The algorithm starts with type definitions (1) which lead to the types STS and ActivityDiagram. In block (2), an empty STS is created and an activity diagram is read as input. The algorithm will eventually add nodes and edges to the empty STS thereby creating a different representation of the input activity diagram. The first step is to create a start node in the STS. This start node is processed by calling the recursive procedure *process* (3). If a node is processed, the procedure first checks what states are reachable in the activity diagram from the node it processes by calling the procedure *successor*. Dependent on the outcome of this procedure, the algorithm chooses one out of three options (3A, 3B or 3C)

If the successor function returns an and-value or a join-value (block 3A and 3B), the algorithm creates one new node that is to be added to the STS. In case of an or-value, the algorithm creates several nodes that are potentially added to the STS. Each node represents a state of the system represented by the activity diagram. If a new node is created, the function addAndProcess is called (4). This function first checks if the new node already exists and, if so, if there is exists a transition between the node it is currently processing and the newly created node.

If both the node and the transition exist, the algorithm does nothing with the new node and does not call itself recursively (4A). If the new node already exists, but there is no transition between the currently processed node and the new node, this transition is added. As in the previous case, the algorithm does not call itself (4B). If both the transition and the node do not exist in the STS, they are both added and the algorithm continues by processing the new node (4C)

The result if algorithm 1 is a non-deterministic STS. Algorithm 2 can be used to transform a non-deterministic STS into a deterministic STS. Algorithm 2 uses the following functions in addition to the functions used in algorithm 1:

- ***getOutgoingLabels(GR,N)*** returns the set of labels labeling outgoing transitions of node *N* in graph *GR*
- ***getDestWithLabel(GR,N,L)*** returns a set of nodes of graph *GR* that is reached by taking a transition from node *N* with an edge labeled *L*. Note that in a deterministic graph the set of nodes always contains one node.
- ***createOrNode(S)*** creates a node containing an or-statement and a set of nodes. This new node represents a "merge" of multiple nodes used to make the STS deterministic. A valid value for the node could be: OR({CD},{D}).
- ***hasNode(GR,N)*** returns True if graph *GR* has node *N*. Otherwise returns False
- ***copyEdgesWithSource(GR,S,N)*** adds an edge to graph *GR* with source *N* and destination *D* for every edge of *GR* with source *S* and destination *D*
- ***removeEdge(GR,S,D,L)*** removes the edge with source *S*, destination *D* and label *L* from graph *GR*
- ***sourceOf(GR,S)*** returns those edges of graph *GR* that have a source *S*
- ***destOf(GR,D)*** returns those edges of graph *GR* that have a destination *D*
- ***removeNode(GR,N)*** removes node *N* from graph *GR*
- ***removeAllEdgesOfNode(GR,N)*** removes all edges in graph *GR* that have *N* either as a source or a destination

In algorithm 2, a number of blocks can be identified. These blocks are numbered an treated after the algorithm is given.

Algorithm 2

```
GRN :: {activity}+ | or({{activity}+})
GRE :: (source :: GRN,dest :: GRN,label::string)
STS :: ({GRN},{GRE})
DONE :: True | False

graph = read(STS)

DONE = False

WHILE NOT DONE DO
    DONE = true
    FOR EACH n ∈ graph.GRN DO
        labels = getLabels(n)
        FOR EACH l ∈ labels DO
            dest = destWithLabel(n,l)
            IF #(dest) = 1 THEN
            ELSE
                tempnode = createOrNode(dest)
                IF hasNode(graph,tempnode) THEN
                ELSE
                    addNode(graph,tempnode)
                END IF
                addEgde(graph,n,tempnode,l)
                FOR EACH d ∈ dest DO
                    copyEdgesWithSource(graph,d,tempnode)
                    removeEdge(graph,n,d,l)
                END FOR
                DONE = False
            END IF
        END FOR
    END FOR
END WHILE

DONE = False
WHILE NOT DONE DO
    DONE = True
    FOR EACH n ∈ graph.GRN DO
        IF (#(sourceOf(graph,n)) = 0 AND n != End) OR (#(destOf(graph,n))=0 AND n != Start) THEN
            removeNode(graph,n)
            removeAllEgdes(graph,n)
            DONE = False
        ELSE
        END IF
```

UITLEG BLOKKEN

Once both original Activity Diagrams have been converted to two deterministic STS diagrams, the intersection can be calculated. This is done by "walking" through the two deterministic graphs and creating a new graph representing the paths that can be taken in both input STS diagrams.

