may occur at the next time instant. Based on this information, one can then verify whether the assumptions still holds at the next time instant.

For synchronous systems, this reasoning only goes through if we assume that the outputs allowed at time t do not depend on the inputs received at the same time unit (but only on previous inputs and outputs). This implies that a delay of at least one time unit exists between a received input and the output which is *caused* by this input. The importance of this assumption is discussed in [Adabi 1995, Broy 1995]. We make the additional assumption that the question of whether a given input is valid after a given execution history (assumption) is independent of the output that will be produced by the component during the same time unit. We call this the **unit delay assumption** [Bochmann 2002b].

5.2. Submodule construction

We consider here the following cases:

1. **Synchronous behaviors with complete specifications:** Each state transition is associated with one interaction at each interface, which may be input or output depending on the nature of the interface. A complete behavior specification means that in each state of the state machine there is a transition for each combination of input values. This means that the behavior specification includes

no assumptions. This case is sometimes called synchronous composition of finite state machines (FSM) (see for instance [Yevtushenko et al. 2000]).

2. **Synchronous behaviors with partial specifications** (see below)
3. **Interleaving behavior with completely defined specifications:** Similar as in case (1) above, the complete specification implies that there is in each state an input transition for each input interaction. No assumption is made about the environment.
4. **Partially defined interleaving behavior** (see below)
5. **Completely specified FSM with interleaved communication (so-called parallel composition):** An FSM (where each transition has an input and an output) may be modeled by an IOA by assuming that each input transition of the IOA is followed by an output transition, and that no new input will be applied before the output has occurred. This alternative occurrence of inputs and outputs can be modeled by IOA behavior guarantees and assumptions (see case (4) above). Because of the completely specified input assumption, however, the detailed treatment is similar to case (3). For more details, see [Petrenko et al. 1998] and [Yevtuschenko et al. 2000].

The cases of completely defined behavior are very similar to what is discussed in Sections 3 and 4. The only difference is that a distinction between input and output interactions must be made. However, this does not impact the equations and algorithms for submodule construction discussed in Sections 3 and 4. But the algorithm for transition pruning must be changed, because inputs to component B are not controllable and the corresponding transitions cannot be simply eliminated as in the case of rendezvous communication. See below for more details.

5.2.1. Synchronous automata with partial behavior specifications

In this case, the automata may not have a transition from some given state for some given combination of inputs received. This means that the assumption is made that such a combination will not occur when the component is in the given state. For submodule construction, Equations (1^{syn}), (3^{syn}) and (5^{syn}) can be applied here, and we get the following proposition:

Proposition 5-1: The maximal reduced solution C_B^{red} for the submodule construction problem for synchronous communication with inputs and outputs is given by the following equation, where the given behavior predicates C_i (for i = A or C) have the form $C_i = C_i^{Ass} \Rightarrow C_i^{Guar}$:

$$[\, C_B(x_A, x_C) \,] = \text{hide}^{(syn)}{}_B [\, C_A(x_B, x_C) \wedge C_C(x_A, x_B) \,] \setminus \text{hide}^{(syn)}{}_B [(\neg C_A{}^{Ass}(x_B, x_C) \vee C_A{}^{Guar}(x_B, x_C)) \wedge (C_C{}^{Ass}(x_A, x_B) \wedge \neg C_C{}^{Guar}(x_A, x_B)) \,] \quad (5^{synP\text{-}IOA})$$

Equation ($5^{synP\text{-}IOA}$) introduces an additional requirement on the behavior of the solution, namely, that the output produced towards the component A does not violate the

assumptions $C_A^{Ass}(x_B, x_C)$ made by this component concerning received inputs. In the case of prefix-closed regular behaviors, this requirement can be satisfied by introducing another step in Algorithm 3-1. The idea is to eliminate from the solution B^1 (obtained in Step 5) all transitions whose output may violate the assumptions of component A. Since each state of the solution B^1 represents a set of state pairs of the form (s_C, s_A), the machine B^1 "knows" in which state the component A may be. We write $s_B = \{<s'_A, s'_C>\}$ for a state s_B of B where the $<s'_A, s'_C>$ are the associated state pairs (see Step 4 of Algorithm 3-1). Therefore, the following algorithm should be performed after Step 5 of Algorithm 3-1:

