Converter between incompatible protocols specified using MSC diagrams

Vera A. Zvereva, School of Computing, NUS.

Let us consider two components P and R which are supposed to exchange messages and data. Each of these components sends and receives messages according to a certain protocol. These protocols can be specified using MSC diagrams (Figure 1).

As easily can be seen protocols shown in Figure 1 are incompatible. So the task is to provide an interface that will make their joint work possible.

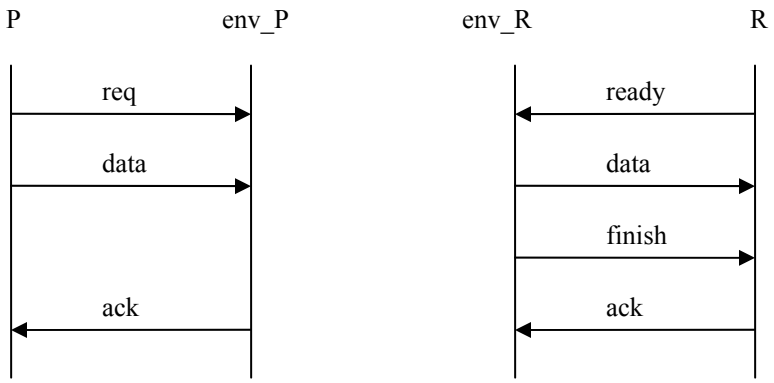

Figure 1

An MSC specifying a protocol contains several lines, each of them shows the events of one component. Let V_P be the set of events located on the line corresponding to the process P. By definition of an MSC the set V_P is linearly ordered. So we can write down the set V_P as $\{v_1^P, v_2^P, \ldots, v_n^P\}$ where n is the number of events in the set V_P ($n = |V_P|$) and $v_i^P \leq v_j^P$ iff $i \leq j$. Actually we are interested only in one line of MSC-diagram which corresponds to component P. Actions located on this line show the order in which P is supposed to send and receive messages. Also the set V_P can be divided into two disjoint sets V_P^I (the set of input actions, or the messages received by P) and V_P^O (the set of output actions, or the messages sent by P).

For further construction let us insert states of the component between input and output actions. They will be denoted as s_i^P, where i shows the number of messages that have been sent or received. The first state s_0^P denotes that no messages have been sent or received yet, and the last state s_n^P means the end of communication (all messages were transmitted). (Figure 2). Being in the state s_i^P component P sends a message v_{i+1}^P (if it

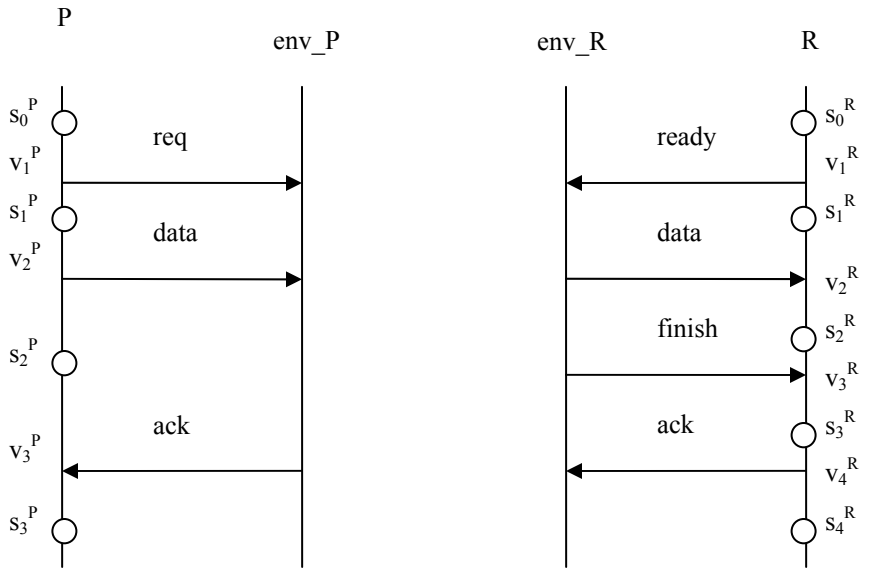

Figure 2

belongs to V_P^O) or expects to receive it (if it belongs to V_P^I) and then moves to the state s_{i+1}^P. Let Q_P be the set of states $\{s_0^P, s_1^P, \ldots, s_n^P\}$.

Definition 1. Intermediate interface

Let us consider two components P and R such that $V_P \cap V_R = (V_P^I \cap V_R^O) \cup (V_R^I \cap V_P^O)$. Intermediate interface for two components P and R is a structure $T = (Q, q_0, q_f V_T^I, V_T^0, \tau_P)$ where

1) Q is the set of states. $Q = Q_P \times Q_R$.
2) $q_0 = (s_0^P, s_0^R)$ is an initial state.
3) $q_f = (s_n^P, s_m^R)$ is a final state ($n = |V_P|$, $m = |V_R|$).

4) $V_T^I = V_P^O \cup V_R^O$

5) $V_T^O = V_P^I \cup V_R^I$

Outgoing messages of the components become the ingoing messages of interface and vice versa. (Figure 3). Let us define $V_T = V_T^I \cup V_T^0$.

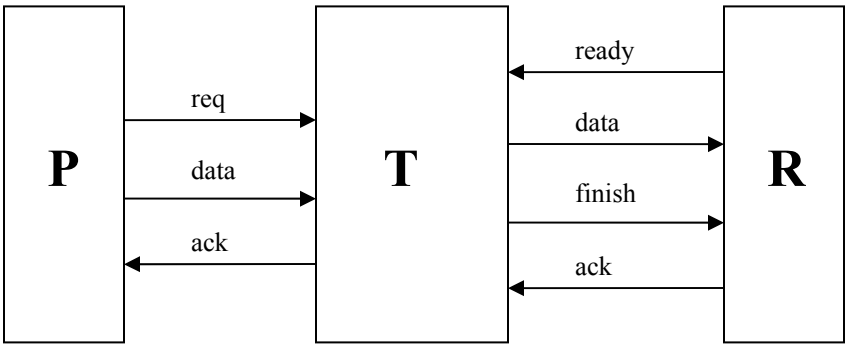

Figure 3

6) $\tau_P \subseteq Q_T \times V_T \times Q_T$ is the set of transitions.

$\tau_P = \{ ((s_i^P, s_k^R), v_{i+1}^P, (s_{i+1}^P, s_k^R))$ where

$v_{i+1}^P \in V_P^O$ or

$v_{i+1}^P \in V_P^I \setminus V_R^O$ or

$v_{i+1}^P \in V_P^I \cap V_R^O$ and $v_{i+1}^P \in \{v_1^R, v_2^R, \ldots, v_k^R\}$, $0 \le i \le |V_P|-1$, $0 \le k \le |V_R|\} \cup$

$\{ ((s_i^P, s_k^R), v_{k+1}^R, (s_i^P, s_{k+1}^R))$ where

$v_{k+1}^R \in V_R^O$ or

$v_{k+1}^R \in V_R^I \setminus V_P^O$ or

$v_{k+1}^R \in V_R^I \cap V_P^O$ and $v_{k+1}^R \in \{v_1^P, v_2^P, \ldots, v_i^P\}$, $0 \le i \le |V_P|$, $0 \le k \le |V_R|-1\}$.

So, if a component wants to send a message it can do it without any restrictions. If it is waiting for a message which does not belong to the set of output actions of the other component then intermediate interface must send the required message by itself. If an expected message is common for two components (in other words, it belongs to the set of output actions of the other component) then interface can send this message only if it have been previously received from the other component.

Some properties of an intermediate interface are listed bellow:

1) The total number of states equals to $(|V_P|+1)\cdot(|V_R|+1)$, but some states may be unreachable. Further, let us refer to this type of intermediate interface as complete one and introduce the concept of refined interface which can be received from intermediate interface by removing all unreachable states.