In addition to the functions used in algorithm 1 and algorithm 2, algorithm 3 uses the following functions:

- ***createNode(S)*** creates a node containing *S*, where S represents a set of nodes from an activity diagram (or GRN)
- ***getNode(GR,S)*** returns the node of graph *GR* containing set *S*

Algorithm 3 is presented below. The blocks identified in the algorithm are treated after the algorithm is given.

Algorithm 3

```
GRN :: {activity}+ | or({{activity}+})
GRE :: (source :: GRN,dest :: GRN,label::string)
STS :: ({GRN},{GRE})
DONE :: True | False

graphA = read(STS)
graphB = read(STS)
graphC = new(STS)

tempnode = createNode(Start)
addnode(graphC, tempNode)
currA = destWithLabel(graphA,Start,'')
currB = destWithLabel(graphB,Start,'')
currC = getNode(graphC, Start)

intersect(currA, currB, currC, '')


FUNCTION intersect (A :: GRN, B :: GRN, C :: GRN, l :: Label)
outA = getOutgoingLabels(graphA, A)
outB = getOutgoingLabels(graphB, B)
outC = outA ∩ B
IF out C = ∅ THEN (* do nothing *)
ELSE
    tempnode = createNode(outC)
    IF hasNode(graphC, tempnode) THEN
        addEdge(graphC, C, tempnode, l)
    ELSE
        addNode( graphC, tempnode)
        addEdge(graphC, C, tempnode, l)
        FOR EACH o ∈ outC DO
            currA = destWithLabel(graphA,currA,o)
            currB = destWithLabel(graphB,currB,o)
            currC = getNode(graphC,outC)
            intersect( currA, currB, currC, o)
        END FOR
    END IF
```

UITLEG BLOKKEN

The result of algorithm 3 is a deterministic STS that represents the match of the original two activity diagrams. If the resulting STS contains an End-node, the match is successful. The final step is to convert the newly created STS into an activity diagram. This can be done in a straightforward way by creating an activity for every node of the STS that represents one activity node (e.g. A and B in the next example) and creating multiple activities and a choice node for every node in the STS that represent multiple activities (e.g. CD).

It is also possible to create an algorithm that detects if there are dependencies between each pair of activities and reintroduce parallelism if possible. Creating such an algorithm is not part of this thesis.

4.4.4 Example

In the following example, two activity diagrams, A and B are being compared. This comparison will eventually lead to a third activity diagram C that represents the matching of A and B. Both diagrams are shown in figure 4.8 and contain the same activities (or nodes), but have a different choreography. All activities in both A and B are atomic. The first step is to translate the activity diagrams into STS, according to algorithm 1. The result of the transformation is shown as STS A and B in figure 4.9. Removing the wait states results in STS A' and STS B'.