Algorithm 5-1 (for avoiding non-specified inputs for component A):

For each state $s_B = \{<s'_A, s'_C>\}$ of B^1 and each transition t from s_B, check that for all pairs $<s'_A, s'_C>$ associated with s_B , there are transitions from s'_A in A that may accept all input interactions that may be produced by B in state s_B and by C in state s'_C If this condition is not satisfied for some transition t, prune transition t using Algorithm 5-2 below. At the end, eliminate all states of B^1 that have become unreachable from the initial state and all transitions from these states.

Because of the unit delay assumption, the possible outputs produced by a transition from the current state should not depend on the inputs that trigger the transition. Therefore, if in state s, there is a transition producing certain combination of values at the output interfaces, then there should be transitions producing the same combination of output values for all combinations of input values that are accepted in state s. These different transitions may, however, lead to different next states. In the following, we write I_t for the combination of input values that trigger a transition t, and O_t for the combination of output values produced by t.

In order to preserve the unit delay assumption during transition pruning, we propose the following algorithm:

Algorithm 5-2 (pruning transitions - synchronous input-output communication):

Input: a set of transitions of the solution state machine B (to be eliminated).

Step (a): For each transition t do the following. (Note: the transition cannot be simply deleted because the input interactions are not controllable by component B; the machine must be able to accept the input in the starting state of the transition. Therefore we may have to eliminate the current state.)

> If there is another transition t' in the starting state of t with $I_{t'} = I_t$ and $O_{t'} = O_t$ (but may enter a different state) then delete transition t.
>
> Else, if there is another transition t' in the starting state of t with $I_{t'} = I_t$ and $O_{t'} \neq O_t$ then all transitions t'' from the starting state with $O_{t''} = O_t$ should be deleted (including t).
>
> Else, prune all transitions that lead to the starting state of t. This case introduces a recursive process and will eventually lead to the elimination of the starting state of t together with all other transitions starting in that state.

Step (b): Delete any state from the state machine B that is not any more accessible from the initial state. Also delete any outgoing transitions from these states.

As an example we consider a variation of the example from Figure 4. We assume that the interface I_A represents input to component B, interface I_B represents output from component A, and interface I_C represents output from B which is input to A, as shown in Figure 8(b). The behaviors of C and A are as shown in Figure 4 (a) and (b). Note, however, that A assumes that in state 1 no input c_2 nor c_3 will occur, in state 2 no input c_1, and in state 4 no input c_1 nor c_3. Therefore, the transition (a_1,i,c_3) of the solution in Figure 4(d) will result in undefined input for A in the joint state (4, (1,2 or 2,4)) shown in Figure 4(e). This transition would therefore be pruned by Algorithm 5-1 which, in turn, reduces the realized subset of C. We note that in this example, all additional transitions included in the maximal solution (as compared with the reduced maximal one) lead to non-specified input for A and will therefore be pruned.

It is interesting to note that the additional transitions provided by the maximal solution (as compared with the **reduced** maximal solution considered here) are not very useful. Under the unit delay assumption, they are either not executable because prevented by component A (as already mentioned in Section 3.3), or they give rise to undefined input for component A and must therefore be pruned. In the example considered above, all additional transitions (for details, see Section 3.3) lead to undefined input, since the interface I_C goes from B to A.

We note that Algorithms 3-2 and 3-4 for finding prefix-closed solutions and solutions without deadlocks, respectively, can be used in the context of input-output interactions, But Algorithm 5-2 must be used for pruning transitions, instead of Algorithm 3-3.

