INFORMATICA 2/88

DERIVING PROTOCOLS FROM SERVICES IN THE FINITE STATE MACHINE REPRESENTATION

UDK 519.713

Monika Kapus-Kolar
Inst. Jožef Stefan, Ljubljana

A method is suggested for derivation of protocols from services, based entirely on the finite state machine representation. The method provides several suggestions for human intervention in the design process and thereby for a great variety of solutions. Other benefits are parametrization, transformations for data-flow optimization, uniform treatment of synchronous and asynchronous channels, a uniform approach to composition and decomposition of entities and thereby a uniform approach to design of services and protocols.

Izpeljava protokolov iz servisov ob uporabi predstavitve s koncnimi avtomati. Predstavljena je metoda za avtomatsko konstrukcijo komunikacijskih protokolov za realizacijo podanega globalnega servisa, ki v celoti temelji na predstavitvi s končnimi avtomati. Metoda je zelo primerna za interaktivno delo, ki vodi v široko paleto rešitev. Druge dobre lastnosti metode so parametrizacija, transformacije za optimizacijo pretoka podatkov, enotna obravnava sinhronih in asinhronih kanalov in enoten pristop h kompoziciji in dekompoziciji osebkov, ki omogoča poenotenje načrtovanja servisov in protokolov.

0. Introduction

Derivation of a communication protocol from a given service specification is one of the most challenging problems in the field of computer networks. Two methods have been proposed so far, which we find particularly interesting, because they provide algorithms for totally automatic construction of a suitable protocol. The method, proposed in [Prin], constructs a Petri-net type protocol specification from a finite-state machine service specification, while the method of [BochGotz] is based on attribute grammars. In our paper, we are proposing a method, based entirely on finite state machines, which follows the selection / resolution principle and is therefore also suitable for construction of protocols in a man-machine dialogue.

We assume that a distributed system consists of a set of entities, communicating with each other and the environment through a set of reliable two-point channels. Some of the channels are unbounded FIFOs with unknown delays (asynchronous), while the others are synchronous (the "rendez-vous" type of communication).

A global service, which a system should provide to the environment, is specified by a finite state machine G, with edges representing an asynchronous transmission or reception of a particular message by the system on a particular external channel or a synchronous external event. Paths, leading from the starting state of G, represent the characteristic sequences of system actions.

A message is a tuple of parameters, posessing explicitely or implicitely stated identifiers and values. The crucial observation about parameters is that each parameter identifier, occurring in a specification, represents a global system variable, which is concurrently updated or read by the entities. Parameters, exchanged in an action, are its **output parameters**, while parameters, generating the values of the output parameters, are the **input parameters** of the action. If an action is not an asynchronous transmission, the values of its parameters can also be obtained from the environment. G must possess the following properties:

Property 0.1: It must not contain two non-terminal states S_1 and S_2, such that for every outgoing edge **a** in S_1, leading to a state **S**, there is also the same outgoing edge in S_2, and vice-versa (equivalent states).

Property 0.2: If in a state S_1, there are two paths P_1 and P_2, such that the action sequence of P_2 is a permutation of the action sequence of P_1, respecting all causality relations of P_1 (P_2 is equivalent to P_1), they must both lead to the same state S_2. Path equivalence is formalized in the section 1.

Property 0.3: If there is an edge, representing an action, requiring a value of a particular parameter as an input, then the edge must be preceded from any direction by some edge, generating the value.

The first step in our protocol design method is to convert G into another finite state machine G_P, which mirrors particular design decisions about parallel execution of external actions of a system, while respecting the causality relations of G. Then the states of G_P, requiring communication between entities, are identified. For each such state, a procedure for exchanging messages on internal channels must be provided by a designer. Such procedures explicitate externally invisible transitions of a system, which are initially hidden in the states of G, as indicated in G_C, an extended version of G_P. Note also that it is execution of the internal procedures, that entities without access to external channels are used for. At that point it may turn out that the task can not be solved with the existing channels. By integrating the internal procedures into G_C, another global system behaviour specification G_I is obtained, which is data-flow optimized into G_O and subsequently used for generation of finite state machines for individual entities.