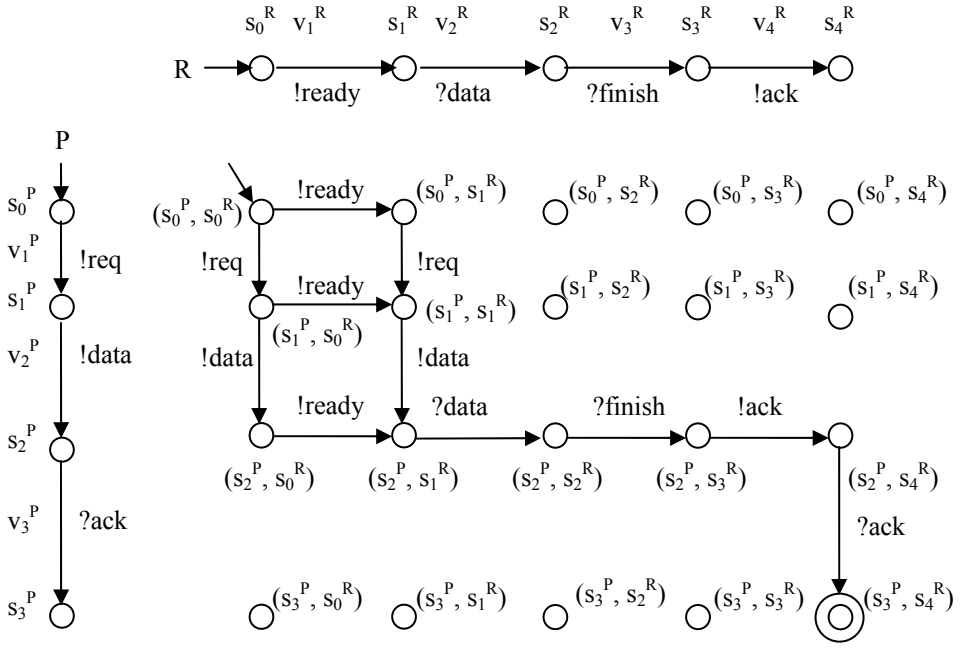

Figure 4

2) For each state $(s_i^P, s_k^R) \in Q_T$ there at most two possible transitions leading to the next state. All states can be divided into groups: when only one of the components (P or Q) can make a move, both of them or none. Then there will be one, two or no transitions correspondingly.

Definition 2. We shall say that protocols of components P and R are adaptable iff the final state q_f is reachable in the complete intermediate interface (i.e., the refined intermediate interface contains the state q_f).

The Figure 4 illustrates a refined intermediate interface for components P and Q shown at Figures 1 and 2. These components are adaptable as the last state (s_3^P, s_4^R) is contained in the refined interface.

Another one example (Figure 5) shows two components that are not adaptable. Each of the components waits for a message from other component, so the intermediate interface cannot produce required message because it is common for two components. And the final state in intermediate interface is not reachable. Figure 6 gives one more example of adaptable protocols.

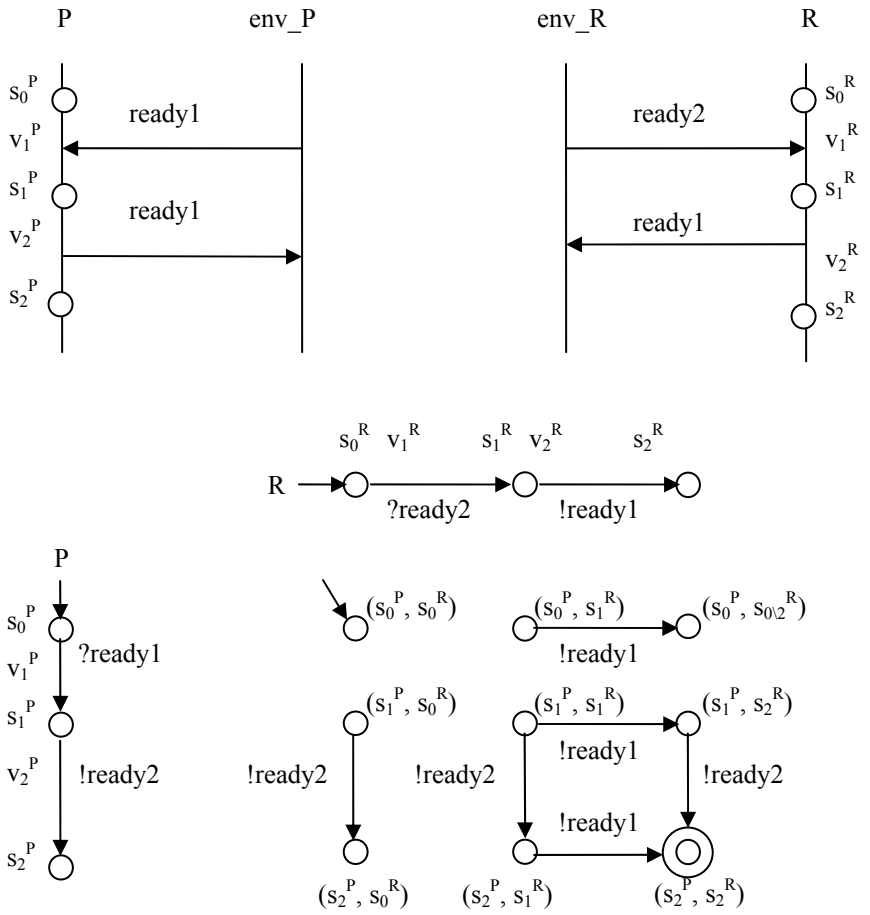

Figure 5

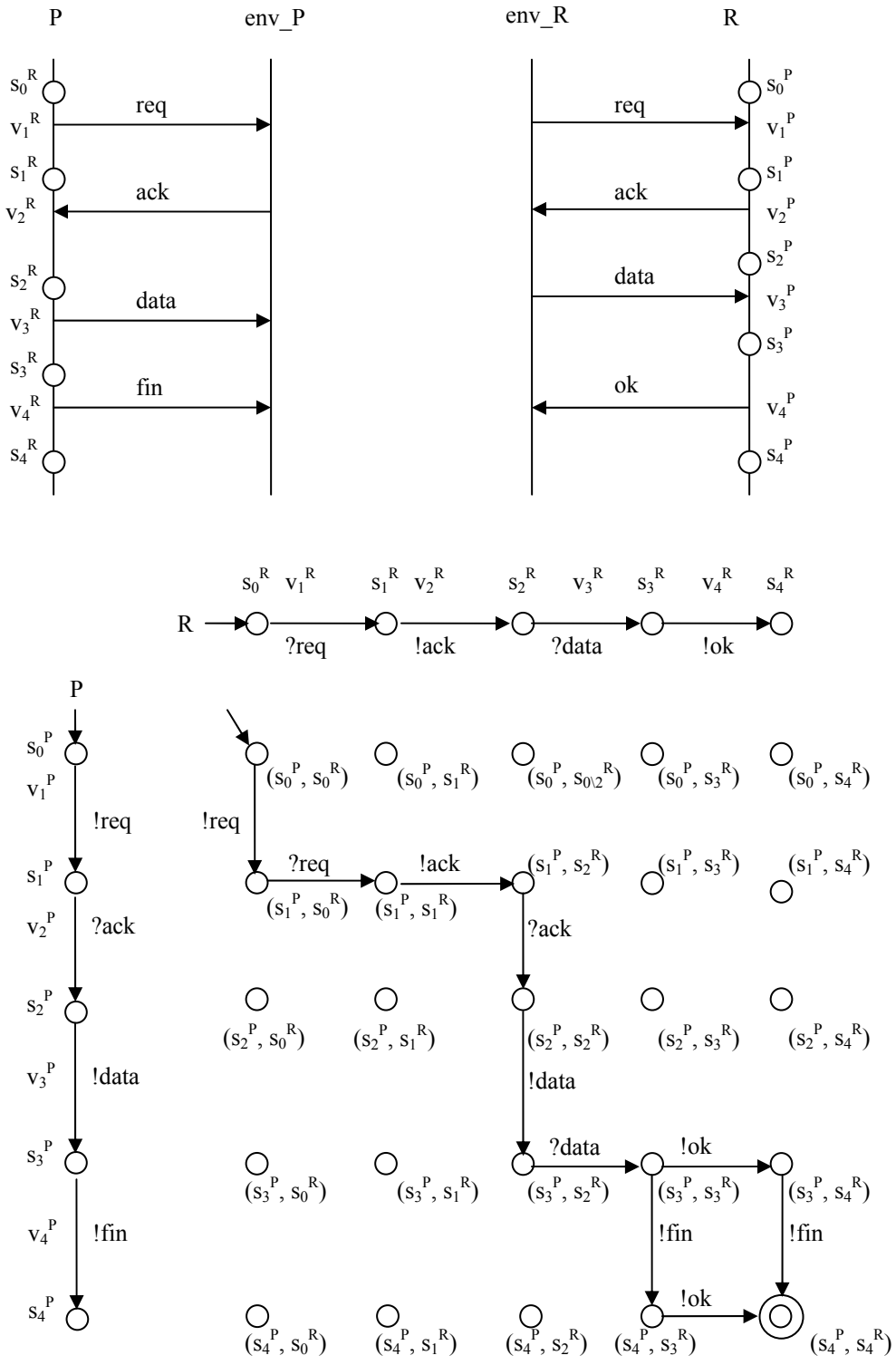
P
env_P
env_R
R
req
ack
data
fin
ok
?req
!ack
?data
!ok
!req
?ack
!data
!fin