5.2.2. Partially specified IOA with interleaving semantics

Similar to Section 5.2.1, in this case the equations (1^{LTS}), (3^{LTS}) and (5^{LTS}) can be applied where the behavior predicates C_i (for i = A, B and C) have the form $C_i = C_i^{Ass} \Rightarrow C_i^{Guar}$. For the reduced maximal solution one obtains the same formula as $(5^{syn\text{-}IOA})$ above, except that the interleaving hiding operator $hide^{(LTS)}$ is used instead of the synchronous hide. For prefix-closed regular behaviors, Algorithms 3-1, 3-2 and 3-4 may be adapted as explained in Section 4.2. Algorithm 3-3 for pruning transitions is replaced by Algorithm 5-3 below, and Algorithm 5-1 for avoiding undefined input for component A is also easily adapted to interleaving semantics.

The transition pruning algorithm is much simpler than in the case of synchronous communication, because with interleaving, each transition has only one interaction, either input or output, and there is no need for the unit delay assumption. The following algorithm can be used:

Algorithm 5-3 (for pruning transitions – interleaved input-output communication):

Input: a set of transitions of the solution state machine B (to be eliminated).

Step (a): For each of the given transition t do the following:

If t produces an output, delete the transition from B.

If t is triggered by input do the following. (Note that the transition cannot be eliminated because input interactions are not controllable by component B).

Prune all transitions that lead to the starting state of t. (Note, this is a recursive process and will eventually lead to the elimination of the starting state of t together with all other transitions starting in that state).

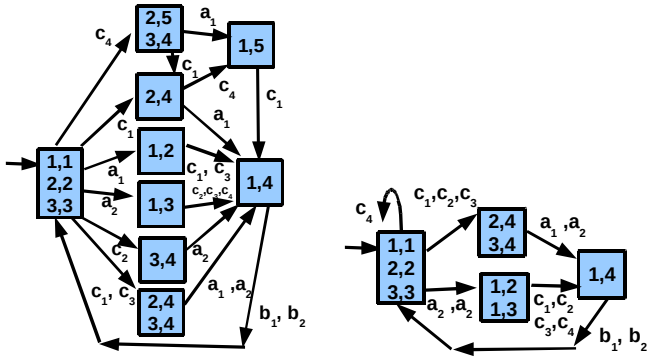
Step (b): Delete any state from the state machine B that is not any more accessible from the initial state. Also delete any outgoing transitions from these states.

An example similar to the example for LTS submodule construction is shown in Figures 9. The main difference with the system shown in Figure 7 is the fact that each interaction has a direction, representing output for one side and input for the other, as shown in Figure 8(c). Using the diagrams of Figure 7(a) and (b) for the definition of the behaviors of C and A, we obtain the diagram of Figure 7(d) as reduced maximal solution. This solution does not generate any undefined input for A; in fact, it does not send any output to A. As discussed above, one would normally prune the transitions c3 and c4 because they lead to deadlocks, but since they are input transitions, they can only be pruned by eliminating their starting state. However, their starting state is the initial state. Therefore there is no solution without deadlock.

In order to avoid these deadlocks in this example, one has to make changes to the given specifications of C or A. One may, for instance, delete the c_3 and c_4 transitions from the behavior of A. Or, if one inverts the direction of these interactions, that is c_3 and c_4 become input to A and output for B, we may delete these transitions from the behavior of B (see Figure 7(d)). In this case, there is another reason for deleting them: they lead to undefined inputs, if they are executed from the initial state before A does its b_1 or b_2 transitions.

We note that we get a completely different system when the direction of the I_A and I_B interfaces are exchanged. In this case, we can assume that the environment will provide an *a*-interaction that matches the preceding *b*-interactions. Note that in the case considered above, it was the responsibility of the joint behavior of A and B to produce a_1 after b_1 and a_2 after b_2. Now we obtain the reduced maximal solution shown in Figure 9(a). We note that in this case the labels a_1 and a_2 written next to the states of Figure 7(a) and (b) can be dropped, since they represent forbidden transition labels from those states, which is, in this case, enforced by the environment, by assumption.