Two algorithms will be used extensively throughout the paper: **Algorithm 0.1**, which introduces to a state machine a new path, and **Algorithm 0.2**, which deletes a particular path.

Algorithm 0.1:

{create a new path P}

```
begin(Algorithm 0.1)
 represent P by a finite set of finite
  segments
 (not all types of the representation might be
  suitable for a particular purpose);
 for every segment, which is a concatenation
     of an action sequence s and an action
     a and should lead from S_I to S_J do
  begin
   if there is no state S_K, different from S_I,
      with a single outgoing edge, namely
      a, leading to S_J, or with a single
      incoming path, namely s, with the
      initial state S_I
   then create S_K as a new state;
   if in S_K, there is no outgoing edge a
   then create an edge a from S_K to S_J
   else
    if the edge a leads to a state, different
       from S_J,
    then exit with error;
   if in S_I, there is no outgoing path s
   then create a path s from S_I to S_K;
   else
    if the path s leads to a state, different
       from S_K,
    then exit with error;
   merge the equivalent states
  end
end(Algorithm 0.1).
```

Algorithm 0.2:

{delete a particular path, leading from S_I to S_J}

```
begin(Algorithm 0.2)
 for every state S of the path do
  begin
   if in S, there are some incoming edges, not
      lying on the path, and also some, lying
      on the path,
   then
    begin
     create a state S_E, equivalent to S;
     redirect the incoming edges of S, not
      lying on the path, to S_E
    end;
 for every edge e of the path do
  if all outgoing edges of the destination
     state of e are lying on the path
  then delete e;
 delete the unconnected states;
 merge the equivalent states
end(Algorithm 0.2).
```

1. Converting a Global Service Specification into an Equivalent Form with a Desired Degree of Parallelism

Service specifications in [BochGotz] use three types of composition (parallel, sequential and alternative). This versatility makes it difficult to identify actions, which could be enabled concurrently, as in a specification, they might lie far apart.

In a finite state machine specification, parallel composition of actions is represented by various permutations of the actions, connecting the same pair of states, with parameters inducing no causality relationship between the actions. The desired degree of parallelism in a global service specification can be achieved by repeated application of **Transformation 1.1**, which increases parallelism, and **Transformation 1.2**, which decreases it.

The idea behind **Transformation 1.1** is that if there is a path P from a state S_1 to a state S_2, the two states may also be connected by all paths, equivalent to P. P_1 **is equivalent to** P_2, iff there is a path P, such that the action sequences of P_1 and P_2 can be generated from P by zero or more applications of **Transformation 1.1**. **Transformation 1.1** generates equivalent paths by repeatedly selecting an action a_2 of the current action sequence and moving it towards the start of the sequence. If in that process a_2 meets an action a_1, such that a_1 is a potential necessary condition for a_2 (**Predicate 1.1**), a_2 may not move any further. We use the word "potential", because G might negate the causality relationship between two actions by providing an alternative path with the two actions in the reverse order.

Predicate 1.1:

(a_1 is a **potential necessary condition** for a_2)

```
begin(Predicate 1.1)
 Predicate 1.1:-
  (a_1 is a synchronous action or a reception
   and
   a_2 is a synchronous action or a trans-
    mission)
  or
  a_1 and a_2 are two actions on the same
   channel
  or
  a_1 generates a parameter value, which is
   read or redefined by a_2
end(Predicate 1.1).
```

Transformation 1.1: If there is a path a_1a_2, connecting S_1 and S_2, and a_1 is not a potential necessary condition for a_2 (**Predicate 1.1**), then it is possible to create (by **Algorithm 0.1**) a path a_2a_1 from S_1 to S_2.