Figure 6

Algorithm for constructing an intermediate interface

First of all, note that we need no to construct the complete intermediate interface and then remove unreachable states. Instead we may construct refined interface directly. This can be done by means of recursive procedure with two parameters representing the current states of components.

```
procedure Construct (int i, int k){
if (!containsState(i, k)){
  addState(i, k);
}else break;
if (i < |V_P|){
if ((v_{i+1}^P ∈ V_P^O) OR (v_{i+1}^P ∈ V_P^I \ V_R^O) OR (v_{i+1}^P ∈ V_P^I ∩ V_R^O AND
containsAction(R, k, v_{i+1}^P)) {
              Construct(i+1, k);
              addTransition((s_i^P, s_k^R), v_{i+1}^P, (s_{i+1}^P, s_k^R));
 }
if (k < |V_R| ){
if ((v_{k+1}^R ∈ V_R^O) OR (v_{k+1}^R ∈ V_R^I \ V_P^O) OR (v_{k+1}^R ∈ V_R^I ∩ V_P^O AND
containsAction(P, i, v_{k+1}^R)) {
           Construct(i, k +1);
           addTransition((s_i^P, s_k^R), v_{k+1}^R, (s_i^P, s_{k+1}^R)) ;
}
}
}
```

Procedure `containsState` checks if a state specified by its two-dimensional index has been added to interface before.

Procedure `addState` adds new state specified by its two-dimensional index.

Procedure `addTransition` adds new transition to the interface under construction. It has three parameters specifying the current state, action and the next state.

Procedure `containsAction` checks if message v_{k+1}^R was previously sent by other component. It has three parameters: name of the other component, number of messages sent or received by the component (i.e. its state), and the message v_{k+1}^R.

To construct a refined intermediate interface for two components we need to call the procedure Construct with both parameters equal to zero:

```
Construct(0, 0);
```

Note that only reachable states will be added to the interface and number of recursive calls is $O(|V_P|\cdot|V_R|)$. To check adaptability of the components it is sufficient to check if the last state which index is (n, m) is contained in the interface. If it is true then the refined intermediate interface under consideration will be right the same adapter we intended to build.

The further development of intermediate interface

The current definition of intermediate interface supposes that some components may be not adaptable. Figure 5 illustrates the classical deadlock situation. The further development of intermediate interface definition will be intended to make it capable to cope with such situations. In example shown in Figure 5 intermediate interface could first sent message ready1 to P (though it has not been produced by R yet). Then interface receives ready2 from P and transmits it to R. After that it needs to wait for message ready1 in order to make sure that R behaves according its protocol. So we can say that interface may "borrow" a message from a component before it actually receives it. But this trick can be done not in all cases. Consider a little different example where component exchanges not ready signals but data (Figure 7).

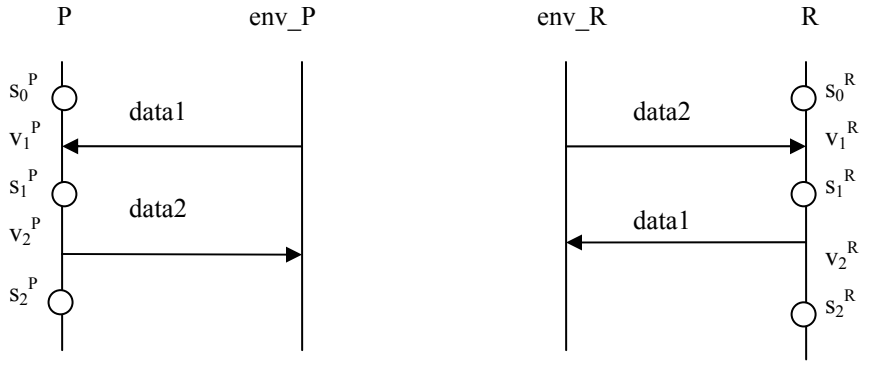

Figure 7

Then intermediate interface obviously cannot generate message data1 or data2 before it is received because it does not know what data the other component actually intends to send. So we come to a conclusion that we need to distinguish between two types of messages: data messages and control ones. The first ones carry a bulk of data, so their content cannot be predicted in advance and an interface has to wait until the message is received before forwarding to the other component. On the other hand, the control messages do not carry data. A component expecting to receive a control message is interested only in fact of its receiving. So an intermediate interface can generate this message instead of a component which is supposed to produce it.

So, let us assume that the set of actions V_P is divided into two disjoint sets V_P^C (control messages) and V_P^D (data messages). Usually each data message is framed with control messages; i. e. components send each other some control messages before data transmission (sending a request for data transfer, receiving readiness acknowledgement) and some messages after data transmission (notification about finishing data transfer, receiving acknowledgment of successful receipt). So we shall group control messages by the names of data messages they correspond to. There will be also pure control messages which do not correspond to any data message. Let V_P^{OD} be the set of outgoing data messages, V_P^{ID} be the set of ingoing data messages, V_P^{OC} be the set of outgoing control messages, V_P^{IC} be the set of ingoing control messages.

More precisely, let us introduce function data_m: $V_P^C \longrightarrow V_P^D \cup \{\varphi\}$ which for a given control message shows the corresponding data message (f(v) = φ means that v is a pure control message which is associated with no data message), and function cont_m: $V_P^D \longrightarrow 2^{V_P^C}$ which for a given data message returns the whole set of control messages associated with it. The control messages associated with a message *data* will be denoted as *sig.data* where *sig* is the unique name among all messages associated with *data*.

In order to make the intermediate interface to be capable to avoid deadlocks in control flow we refuse the name matching of control messages. So if a component expects to receive a message then the intermediate interface must generate this message by itself. And if a component wishes to send a message then the interface must be ready to accept it and will be blocked until it receives the expected message from the component.

On the other hand, the name matching of data messages remains and we suppose that data messages of different protocols those names are equal have the same semantics and there is no incompatibility in data format.

Definition 3. Intermediate interface

So the new definition for intermediate interface of two components P and R will determine it as a structure T = (Q, q_0, q_f V_T^I, V_T^0, τ_P) where

1) Q, q_0, q_f, V_T^I, V_T^O are defined in the same way as before.

2) The set of transitions $\tau_P \subseteq Q_T \times V_T \times Q_T$ is the set of transitions.

τ_P = { $((s_i^P, s_k^R), v_{i+1}^P, (s_{i+1}^P, s_k^R))$ where

$v_{i+1}^P \in V_P^C$ or

$v_{i+1}^P \in V_P^{OD}$ or

$v_{i+1}^P \in V_P^{ID}$ and $v_{i+1}^P \in \{v_1^R, v_2^R, \ldots, v_k^R\}$, $0 \le i \le |V_P|-1$, $0 \le k \le |V_R|\} \cup$

{ $((s_i^P, s_k^R), v_{k+1}^R, (s_i^P, s_{k+1}^R))$ where

$v_{k+1}^R \in V_R^C$ or

$v_{k+1}^R \in V_R^{OD}$ or

$v_{k+1}^R \in V_R^{ID}$ and $v_{k+1}^R \in \{v_1^P, v_2^P, \ldots, v_i^P\}$, $0 \le i \le |V_P|$, $0 \le k \le |V_R|-1\}$.

The difference from the previous definition is that an ingoing control message is generated by an interface even if it has not been received yet. This may seem to be strange and leading to incorrect behavior but later the interface has to wait for this message from the other component and will be blocked until message is received. The only restriction is that both components, P and R, should not be allowed to proceed with their execution until the intermediate interface reaches its last state. Otherwise, it may happen that P has sent and received all messages from its protocol and an intermediate interface is waiting for a message from R. But P considers that protocol execution has finished successfully and begins to execute new task while R may experience some problems.