The concept of "maximal solution" in the context of specifications with assumptions and guarantees means that the guarantees are weakest and the assumptions strongest. For instance one could propose the much simpler solution shown in Figure 9(b); however, it makes fewer assumptions about the order between *c* and *a* interactions.



(a) (b)

Figure 9: Two solutions to the submodule construction problem for behaviors of C and A as shown in Figure 7 and the interfaces of Figure 8(c), but input-output interchanged for interfaces I_A and I_B

6. CONCLUDING DISCUSSION

The problem of submodule construction (or sometimes called equation solving) has some important applications for real-time control systems, communication gateway design, testing of embedded components, and component re-use for system design in general. Several equations and algorithms for solving this problem have been presented in the literature based on different specification formalisms for the dynamic behavior of the desired global system and the existing submodule, and depending on different communication patterns between the different system components (see references in the Introduction). In this paper, we have shown that this problem can also be formulated in a more general setting using first-order logic. It turns out that solutions to this problem in logic are quite simple. We show in this paper that these solutions (and their proof of correctness) can be mapped into the different specification formalisms and communication patterns considered in the earlier work. Therefore this paper provides, in a sense, new proofs of correctness for the solutions of the submodule construction problem described in earlier work.

One of the contributions of this paper is to show the similarity of the different equations and algorithms that were developed for the different settings. In logic, the corresponding problem presents itself as a question of equation solving. The basic solution of Equations (3) can be proved through elementary transformations of first-order logic in four lines. The proof that it is maximal is of similar complexity. And the improved, so-called reduced solution of Equation (5) needs a few more lines to be justified. We note that Equation (5^{syn}) for synchronous communication is obtained from Equation (5) by rewriting using the definition of the hiding operator. And Equation (5^{syn}) leads directly to Algorithm 3-1 because, in the case of regular behaviors, the logic operators used in the equation (e.g. complement, product and hiding) correspond directly to operations on regular behaviour definitions, as explained in Section 3.4. Therefore the proof of Equation (5) in the logic context provides at the same time a proof of correctness for Algorithm 3-1. Details are explained in the proof of the algorithm. Algorithm 3-1 is

similar to part of what is described in [Yevtushenko 2000] where Equation (3) is presented in a slightly different notation.

Most papers on submodule construction use either a language-based approach proposed by [Ramage 1989] or algorithms related to Equation (3). We have shown in Section 2.3 that Equation (5) leads to equivalent solutions that are more compact. Equation (5) was first presented in [Bochmann 1980], in a slightly different notation and in an informal manner. A formal proof was given in [Hagverdi 1999].

We show in Section 4 how the equations for synchronous communication, e.g. Equation (5^{syn}), can be converted into the context of interleaving semantics by providing a simple scheme for simulating interleaved communication by synchronous communication. A proof following the same steps as the proof of Equation (5) can be used to prove Equation (5^{LTS}), which has the same form as Equation (5^{syn}), except that the hiding operator for interleaving semantics is used. Because of the same form of the equations, the algorithms for synchronous communication can also be used for interleaving semantics if the composition and hiding operators for interleaving semantics are used.

Much work has also been done on submodule construction for interleaved input/output communication. This includes on the one side much work on designing controllers for discrete event systems (following the approach of [Ramage 1989]) where one may consider that inputs to the controller are not controllable while outputs are controllable. On the other side, there are several works that consider specifications in the form of IOA [Lynch 1989], IOTS [Tretmans 1996] or interface automata [DeLuca 2001] and present solutions to the submodule construction problem [Qin 1991, Drissi 1999 and 2000, Bochmann 2002, Bhaduri 2008]. [Drissi 2000] uses an approach like Equation (3) and also talks about optional progress properties. The proofs in these papers are relatively complex. Therefore I think that this paper provides new, simple proofs of correctness for several earlier approaches to submodule construction.