To achieve the highest possible degree of parallelism, **Transformation 1.1** should be applied as long as possible. On the other hand, we want G_P to be a **finite** state machine, but if there is a cycle **C** and an action **a**, such that it can move through the cycle for ever (as no action of the cycle is a potential necessary condition for it) and **C** contains at least two different actions, the set of paths, equivalent to the cycle, is infinite and **can not be described by a finite state machine**. Therefore **Transformation 1.1** must be applied under designer's control.

If a_1 is not a potential necessary condition for a_2, a system is free to execute the actions in the reverse order, because the environment **can not observe it**. **Transformation 1.1** adds a path, which represents execution of the actions in the reverse order, but as the environment can not observe the existence of such a path, a designer is also free to delete it from G by **Transformation 1.2**.

Transformation 1.2: If there is a path a_1a_2 from S_1 to S_2 and a path a_2a_1 from S_1 to S_2 and a_2 is not a potential necessary condition for a_1 (**Predicate 1.1**), then it is possible (by

Algorithm 0.2) to delete the path a_1a_2.

2. Identifying States, Which Require Internal Communication

Some states in G_P might require communication between entities. In this section we introduce **Algorithm 2.2**, which generates another global system specification G_C by extending G_P with internal communication requirements.

Observing the actions, possible in a given state **S**, some of them may be enabled simultaneously and some not. Simply speaking, a set of actions may be enabled simultaneously, if they are in parallel or in exclusive composition. The next design step is to identify in each state **S** a set of exclusive compositions of parallel compositions of multisets of actions, possible in **S**, which might be selected by a system for simultaneous enabling. **Algorithm 2.1**, if not effected by designer's decisions, generates a solution with the highest possible degree of parallelism and minimal amount of internal communication, securing complete implementation of a service. The algorithm should be called systematically from **Algorithm 2.2**.

Algorithm 2.1:

{**Al** : the set of all **alternatives** of a given state **S**}
{Al_G : the set of **groups of alternatives**, which may be selected for simultaneous enabling in **S**}

begin{Algorithm 2.1}
  find **A**, the set of all actions possible in **S**;
  find **Al**, the set of all non-empty multisets of actions in **A**, such that the members of each multiset are in parallel composition and lead to a final state or a state with an outgoing edge, labeled by an action **a**, which is not in parallel composition with the members of the multiset or must not be added to the multiset because of a designer's decision
  {members of a multiset are in parallel composition, iff they may access their parameters simultaneously, each permutation of them is represented by an outgoing path in **S** and any two prefixes of the paths with the same multiset of actions lead to the same state};
  find Al_G, the set of all subsets of **Al**, which are maximal in respect to the following property **P** (for special control purposes, a designer may also decide to cover **Al** with subsets, which do poses the property **P**, but are not maximal):
  {a subset **X** of a set **Y** is maximal in respect to a property **P**, iff it has the property **P**, but can not be extended by any other members of **Y** without loosing the property}
  if all members (alternatives) of a member **X** of **Al** are enabled simultaneously, the entities, participating in their execution, are always able to select one of the alternatives without any internal communication
  (the global decision is equivalent to a set of local decisions)
end{Algorithm 2.1}.

Al in **Algorithm 2.1** answers the question, which actions may be enabled simultaneously, because they are in parallel composition, but one has to be careful. First, although we wish to enable simultaneously as many actions as possible, strict application of that rule might lead to an incomplete implementation of a service. Second, if in a state **S**, there is a loop with all edges labeled with the same label, it is possible to define an infinite **Al**, which requires careful definition of Al_G and careful construction of paths in **Algorithm 2.2**.

The idea behind grouping of alternatives is that a global decision procedure for selecting an alternative for actual execution might to some extent be performed as a set of local decision procedures. Respecting the property minimizes the amount of internal communication and at the same time provides a solution to the problem that actions for further execution can only be discussed among entities in terms of their a priori properties (as the only a priori property of a reception is its channel, it might not be possible to distinguish between two alternatives).