The data messages are still can be received be intermediate interface from a component without any restriction. But a data message cannot be sent to a component if it has not been generated by another component before. The situation when data message is expected be received by one component but it cannot be produced by the second component is supposed to be impossible because in that case the intermediate interface would be unable to generate the data message to a component which waits for it. So we suppose $V_P^{ID} \subseteq V_R^{OD}$ and $V_R^{ID} \subseteq V_P^{OD}$.

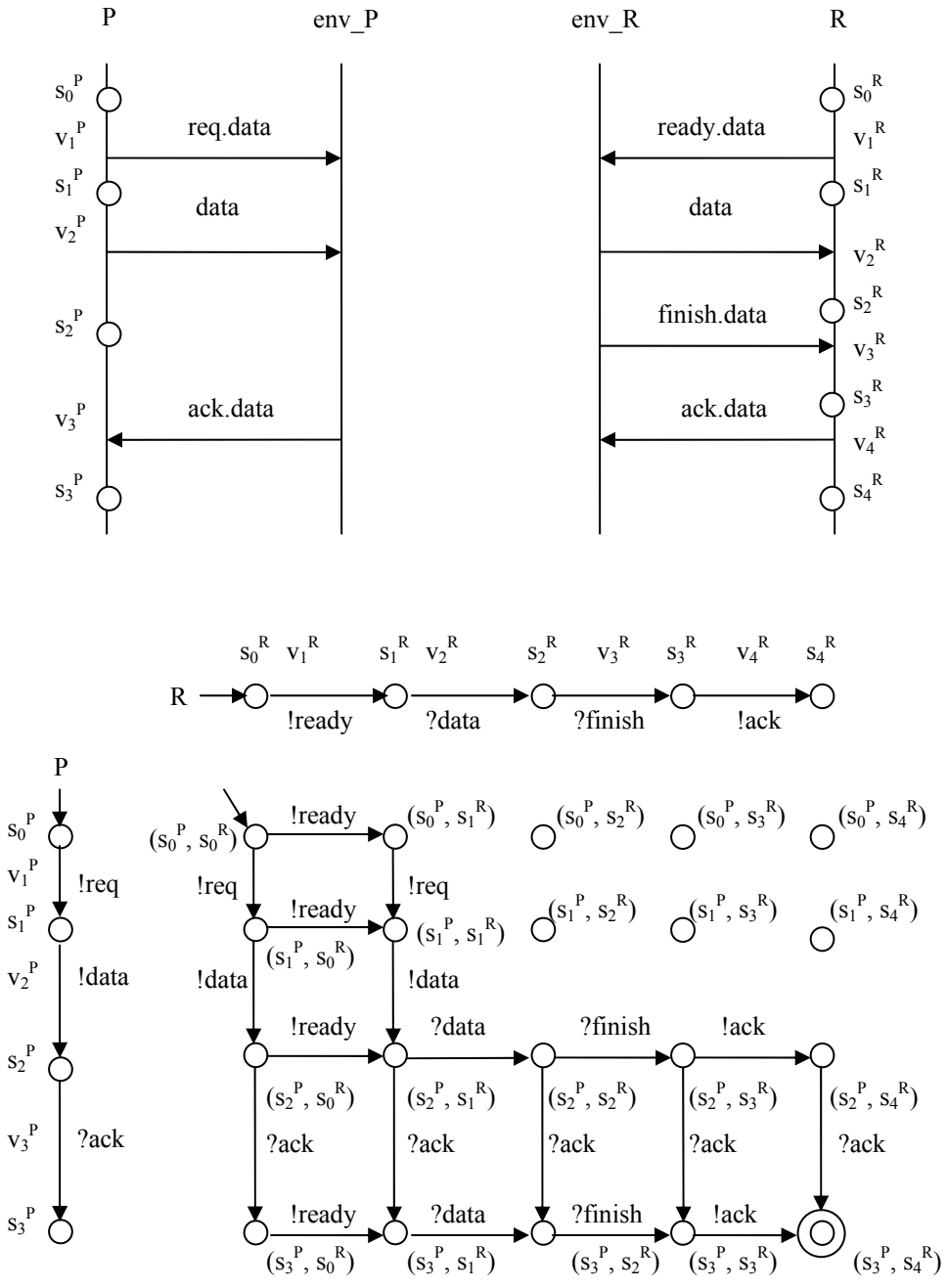

Figure 8

Figure 8 illustrates how the intermediate interface shown in Figure 4 will change. And Figure 9 shows the ability of redefined interface to cope with situations that traditionally are considered as deadlock.

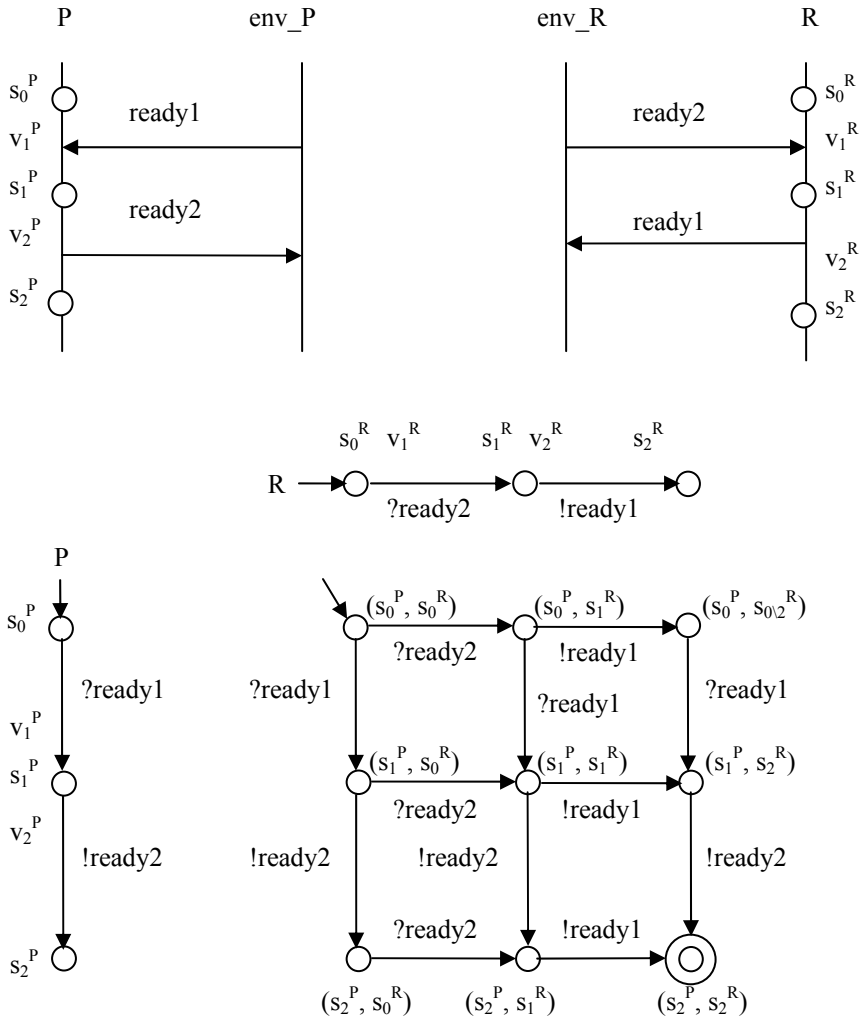

Figure 9

Intermediate interface for MSC-graph.

According to the definition given in [4] an MSC-graph is a graph which nodes are labeled with MSCs and which edges correspond to concatenation if MSCs. Formally, an MSC-graph consists of a set of vertices V, a binary relation over V, an initial vertex, a set of terminal vertices, and a labeling function that maps each vertex v to an MSC. The MSC-graph is interpreted in an operational way as follows. Execution starts at the initial vertex. Next, it continues with a node that follows one of the outgoing edges of the initial node. After execution of the selected node, the process of selection and execution is repeated for the outgoing edges of the selected node. Execution of a terminal node means that execution of the given MSC-graph finishes. If there is more than one outgoing flow line from a node this indicates an alternative. The conditions in MSC-graph can be used to indicate global system states or guards and impose restrictions on the MSCs that are referenced in the MSC-graph nodes.