The problem of submodule construction is addressed in this paper in its most simple architecture, as shown in Figure 2(b). In practical applications, one often encounters the problem in slightly more general architectures, such as the following:

- More than two system components: Let us consider the situation where all components but one have a known behaviour, and the remaining component should be designed such that a desired global behaviour is obtained for the overall system. This situation can be reduced to Figure 2(b) by constructing the behaviour of component A in the figure as the composition of all known component behaviors.
- More visibility for the controller: The architecture of Figure 2(b) results in minimal visibility; for instance, the interactions between A and B are not involved in the desired behaviour of C, and component B (e.g. the controller) cannot see the interactions between component A and the environment. There have been various papers that deal with more general situations [Drissi 1999,

Yevtushenko 2000], whereas the basic principles remain the same. However, it is important to note that the submodule construction becomes much simpler when the unknown component B is able to observe all interactions in the system, including all interactions at the interface between A and the environment. In this case the hiding of the interactions at this interface, during Step 3 of Algorithm 3-1, is not necessary and therefore does not introduce any non-determinism (which simplifies all subsequent operations).

This paper provides the following new contributions:

- Formulation of an equation solving problem in logic and equations for the maximal and reduced maximal solutions, together with proofs of correctness (see Section 2). It is also shown that these equations can be applied to the submodule construction problem in the context of the different communication paradigms considered in this paper (see Sections 3, 4 and 5).
- A clarification of the merits of the maximal and reduced maximal solutions (see Sections 2, 4 and 5).
- A component composition paradigm for synchronous rendezvous communication where each system transition implies simultaneous rendezvous interactions at all component interfaces.
- A uniform presentation of algorithms for deriving prefix-closed and deadlock-free solutions (in the case of regular behaviors) and the characterization of transition pruning in the context of the different communication paradigms (see Sections 3.5, 5.2.1 and 5.2.2).
- Algorithms for finding progressive solutions (see Sections 3.6 and 4.2).

We note that the solution algorithms are restricted to regular behavior specifications, however, the solution equations derived for the different communication paradigms are of quite general nature within the context of trace semantics, that is, when the behaviors of the system, and its components, are characterized by sets of possible execution histories. A simple example where the traces of the given component specifications are given by logic properties is discussed in Section 3.2. The solution equations also apply to extended state machines with interaction parameters and state variables, as discussed in [Daou 2005].

There are several works on submodule construction that go beyond trace semantics. One area of concern are hard real-time properties that are included in the specifications (see for instance [Brandin 1994, Maler 1995, Drissi 2000]). Other work deals with conformance relations finer than trace inclusion, for instance [Qin 1991] considers the bisimulation relation; [Tao 1995] considers reduction of nondeterminism, and [Thistle 1995] considers liveness assertions. It is not clear whether the logic-based approach would be useful to describe such situations.

ACKNOWLEDGEMENTS

I would like to thank the late Philip Merlin with whom I started my work in the area of submodule construction. I would also like to thank Nina Yevtushenko (Tomsk University, Russia) for many discussions about submodule construction algorithms and the idea that a generalization of the concept could be found for different specification formalisms. I would also like to thank Bassel Daou for many inspiring discussions on the topic, and finally would like to mention that the work of my former PhD students Z.P. Tao and Jawad Drissi also contributed to my understanding of this problem.

REFERENCES

ABADI, M. AND L. LAMPORT 1995, Conjoining specifications, ACM Transactions on Programming Languages & Systems, vol.17, no.3, May 1995, pp. 507-34.

ABITEBOUL, S., R. HULL AND V. VIANU, Foundations of Databases, Addison-Wesley, 1995.