If only a partial implementation of a service is required, **Algorithm 2.1** is the most suitable point for human intervention. Partial implementations can be generated by definition of incomplete sets of alternatives or groups of alternatives.

Algorithm 2.2:

begin{Algorithm 2.2}
  Open:= [starting state of G_P];
  Closed:= [];
  G_C is just the starting state of G_P;
  while not Open=[] **do**
   begin
    move a state S_D from **Open** to **Closed**;
    find (by **Algorithm 2.1**) **Al**(S_D) and $Al_G(S_D)$;
    for each member A_G of $Al_G(S_D)$ **do**
     begin
      if A_G is not the only member of $Al_G(S_D)$ or special guarding is required
      then add to G_C a τ_D edge from S_D to a new state S_I
        (a state is new, iff there is no state with the same name neither in G_P nor in G_C)
      else S_I:= S_D;
      find **I**(A_G), the set of input parameters of A_G;
      if I(A_G) is not empty
      then
       begin
        for each member **In** of I(A_G) **do**
         begin
          find **U(In)**, the set of entities, using the value of **In** in execution of A_G;
          find K(**In**), the set of entities, knowing the value of **In**
         end;
        create an edge τ_P from S_I to a new state S_A
       end
      else S_A:= S_I;
      {create in S_A a graph G_A, representing execution of A_G:}
      for each outgoing path of S_D in G_P, representing execution of one of the members of A_G, **do**
       create the same outgoing path in S_A in G_C
       {Add as few new edges as possible (**Algorithm 0.1**), but keep paths, belonging to different groups of alternatives, disjoint. Do not use in G_A any old state names.};
      find **Pr**(A_G), the set of all entities, participating in execution of A_G;
      for each member **E** of **Pr**(A_G) **do**
       select **T(E)**, the set of all action sequences with a length>=0, executed by **E** as part of execution of A_G, after which **E** might decide to abandon execution of A_G and enter a synchronization procedure
       {although **T(E)** is selected by a designer, it has some mandatory members: the sequences, after which **E** has no asynchronous transmission to execute in A_G};

```
     find T, the set of all synchronization
      states of G_A
      {S is a synchronization state of G_A, iff
       for every entity E, the projection of a
       path from S_A to S on the actions of E is
       in T(E)};
     find T_N, a version of T, in which every
      member is replaced by its old name (the
      name of the equivalent state in G_P);
     for each member S_N of T_N do
      begin
       if S_N is not yet in G_C
       then add S_N to G_C;
       if not S_N in Closed
       then add S_N to Open
      end;
     for each member S_S of T do
      begin
       find its old name S_N;
       create in G_C a τ_S edge from S_S to S_N
      end
    end
  end;
  terminal states of G_C := terminal states of G_P
 end{Algorithm 2.2}.
```

Each edge τ_P requires execution of a **parameter distribution procedure**.

Each state S_D, coming onto **Open** in **Algorithm 2.2**, requires a global decision, what to do next, and is therefore called a **decision state**. If in S_D, there are several groups of alternatives or special guarding is necessary, then S_D requires execution of a **decision procedure**. In G_I, decision procedures are represented as trees of τ_D edges in decision states (S_D).

After a group of alternatives A_G has been selected and enabled, it starts executing. After executing A_G for some time, control of the participating entities is gradually transferred to a **synchronization procedure**. States of G_A, in which **all** the entities might enter a synchronization procedure, are called **synchronization states**. In G_C, synchronization procedures are represented as τ_S edges in synchronization states (S_S). With the help of a synchronization procedure, a system synchronizes to a state S_N of G_P, which corresponds to the currently active synchronization state.