Taking the definition of an MSC-graph as a basis for our arguments let us consider the directed graph G where each node is labeled with a number of MSCs called transaction. Formally, let *Trans* be the finite set of transaction, with each transaction an index i from a set I = {0, 1, ..., N-1} is associated. So, the total number of transactions is assumed to be equal to N. Let Tr_0 be the initial transaction in the graph and $Trans_f$ be the subset of Trans denoting the final states of the graph. The *Edges* relationship will be defined as follows: Edges $\subseteq$ Trans $\times$ Trans.

Let us introduce the function *Next*: Trans $\rightarrow 2^{Trans}$, next(Tr_i) = { Tr_j | (Tr_i, Tr_j) $\in$ Edges}. For a given transaction Tr_i it shows the set of transactions directly following it.

A transaction Tr_i involves a set of components $Comp_i \subseteq Comp$, where Comp is the set of all components in the system. This set is finite and the total number of components equals to M so that Comp = {P_k | k$\in$1..M}. The function *Agents*: Trans $\rightarrow 2^{Comp}$ shows the set of components involved in a transaction: Agents(Tr_i) = $Comp_i$.

Each transaction Tr_i consists of a set of MSCs {M_i^P | P $\in$ $Comp_i$ }, one MSC for each component. These MSCs define protocols which the components are designed to execute. A component starts its execution with the corresponding MSC from the transaction Tr_0 and then each time chooses one of the transactions following the current one. The concatenation of MSCs will be interpreted in asynchronous way so that one of the MSC agents may execute all its actions and proceed to the next MSC while other agents may have not still finished the execution of previous MSC contrary to synchronous concatenation where no action of the next MSC can be executed until all agents of previous MSC executed all their actions. The main assumption here is that we have a mechanism providing the guarantee that in case of a branch all components will choose the same transaction. For example, in Figure 10 either both components P and Q will choose transaction Tr_1 or they both will choose Tr_2 instead.

During the execution of a transaction components asynchronously exchange data and control messages each of them according to its own protocol specified with MSC. Obviously, the protocols defined by MSCs that are contained in the same transaction may occur to be incompatible. We have already introduced the notion of intermediate interface that can serve as an environment for both components making their joint work

possible. Now the goal is to define the similar concept for the graph of MSCs so that each component passing through a transaction will execute its own protocol regardless of that other components may have different protocols.

Let M_i^P be the MSC from the transaction Tr_i specifying the protocol of component P on the assumption that $P \in Agents(Tr_i)$. If $P \notin Agents(Tr_i)$ we shall assume that M_i^P is an empty MSC not containing any actions.

Let V_P be the set of actions that component P may perform and $V = \bigcup_{P \in Comp} V_P$. $V(M_i^P)$ will denote the set of actions of the MSC M_i^P. As have been mentioned above this set is linearly ordered and $v_k(M_i^P)$ denotes the k-th action of the MSC M_i^P. Then, states are inserted between actions so that $s_k(M_i^P)$ denotes the state when the first k actions from protocol were performed. For an empty MSC we suppose it has the only state $s_0(M_i^P)$. And let $(S(M_i^P))$ be the set of states $\{s_k(M_i^P),\ 0 \le k \le |\ V(M_i^P)\ |\}$. And suppose $S_P = \bigcup_{i \in I} (S(M_i^P) \setminus \{s_k(M_i^P), k = |V(M_i^P)|\})$, i.e. it is the whole set of states of component P except the last states of the MSCs which are identified with the first state of next MSC; and $S = \bigcup_{P \in Comp} S_P$. Suppose $s_P^0 = s_0(M_0^P)$, so that s_P^0 is the first state of component P (the first state of an MSC from the initial transaction). To simplify the notation let us introduce the function $curAct : S \to V$, such that $curAct(s_k(M_i^P)) = v_{k+1}(M_i^P)$, and function $nextState : S \to 2^S$, such that $nextState(s_k(M_i^P)) = \left\{ \begin{array}{l} \{s_{k+1}(M_i^P)\},\ if\ k \neq |V(M_i^P)| - 1; \\ \{s_0(M_j^P) \mid Tr_j \in next(Tr_i)\},\ if\ k = |V(M_i^P)| - 1 \end{array} \right\}$.

The intermediate interface needs to have queues to store temporarily the data transmitted from one component to another. The queues will be represented as lists of actions. If symbol '.' denotes the concatenation operation then the notation $q_1 = q.a$ will mean that the queue q_1 is received from the queue q by putting the action *a* into it so that the action a is the last element in q_1. The notation $q_1 = a.q$ means that action *a* is the first element in q_1 and after removing it from q_1 the queue q will be received. The empty queue will be denoted with symbol λ. Let Q_P be the set of all queues that can be composed from the actions belonging to the set V_P.

A transaction Tr_i may be performed several times by the components, i.e. once visited it components may enter the same transaction one more time. We need to distinguish messages sent in different passes through a transaction. Let us consider the message $v_k(M_i^P)$. It will be denoted as $v_k^1(M_i^P)$ when sent for the first time. The message $v_k(M_i^P)$ sent during the l-th visit of the transaction Tr_i will be denoted as $v_k^l(M_i^P)$. When in transaction Tr_i data message is transmitted by the component P_j to the component P_k interface assigns to this message an index which shows how many times transaction Tr_i has been visited by P_j and then stores the message it in the queue of P_k. Index is necessary

because the component of destination may have visited transaction Tr_i less number of times so it will need to read the previous messages having the lesser index.

Another one important assumption is that all queues are bounded, so that no one of them can grow to infinite length. It is easily can be seen that graph shown in Figure 10a does not satisfy this condition because component P may transmit data infinitely often while Q may not to take this data from a queue staying in the initial state. If there is a data message transmitted from Q to P instead of ack (Figure 10b) then the graph will satisfy the condition as each time after sending a message to Q component P will have to wait a response in Tr_2. There exists a formal sufficient condition for the queues to be bounded [4].

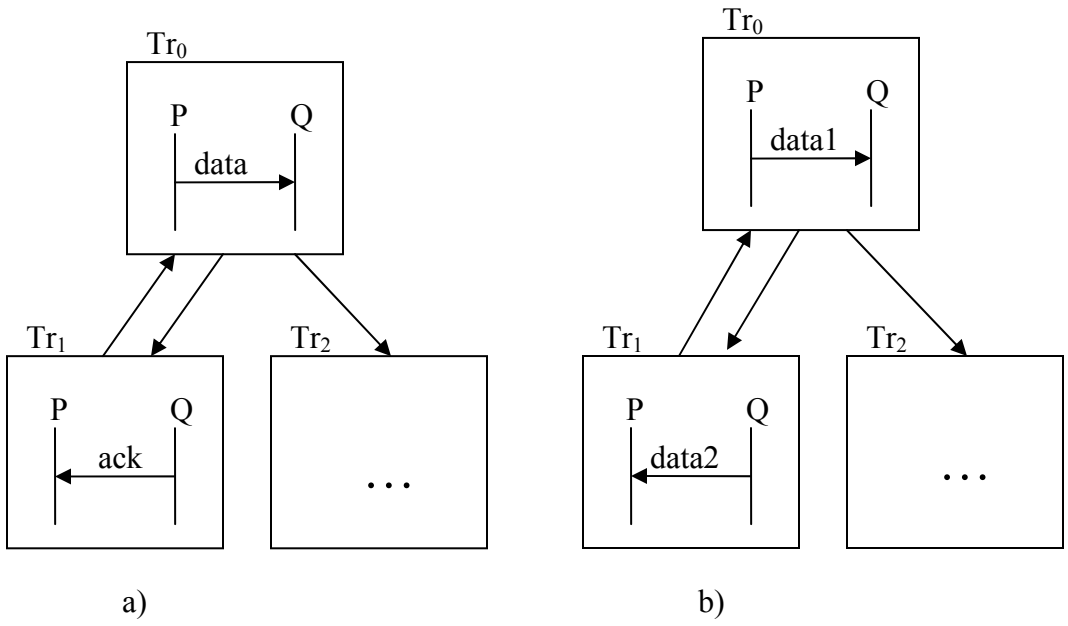

Figure 10

Let us consider the communication graph H_G of MSC-graph G: the set of vertices is the set Comp, and there is an arc from component P to component Q if P sends a data message to Q in the MSC M_i^P of some transaction Tr_i. For a set of transactions T we refer to the set of components that send or receive a data message in the MSC of some transaction in T as the set of active components in T. For every cycle in MSC-graph G consider the subgraph of communication graph H_G induced by the set of active components of the cycle. MSC-graph is called bounded if it has the property that for each cycle the above mentioned subgraph of communication graph is strongly connected. For a bounded MSC graph the queues of all components have the finite length.