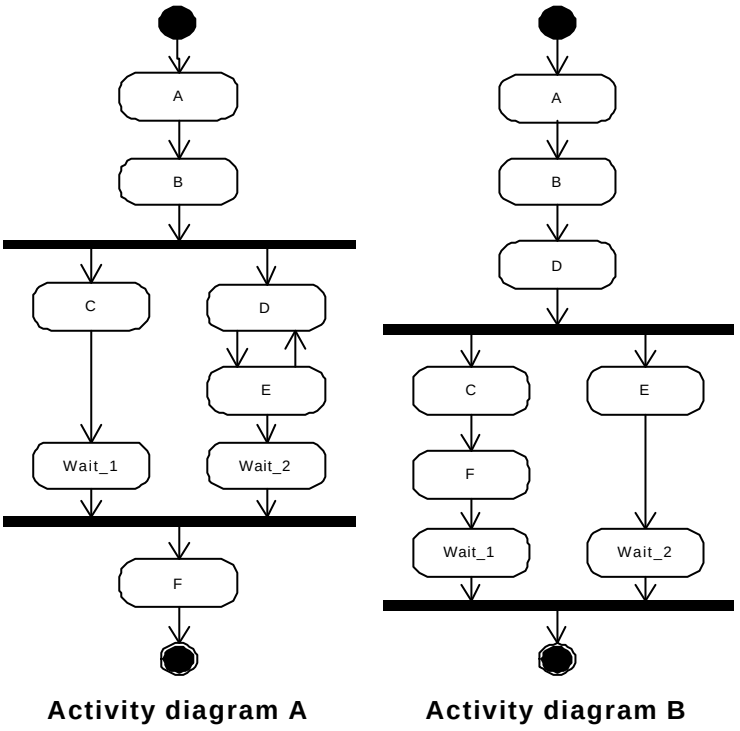

Figure 4.8: Example activity diagram

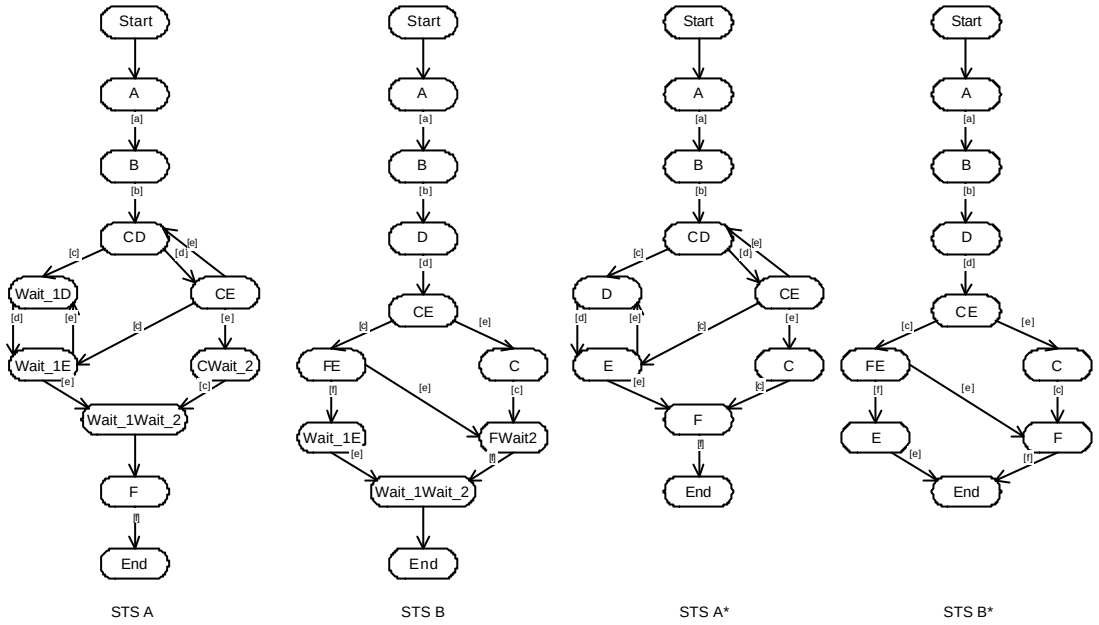

Figure 4.9: Non-deterministic STS

The newly created STS diagrams A' and B' can be non-deterministic. In fact, STS A' is non-deterministic (e.g. node CE has two outgoing transitions labelled e). Using algorithm 2, the STS diagrams are transformed into STS A'' and STS B'', shown in figure 4.10 that are both deterministic.