The aim of firing a synchronization procedure after successful execution of one of the enabled alternatives is distribution of the knowledge that the actions, guarded by the alternative, are now enabled. The aim of firing a synchronization procedure before successful execution of any of the enabled alternatives is resynchronization of a system, after which another group of alternatives may be selected. This might be necessary, if the environment is not forcing the same group of alternatives as the system and does not cooperate promptly.

To minimize the amount of internal communication, an entity should fire a synchronization procedure only when it has no other action to execute without cooperation of the environment, but in principle, a designer might also define some additional synchronization states. When entering a synchronization procedure, the entity does not know, which of the enabled actions have already been executed by other entities. Therefore definition of synchronization states should be consistent, as stated in **Algorithm 2.2**.

If in a decision state, there are several groups of alternatives and a system is trying to execute one of them by repeatedly selecting a group, trying for some time to execute it and (if not successful) resynchronizing, some actions are enabled infinitely often, but not all the time. If the pending actions are synchronous, this is a degradation of fairness of the system, which is due to a particular distribution of external channels among the entities.

3. A General Design Method for Internal Procedures

The next task is to construct a finite state machine G_D by integrating into G_C the necessary internal procedures. G_D should represent the total behaviour of a system in a concise style, similar to that of G_P.

In G_P, all actions are on external channels, which have two end-points, but are observed only from the side of the system, while the actions, constituting internal procedures, are on internal channels with both end-points within the system. An action on an asynchronous internal channel actually consists of two events: transmission of a message and reception of the message. To retain the specification style of G_P, all actions should be represented in G_D as single events and their granularity should not become apparent before projecting G_D onto individual entities.

Let's ignore for a moment the external actions of a system and concentrate on its internal actions - the protocol. We argue that **a general purpose protocol should be specified by a single deterministic finite state machine P, representing the characteristic sequences of message transmissions and synchronous events.** In that way, a designer is forced to concentrate entirely on inter-entity causality relations of the protocol and not to rely upon intra-entity causality relations, which should be treated as **implementation details**. The approach is a direct application of the "empty medium abstraction" heuristic, which has proved to be useful for protocol verification, to protocol synthesis.

Specifications of individual entities can be generated from a global protocol specification P by **Algorithm 3.1** and **Transformations 1.1** and **1.2**. **Algorithm 3.1** projects P on one of the entities E, so that all actions on its incoming channels become receptions. Then **Transformations 1.1** and **1.2** are applied to specifications of individual entities to obtain the desired degree of intra-entity parallelism. In the two transformations, E represents a system, and the entities, cooperating with it, represent its environment.

Algorithm 3.1:

{projecting a global protocol specification P onto an individual entity E}

```
 begin{Algorithm 3.1}
  while applicable do
   begin
    if there is an edge from S_1 to S_2, labeled
       by an action on a channel, which is not
       connected to E
       or
       there is an edge a from S to S_1 and an
       edge a from S to S_2
    then merge S_1 and S_2 into a single state;
    if there are two or more a edges from S_1 to
       S_2
    then replace them by a single a edge
   end
 end{Algorithm 3.1}.
```

Application of **Transformations 1.1** and **1.2** might result in several different sets of individual entity specifications. But this ambiguity of a global protocol specification P is not a deficiency of the specification method: As delays of all asynchronous channels are totally unknown, the sets can not be distinguished by observing the entities for a finite period of time, hence the ambiguity is immaterial and any attempt to remove it (by explicitly mentioning asynchronous receptions in the global state machine or by specifying the protocol by a set of local state machines) is an **overspecification** and should be avoided.

The basic problem in protocol synthesis is to avoid deadlocks, unspecified receptions and unspecified parameters. When designing a global protocol specification of our type, those design errors can be avoided by respecting five simple common sense **Rules 3.1** to **3.5**.