Definition 4. Intermediate interface

The definition of intermediate interface T may be extended for the case of the graph of transactions as follows. Let T be the structure $(Q, q_0, q_f V_T^I, V_T^0, \tau_P)$ where

1) S is the set of states. $S = \prod_{P \in Comp} S_P \times \prod_{P \in Comp} Q_P$ where $\prod$ means the Cartesian product of the specified sets. So the state of intermediate interface includes the states of all components and the current state of their queues.

2) $s_0 = (s_0^{P_1}, ..., s_0^{P_M}, \underbrace{\lambda, ..., \lambda}_{M})$ is an initial state.

All components are at their first state, the queues are empty and no transaction has been visited.

3) $S_f = \{(s_{k_1}^{P_1}, ..., s_{k_M}^{P_M}, \lambda, ..., \lambda), k_l = |V(M_i^{P_l})|, Tr_i \in Trans_f\}$ is the set of final states. Final states are those which are formed by the last states of MSCs contained in the transactions taken from the set of the final transactions $Trans_f$ and all queues should be empty.

4) $V_T^I = \bigcup_{P \in Comp} V_P^O$;

5) $V_T^O = \bigcup_{P \in Comp} V_P^I$;

and $V_T = V_T^I \cup V_T^O$.

6) $\tau_T \subseteq S_T \times V_T \times S_T$ is the set of transitions.

$$\tau_T = \bigcup_{j=1,..,M} \{(s^{P_1}, ..., s^{P_j}, ..., s^{P_M}), q^{P_1}, ..., q^{P_M}),$$

$$v, (s_1^{P_1}, ..., s_1^{P_j}, ..., s_1^{P_M})), q_1^{P_1}, ..., q_1^{P_M})) |$$

$$v = curAct(s^{P_j}), \forall P \in Comp\ s_1^P \in nextState(s^P),$$

(1) $v \in V_{P_j}^C \cup (V_{P_j}^{OD} \setminus \bigcup_{P \in Agents(Tr_i)} V_P^{ID}), \forall R \in Comp\ q_1^R = q^R, or$

(2) $v \in V_{P_j}^{OD} \cap \bigcup_{P \in Agents(Tr_i)} V_P^{ID}$,

$q_1^{P_m} = q^{P_m}.v\ if\ P_m = dest(v), and\ q_1^{P_m} = q^{P_m}\ otherwise, or$

(3) $v \in V_{P_j}^{ID}, q^{P_j} = v.q_1^{P_j}; q^{P_l} = q_1^{P_l}, if\ l \neq j.$

}

As before, intermediate interface generates control messages whenever a component waits for it. If a component wants to transmit a data message then intermediate interface

puts it into queue of that component which it wants to send data to. The function $dest : V \to Comp$ shows the receiver of the message. For an ingoing data message intermediate interface puts it out from the queue and delivers to the component making sure that the sender transmitted this data while it had been in the transaction for the same number of times as the receiver has been. If a data message has no receiver (i.e. a component that needs the data) then the message is not put into queue (ignored).

The SystemC implementation

For an implementation of above-stated ideas we chose the SystemC, a modeling language based on C++ which supports different levels of abstractions including system level modeling. The components participating in an MSC graph execution are presented as SystemC modules that exchange messages through FIFO channels. The declaration of data and control messages differ only in the type of data transmitted through a channel. For a control messages this type is Boolean while for data messages it can vary from C++ or SystemC data types to user defined ones.

So the task was to write the intermediate interface generator. As an input it takes the global HMSC structure and the views of separate components specified in SystemC. And after processing and checking these data produces the SystemC code for an intermediate interface. The important feature is that components views are specified in SystemC, so that after generation interface can be compiled together with the components code in order to produce the binary file for the whole system. The resulting program first initializes the components and the interface, then runs the simulation and writes a simulation trace in an output file. The scheme of the interface generation is shown at Figure 11.

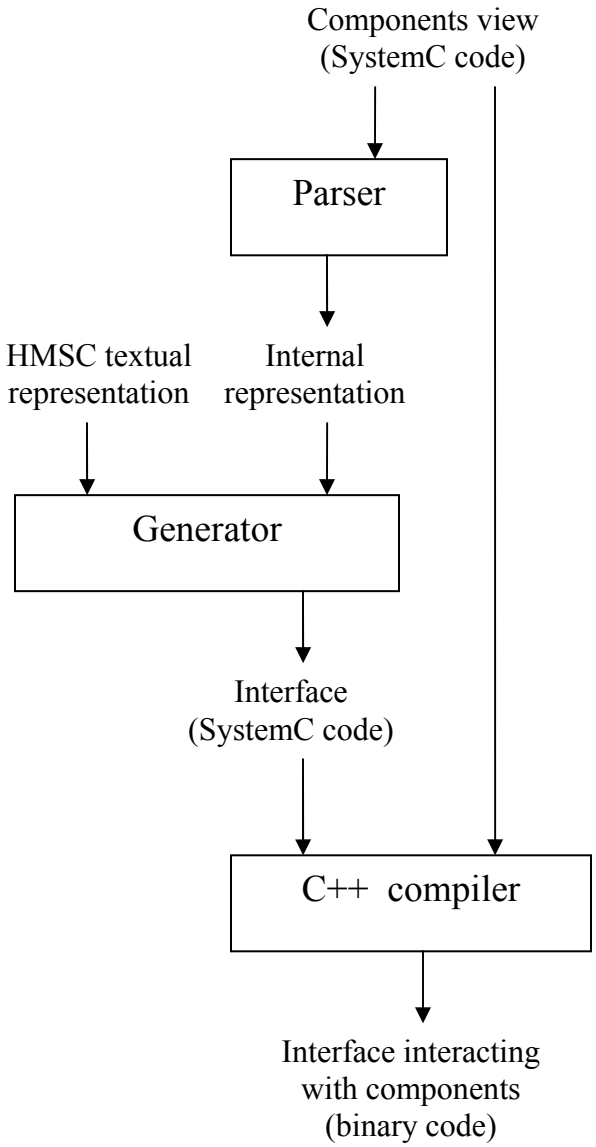

Figure 11. Interface generation

```
A => B
A => C
B => A
```

Figure 12. Textual representation of HMSC's transitions.

Interface generation

In this paragraph the interface generator will be described in more detailed way. As have been already mentioned as an input the generator requires to specify the transitions of an HMSC and the views of the components involved in it. The HMSC specification is simply a text file in a format shown in Figure 12. So, it just describes nodes and transition between them (i.e, the global control flow).

The views of the components are specified in SystemC. The generator written in C++ first parses the components code to obtain the internal representation of MSC's structure then it produces the SystemC code for an intermediate interface. Actually we are concerned only with a little subset of SystemC which is enough for specifying the modules, their ports and sending or receiving messages. The parser in unable to cope with any code which user may write in SystemC, but it is possible to specify some additional functions or classes in different C++ file and then include it into the project.

The example of a component view is listed bellow.

```
#include "StdAfx.h"
#include "Path.h"

SC_MODULE(master){   //the beginning of the master component definition

  //the ports declaration
  sc_fifo_out<bool> reqA;

  sc_fifo_in<bool> nograntB;
  sc_fifo_out<sc_int<32> > transferB;

  sc_fifo_in<bool> grantC;
  sc_fifo_out<sc_int<32> > transferC;

  //some local variables ignored by the generator
  Path* path;
  sc_int<32> data;

  SC_CTOR(master){  //components constructor

        path = new Path("Path.txt");
        data = rand();
        SC_THREAD(main_action);
  }


  //specification of an MSC for the node A
  void NodeA(){

          //sends a request
         reqA->write(true);
  }

   //specification of an MSC for the node B
   void NodeB(){

         //waits for nogrant
        bool b = nograntB->read();

         //sends data
        transferB->write(data);
   }

   //specification of an MSC for the node C
   void NodeC(){
```

```
          //waits for grant
          bool b1 = grantC->read();

          // sends data
          transferC->write(data);

    }
           ...

  void main_action(){

      char* next = path->nextNode();

      while(strcmp(next,"")!=0){

     //gets the next node name and performs the corresponding procedure
           if (strcmp(next,"A") == 0) NodeA();
           if (strcmp(next,"B") == 0) NodeB();
           if (strcmp(next,"C") == 0) NodeC();
           if (strcmp(next,"D") == 0) NodeD();
           if (strcmp(next,"E") == 0) NodeE();
           if (strcmp(next,"F") == 0) NodeF();
           if (strcmp(next,"G") == 0) NodeG();
           if (strcmp(next,"H") == 0) NodeH();

           next = path->nextNode();
      }
  }
};
```

Simulation

The interface produced by the generator satisfies the formal specification given in previous section. It exchanges messages with the components so that to provide effective data transfer. All ingoing data messages (that are outgoing for the component) are put into the queue and then forwarded to the destination component. The only exception is when some component sends data message which has no receiver. It may occur when the MSC of this component contains the outgoing data message and no other MSCs in the node contain the ingoing data message with the same name. Then such data messages are simply lost. The situation when a component expects to receive data message while no other component sends it is not allowed and the generator will produce an error.