AHO, A. V., R. SETHI AND J. D. ULLMAN 1986, Compilers, Principles, Techniques and Tools, Addison Wesley, 1986.

BOCHMANN, G. V. AND P. M. MERLIN 1980. On the construction of communication protocols, ICCC, 1980, pp.371-378, reprinted in "Communication Protocol Modeling", edited by C. Sunshine, Artech House Publ., 1981; russian translation: Problems of Intern. Center for Science and Techn. Information, Moscow, 1981, no. 2, pp. 146-155. See also P. MERLIN AND G. V. BOCHMANN 1983, On the Construction of Submodule Specifications and Communication Protocols, ACM Trans. on Programming Languages and Systems, Vol. 5, No. 1 (Jan. 1983), pp. 1-25.

BOCHMANN, G. V. 2002a, Submodule construction and supervisory control: a generalization, in Proc. of Int. Conf. on Implementation and Applications of Automata, Aug. 2001 (invited paper), Springer Lecture Notes, 2002.

BOCHMANN, G. V. 2002b, Submodule construction for specifications with input assumptions and output guarantees, in Proc. FORTE'02 (22st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems), Chapman&Hall, 2002, pp.

BOCHMANN, G. V. 2009, Using first-order logic to reason about submodule construction, Proc. IFIP Intern. Conf. on Formal Techniques for Distributed Systems, Lisbon, Portugal, June 2009, Springer Verlag, LNCS 5522.

BRANDIN, B. A. AND W. M. WONHAM 1994, Supervisory Control of Timed Discrete-Event Systems, IEEE Tran. on Automatic Control, Vol.39, No.2, Feb. 1994.

BROY, M. 1995, Advanced component interface specification, Proc. TPPP'94, Lecture Notes in CS 907, 1995, pp. 369-392.

BUFFALOV, S., K. EL-FAKIH, N. YEVTUSHENKO AND G. V. BOCHMANN 2003, Progressive solutions to a parallel automata equation, Proc. FORTE Conf. (IFIP), Sept. 2003, Berlin, LNCS 2767, Springer Verlag, pp. 367-382.

DAOU, B. AND G. V. BOCHMANN 2005, Submodule construction for extended state machine models, Proc. IFIP Intern. Conf. on Formal Techniques for Networked and Distributed Systems - FORTE 2005, Taiwan, 2005, Springer LNCS 3731, 2005, pp. 396-410.

DE LUCA, A., HENZINGER, T.A. 2001, Interface automata, Proc. 8th European Software Engineering Conf. held jointly with 9th ACM SIGSOFT FSE 2001, pp. 109-120.

DRISSI, J. AND G. V. BOCHMANN 1999, Submodule construction tool, in Proc. Int. Conf. on Computational Intelligence for Modelling, Control and Automation, Vienne, Febr. 1999, (M. Mohammadian, Ed.), IOS Press, pp. 319-324.

DRISSI, J. AND G. V. BOCHMANN 2000, Submodule construction for systems of timed I/O automata, technical report, see also J. Drissi, PhD thesis, University of Montreal, March 2000 (in French).

BHADURI, P., RAMESH, S. 2008, Interface synthesis and protocol conversion, Formal Aspects of Computing 20(2), pp. 205-224 (2008).

HAGHVERDI, E. AND H. URAL 1999, Submodule construction from concurrent system specifications, Information and Software Technology, Vo. 41 (1999), pp. 499-506.

HOARE, C. A. R. 1985, Communicating Sequential Processes, Prentice Hall, 1985.

KELEKAR, S. G. H., Synthesis of protocols and protocol converters using the submodule construction approach, Proc. PSTV, XIII, A. Danthine et al (Eds), 1994.

KIM, T., T.VILLA, R.BRAYTON, A.SANGIOVANNI-VINCENTELLI 1997. Synthesis of FSMs: functional optimization. Kluwer Academic Publishers, 1997.