Considering only the basic semantics of a state machine, each node represents an exclusive composition of the outgoing paths, but in protocol specification, there is also another, equally important type of composition - the parallel composition of actions. Parallel composition of actions can be described by exclusive composition of their permutations, but this mental task is not trivial enough to be carried out subconsciously. A potential deadlock or an unspecified reception occurs whenever some actions are in parallel composition by the nature of the system architecture, but that fact is not properly described by a state machine, usually because a designer is not aware of the existence of the parallel composition.

Rules 3.1 and **3.2** define paths, which must mandatory be specified, while **Rules 3.3** to **3.5** define some mandatory properties of the specified paths.

Rule 3.1: If **A** is a subset of actions, which are labels of the outgoing edges of a state **S**, such that every entity participates in execution of at most one member of **A** (an asynchronous transmission has one participant, the sender, and a synchronous action has two participants) - the actions are in parallel composition, then every permutation of the members of **A** must be represented by an outgoing path of **S**, as no entity is allowed to make any assumptions about execution of the actions of other entities, which it is not guarding. In the case of parametrization, any two actions, possible in a state **S**, on different channels, which are not both synchronous, must also be considered as in parallel composition and obey **Rule 3.1**, although the actions share a participant. This is to guarantee the soundness of **Rule 3.5**.

Rule 3.2: If in a state **S**, there is an outgoing path a_1a_2 and, by **Rule 3.1**, an outgoing edge a_2 must not be created in **S** without creating an outgoing path a_2a_1, then the path must actually exist.

Rule 3.3: If in a state **S**, there are two outgoing paths with the same multiset of actions **M**, such that no two different members of **M** belong to the same channel and no two different synchronous members of **M** share both participants, then the two paths must lead to the same state, as no entity can communicate to the rest of the system any information about the order, in which it has executed the actions of **M**.

Rule 3.4: Projection onto any entity must have **Property 0.3**.

Rule 3.5: If two actions are in parallel composition and one of them is generating a value of a parameter, then the other must neither read nor redefine the value.

Formal proof of the rules if outside the scope of the paper. Intuitively, they prevent unspecified receptions, because receptions are hidden in transmissions, they prevent deadlocks, because there is no state without transmissions and they guarantee coordinated progress of all participating entities, because any assumptions about non-existing information exchanges are avoided.

Returning to our original task, we point out that the initial service specification **G** for a system **S** under design should be obtained by the same method. **S** should be considered as an entity of a wider **closed** system **W**, consisting of **S** and the relevant entities, external to **S**. A designer should first specify a "protocol" for the system **W**, so that he is forced to think about implications of communication on the channels, connecting entities, external to **S**, on the service requirements for **S**. Then **G** can be generated by **Algorithm 3.1**.

As suggested in the section 4, the method should also be used for design of internal procedures, introduced by G_C.

4. Design of Parameter Distribution, Decision And Synchronization Procedures

In the section 2, we have defined three types of internal procedures: parameter distribution procedures, decision procedures and synchronization procedures. The nature of a protocol is mainly determined by decision procedures, while procedures of the other two types only play an auxiliary role. In our method, design of internal procedures and their integration is guided by eight basic heuristics:

Heuristic 4.1: Initially, each internal procedure should appear in the specification separated from the others. Message merging is subject to the final optimization (section 5).

Heuristic 4.2: An internal procedure should initially be scheduled just before its results are necessary. Earlier scheduling is subject to the final optimization.

In particular, parameter distribution procedures are inserted in G_C instead of τ_P edges. Decision procedures are inserted in G_C instead of τ_D trees, so that the starting state of a procedure is located at the root and its terminal states at the leaves of a tree. For synchronization procedures, the simplest kind of their integration into G_C is a bit more complicated and will be discussed later. The place for their integration is indicated by τ_S edges.

Heuristic 4.3: To prevent harmful re-ordering of messages, belonging to various internal procedures, during their transport, all participants of an internal procedure must agree on its termination, so that the internal procedures can be treated as atomic. Note that this is a general solution to the problem, described in the section 3.3 of [BochGotz]. If some of the messages are redundant, they can be deleted in the final optimization, which might sometimes result in the solution from [BochGotz].