The control messages can be generated in advance thus avoiding deadlocks in control flow. Actually, interface does not perform name matching for the control messages as it does not care whether there is some component that produces this control message or not. When the component reaches the state when it waits for a certain control message interface generates it by itself.

In general case, if MSC graph contains cycles then some components may go ahead for an unbounded number of steps regarding the other components. As interface temporarily stores all data messages in queues, they may be overflowed. To avoid this situation interface provides the check that no MSC graph node is entered again by the component until all other components finished its execution. Thus, two copies of the same node can not coexist at the same moment. The component which is ready to enter the node once more will be blocked until other components finish the previous execution of that node.

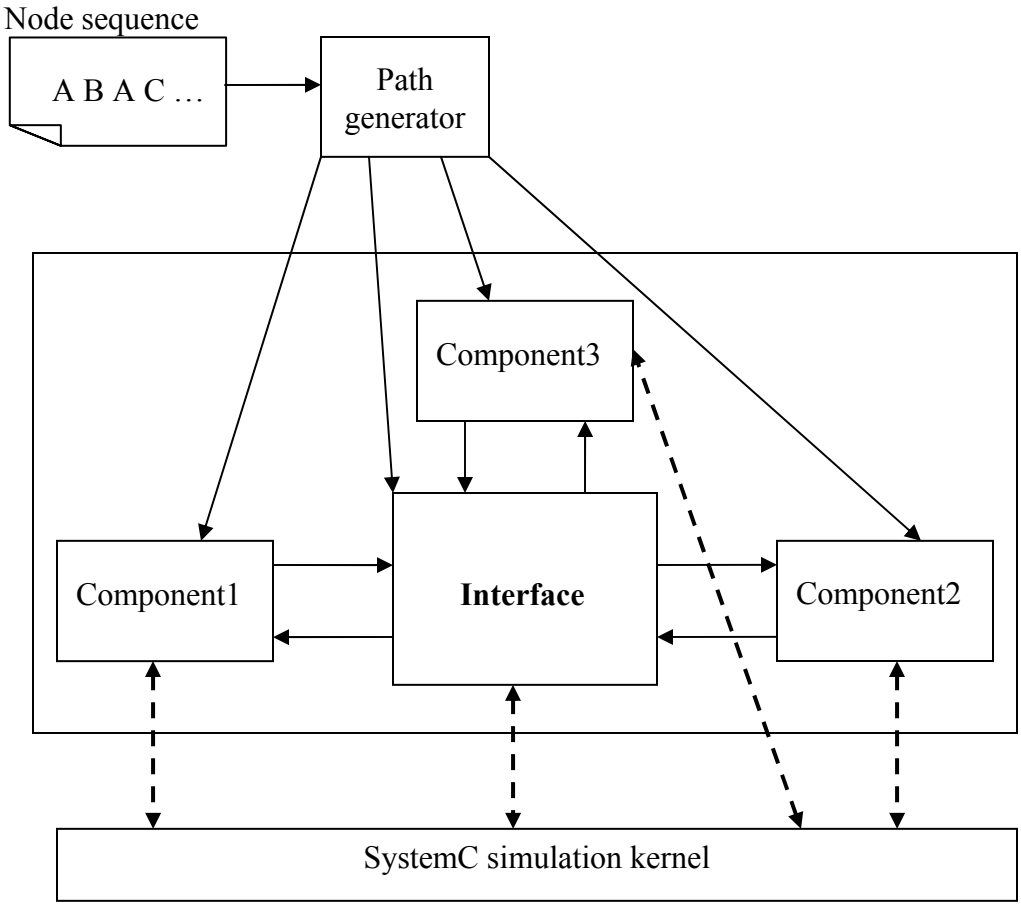

Figure 13. Simulation scheme

Another challenge is handling branches in MSC graph in such a way as to make sure that all components make the same choice of the next node. In order to cope with this problem the decision was made to implement a routine which would generate a graph path. Both interface and components address this function to obtain the next node for the execution. At present moment the path generator is implemented in a very simple way, it just reads the sequence of nodes from a text file. Interface makes sure that the node sequence actually represents a graph path and expects that components will perform actions specified the current node. If it is not so, then interface produces an error.

The scheme shown in Figure13 depicts more precisely the interaction of components during the simulation.

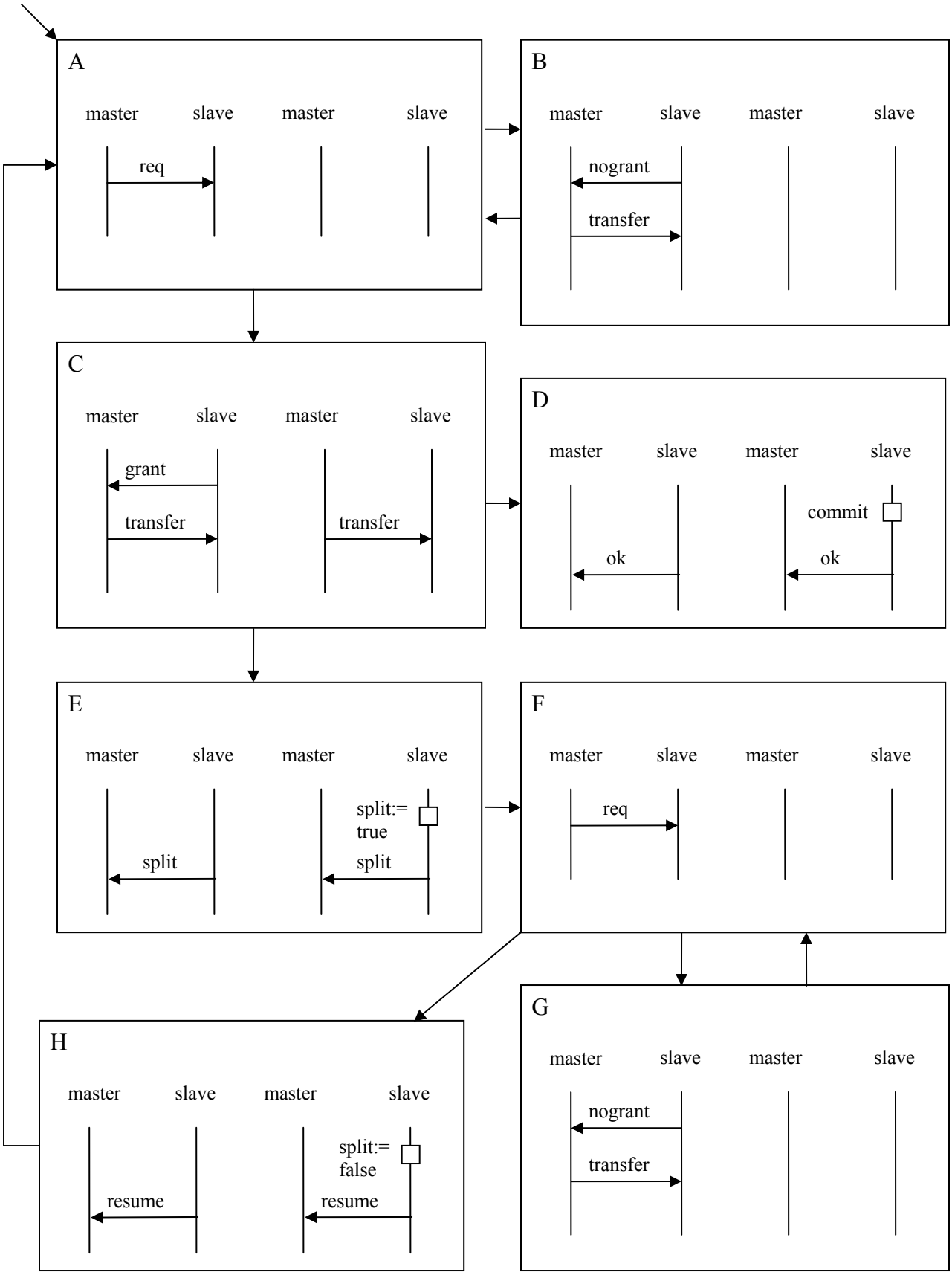

Figure 14. Bus controller example. Two components: master and slave.