KUMAR, R., S. NELVAGAR, S.I. MARCUS 1997, A discrete event systems approach for protocol conversion, Discrete Event Dymaic Systems: Theory & Applications, 7 (3), pp. 295-315, 1997.

LARSEN, K.G., XINXIN, L. 1990, Equation solving using modal transition systems, Proc. IEEE Symp. on Logic in Computer Science, 1990, pp.108-117.

LYNCH, N. A. AND M. R. TUTTLE 1989, An introduction to input/output automata, CWI Quarterly, 2(3), 1989, pp. 219-246.

MALER, O., A. PNUELI AND J. SIFAKIS 1995, On the synthesis of discrete controllers for timed systems, STACS 95, Annual Symp. on Theoretical Aspects of Computer Science, Berlin, 1995, Springer Verlag, pp. 229-242.

MISRA, J. AND K. M. CHANDY 1991, Proofs of networks of processes, IEEE Tr. on SE, Vol. SE-7 (July 1991), pp. 417-426.

PARROW, J. 1989, Submodule Construction as Equation Solving in CCS, Theoretical Computer Science, Vol. 68, 1989.

PETRENKO, A., N. YEVTUSHENKO, G. V. BOCHMANN AND R. DSSOULI 1996, Testing in context: framework and test derivation, Computer Communications Journal, Special issue on Protocol engineering, Vol. 19, 1996, pp.1236-1249.

PETRENKO, A. AND N. YEVTUSHENKO 1998, Solving asynchronous equations, in Proc. of IFIP FORTE/PSTV'98 Conf., Paris, Chapman-Hall, 1998.

QIN, H. AND P. LEWIS 1991, Factorisation of finite state machines under strong and observational equivalences, Journal of Formal Aspects of Computing, Vol. 3, pp. 284-307, 1991.

RAMADGE P. J. G. AND W. M. WONHAM 1989, The control of discrete event systems, in Proceedings of the IEEE, Vo. 77, No. 1 (Jan. 1989).

TAO, Z. P., G. V. BOCHMANN AND R. DSSOULI 1995, A model and an algorithm of subsystem construction, in proceedings of the Eighth International Conference on parallel and distributed computing systems, Sept. 21-23, 1995 Orlando, Florida, USA, pp.619-622.

TAO, Z., G. V. BOCHMANN AND R. DSSOULI 1997, A formal method for synthesizing optimized protocol converters and its application to mobile data networks, Mobile Networks & Applications, vol.2, no.3, 1997, pp.259-69. Publisher: Baltzer; ACM Press, Netherlands.

THISTLE, J. G. 1995, On control of systems modelled as deterministic Rabin automata, Discrete Event Dynamic Systems: Theory and Applications, Vol. 5, No. 4 (Sept. 1995), pp. 357-381.

TRETMANS, J. 1996, Test generation with inputs, outputs and quiescence, Proc. 2nd Intern.l Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Springer Verlag, 1996, pp. 127-146.

YEVTUSHENKO, N., T.VILLA, R.BRAYON, A.PETRENKO, A.SANGIOVANNI-VINCENTELLI 2000, Synthesis by language equation solving (exended abstract), in Proc.of Annual Intern.workshop on Logic Synthesis, 2000, 11-14; complete paper in Conference on Computer-Aided Design (ICCAD '01), 2001, pp. 103; see also Solving Equations in Logic Synthesis, Technical Report, Tomsk State University, Томск, 1999, 27 p. (in Russian) or Sequential Synthesis by Language Equation Solving, http://www.cs.berkeley.edu/~bodik/teaching/cs294/papers/language.pdf

YEVTUSHENKO, N., VILLA, T., BRAYTON, R., PETRENKO, A., VINCENTELLI, A.S. **2008,** Compositionally progressive solutions of synchronous FSM equations, Discrete Event Dynamic Systems, 18 (1), 2008, pp. 51-89.