Heuristic 4.4: The main point in design of an internal procedure is to determine for each of its terminal states T the **synchronization set** **Sy**(T), the set of all entities, which must know that the system will progress through T. As at that point of design, internal procedures are scheduled just in time, the members of a synchronization set **Sy**(T) are exactly the entities, executing the actions, possible in T. When the participants of an internal procedure have reached an agreement on its termination (which is in a terminal state T), the members of **Sy**(T) must know, that the execution has terminated in T.

Heuristic 4.5: As the basic aim of an internal procedure is to lead a system to a particular state, it should be designed as an exchange of proposals about the terminal state, which the procedure should reach, and sets of terminal states, suggested by various participants, should be explicitly visible in the messages, so that the terminal state, which a path is leading to, can be calculated as an intersection of the sets, exchanged along the path. Beside that, terminal states **must** appear in the messages with the same names as in G_C. If the requirements are too rigorous, they can be overcome in the final optimization.

Heuristic 4.6: If an internal procedure is a parameter distribution procedure, it must communicate the necessary parameter values from the members of the relevant K sets to the members of the relevant U sets (see **Algorithm 2.2**).

Heuristic 4.7: We require that internal procedures are provided by a designer (in the spirit of the section 3), but this is not a serious drawback for the automatization of the protocol design process, as in practice, decision procedures, and even more procedures of the other two types, are drawn from a small set of types, which can be pre-constructed and used with suitable parameters, whenever necessary. An internal procedure must respect **Rules 3.1** to **3.5**, where **Rule 3.4** must be checked in regard to the rest of the system specification.

Internal procedures can not be designed in an optional order. The algorithm is the following:

1. Determine synchronization sets of parameter distribution procedures and design the procedures.
2. Determine synchronization sets of decision procedures and design the procedures.
3. Determine synchronization sets of synchronization procedures and design the procedures.

Now we are ready to define an algorithm for integrating into G_C a synchronization procedure. Observing a graph G_A, generated by **Algorithm 2.2**, it is not sufficient to replace by some procedures the τ_B edges in its synchronization states. The whole G_A, together with its τ_B edges, must be replaced by a graph G_B (the starting state of G_B is the starting state of G_A, the terminal states of G_B are those, pointed to by τ_B edges), concisely representing the action sequences of the expression:

$$|_{E \in Pr(A_G)} (+_{s \in T(E)} (s.P(s)))$$

The expression has the following meaning: For each member **s** of a T(E), design an internal procedure P(**s**), put s and P(s) into sequential composition, put the expressions, belonging to various members of T(E), into **or** composition, then put the expressions, belonging to various member of $Pr(A_G)$, into parallel composition.

With other words: each entity **E**, participating in execution of an A_G, executes an action sequence **s**, mandatory followed by a procedure **P(s)**, which distributes the knowledge of **E** about **N**, the set of the possible terminal states of G_B, as known by **E** after execution of **s**, to the members of the union of the synchronization sets of those states. **M** is a member of **N**, iff in G_A, there is a synchronization state **S**, connected with **M** by a τ_B edge, reachable from the starting state of G_A by a path, whose projection onto **E** is **s**.

The terminal state **T**, to which a path of G_B should lead, can be determined from the path by **Heuristic 4.5**. The requirements of **Heuristics 4.3** and **4.4** must be fulfilled on G_B as a whole. It turns out, that it is sufficient to fulfil **Heuristic 4.4** for each P(s), but for **Heuristic 4.3** that might not be true. Hence, it is necessary to "blow" each terminal state T of G_B into a termination agreement procedure for all entities, participating in G_B. Procedures in all terminal states of G_B must be the same.