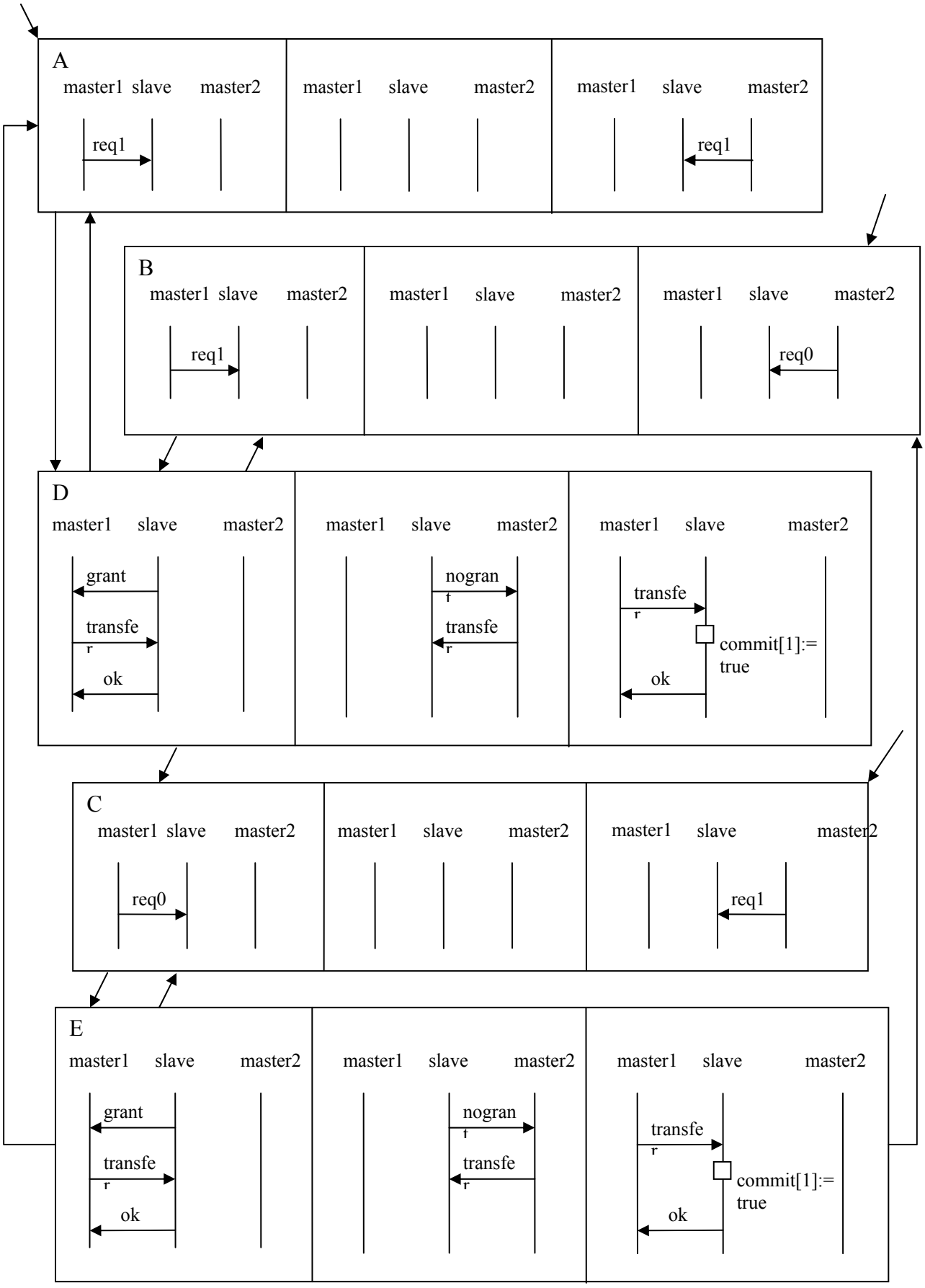

Figure 15. Bus controller example. Three components: two masters and slave.

Results

Two examples were passed through the interface generator. One of them is shown at Figure 14. It represents the bus controller specification where each component has its own view on the message sequence transmitted between it and the interface which in this case is called a bus controller. For example, slave component does not know about the request sent by master to the interface but it will not receive the data until master component send *request* and then the data message *transfer*.

Another example is shown at Figure 15. It contains three components: two masters and one slave; master1 has higher priority then master2. So each time master1 sends *req1* (nodes A and B) controller produces grant message to it and then forwards data to slave component while master2 receives *nogrant* and the data it transfers are lost (node D). Master2 can successfully transmit data (node E) only if master1 has sent a *req0* message (node C). Execution may start at any of three nodes A, B or C.

The node sequence and the trace produced as a result of simulation is listed bellow.

Master and slave

```
Path: A B A C E F G F H A B A C D

Control message reqA received from master
Control message nograntB sent to master
Data message transferB = 41 received from
master
Control message reqA received from master
Control message grantC sent to master
Data message transferC = 41 received from
master
Control message splitE sent to master
Data message transferC = 41 sent to slave
Control message reqF received from master
Control message nograntG sent to master
Control message splitE received from slave
Control message resumeH received from slave
Data message transferG = 35 received from
master
Control message reqF received from master
Control message resumeH sent to master
Control message reqA received from master
Control message nograntB sent to master
Data message transferB = 35 received from
master
Control message reqA received from master
Control message grantC sent to master
Data message transferC = 35 received from
master
Control message okD sent to master
```

Two masters and slave

```
Path: B D A D C E A D

Control message req1B received from master1
Control message grantD sent to master1
Control message req0B received from master2
Control message nograntD sent to master2
Data message transferDm1 = 41 received from
master1
Control message okD sent to master1
Data message transferDm1 = 41 sent to slave
Data message transferDm2 = 35 received from
master2
Control message req1A received from master2
Control message req1A received from master1
Control message okD received from slave
Control message grantD sent to master1
Control message nograntD sent to master2
Data message transferDm1 = -66 received
from master1
Control message okD sent to master1
Data message transferDm1 = -66 sent to
slave
Data message transferDm2 = 35 received from
master2
Control message req1C received from master2
Control message grantE sent to master2
Control message req0C received from master1
Control message nograntE sent to master1
Data message transferEm2 = 35 received from
```

```
Data message transferC = 35 sent to slave
Control message okD received from slave
Interface reached the last state
```

```
master2
Control message okE sent to master2
Control message okD received from slave
Data message transferEm2 = 35 sent to slave
Control message req1A received from master1
Control message grantD sent to master1
Control message req1A received from master2
Control message nograntD sent to master2
Control message okE received from slave
Data message transferDm1 = -124 received from master1
Control message okD sent to master1
Data message transferDm1 = -124 sent to slave
Data message transferDm2 = -31 received from master2
Control message okD received from slave
Interface reached the last state
```

References

1. Luca de Alfaro and Thomas A. Henzinger. Interface automata. *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering* (FSE), ACM Press, 2001, pp. 109-120.
2. R. Passerone, J. A. Rowson, A. L. Sangiovanni-Vincentelly. Automatic Synthesis of Interfaces between Incompatible Protocols, in *DAC* (San Francisco, CA), June 1998.
3. Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. *Proceedings of the International Conference on Computer-Aided Design* (ICCAD), IEEE Computer Society Press, 2002, pp. 132-139.
4. R. Alur, M. Yannakakis. Model Checking of Message Sequence Charts. *Proceedings of the 10th International Conference on Concurrency Theory*, 1999.