The principles, used in the design of G_B, lead to another heuristic for construction of internal procedures:

Heuristic 4.8: The first step in design of an internal procedure is to identify the knowledge, which is to be communicated. For each piece of knowledge (which might be a parameter value or a set of suggested terminal states), construct a procedure, which conveys the knowledge from its source to its destination. Put all such procedures into parallel composition and finally put the resulting procedure into sequential composition with a termination agreement procedure for all potential participants.

The result of the integration of internal procedures into G_C is a finite state machine, which might have some equivalent states, that have to be merged. Beside that, it might be necessary to introduce some new paths, required by **Rules 3.1** and **3.2**. As shown in the section 5, the resulting machine G_I is further optimized into G_O.

5. Final Optimization of a Global Service Provider Specification

Final optimization is performed by application of **Transformations 5.1** to **5.4**. The transformations address **Rules 3.1** to **3.5**, which use the notion of an action participant. The external actions of a system (those from the initial service specification) must be treated as internal actions of particular entities, which are their only participants. The transformations may only be applied, if they do not change the order of external actions.

Transformation 5.1: If **Rules 3.3** to **3.5** are not violated, then it is possible to introduce (by **Algorithm 0.1**) a particular path and all paths, required by **Rules 3.1** and **3.2**.

The transformation could be used for increasing parallelism or for moving scheduling points of internal procedures.

Transformation 5.2: Let **O** be the set of outgoing edges of a state S, Identify **P(O)**, the set of all paths, mandatory in **S** by **Rule 3.1**. Suppose that a member **a** of **O** is removed from **S**. Identify P(O\a). If **Rules 3.4** and **3.5** are not violated, then it is possible to delete (by **Algorithm 0.2**) the members of P(O) and introduce to **S** (by **Algorithm 0.1**) the members of P(O\a) and all paths, required by **Rules 3.1** and **3.2**.

The transformation could be used for decreasing parallelism or for deleting redundant actions.

Transformation 5.3: If **Rules 3.1** to **3.5** are not violated, it is possible to apply a particular change of edge labels and merge the resulting equivalent states and edges.

The transformation could be used for decreasing the number of message types or for the final naming of messages.

Transformation 5.4: If **Rules 3.3** to **3.5** are not violated, then it is possible to replace (by **Algorithms 0.1** and **0.2**) a path between a pair of states by another path between the same pair of states and then add (by **Algorithm 0.1**) all paths, required by **Rules 3.1** and **3.2**.

The transformation could be used for changing the order of actions or for merging of actions (messages).

Whenever possible, the transformations should be applied to such parts of a finite state machine, that **Rules 3.1** and **3.2** do not induce any new paths or their destination states are determined by **Rule 3.3**. For instance, if an action is executed without knowing, if it will be necessary at all (optimistic scheduling, introduced e.g. by **Transformation 5.1**), the destination state of a new path p_1, which includes an unnecessary execution of the action, must be provided by a designer. The suggested heuristic is to direct p_1 to the same state as p_2, which consists of the same sequence of actions as p_1, except that the unnecessary action is deleted.

6. Conclusions

In comparison to [BochGotz], which generates an unique solution, our method provides several suggestions for human intervention in the design process and thereby for a greater variety of solutions. Other benefits are parametrization, transformations for data-flow optimization, uniform treatment of synchronous and asynchronous channels, a uniform approach to composition and decomposition of entities and thereby a uniform approach to design of services and protocols. Similar conclusions can be drawn when comparing our method to [Prin].

If necessary, the design process can be fully automatized. The only condition is existence of parametrized transport procedures and termination agreement procedures and of some rules, which prevent the process from construction of infinite machines.

References

[Prin] Prinoth,R.: "An Algorithm to Construct Distributed Systems from State-Machines", in Sunshine,C.(ed.): "Protocol Specification, Testing, and Verification", pp.261-282, North-Holland, 1982

[BochGotz] Bochman,G.v., Gotzhein,R.: "Deriving Protocol Specifications from Service Specifications", Proceedings of the ACM SIGCOM Symposium, pp. 148-156, 1986