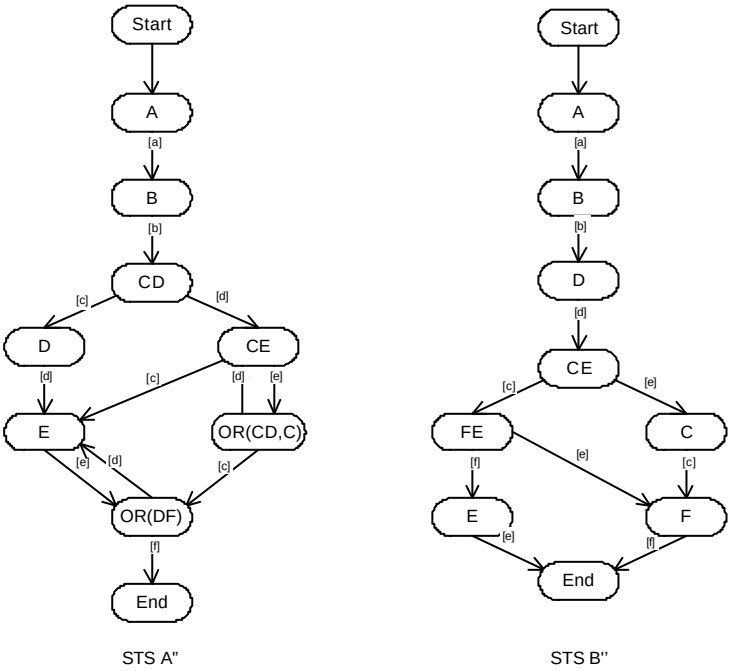

Figure 4.10: Deterministic STS

Using STS A'' and STS B'' as input, algorithm 3 creates the intersection of these diagrams, represented as STS C in figure 4.11. STS C can be transformed back into an activity diagram. Using the "straightforward" method, the result is activity diagram C. If a better algorithm is used that can detect parallelism, STS C can be transformed into activity diagram C'

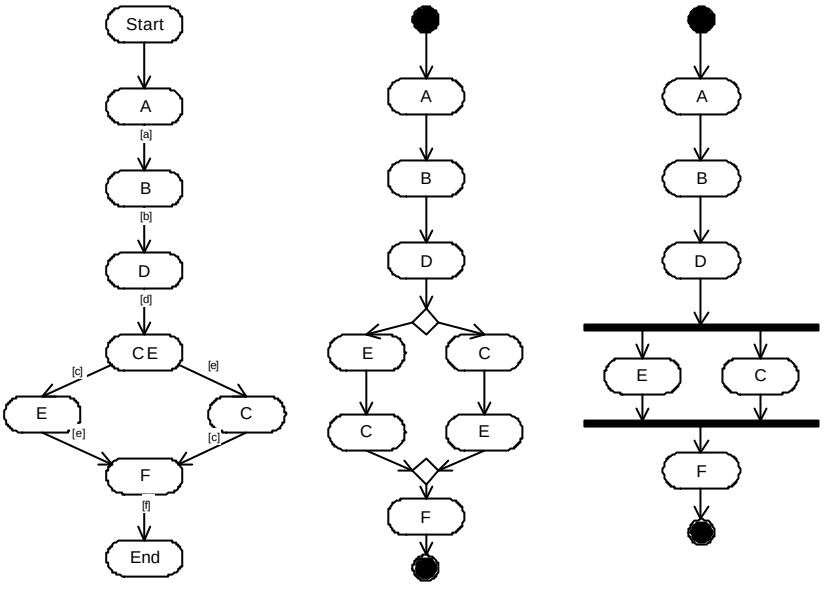

Figure 4.11: Result of matching

4.4.5 Transformation to ebXML

In ebXML, binary collaborations expressed in UML activity diagrams. The algorithm designed for matching activity diagrams can therefore be used to match binary collaboration. The solution described in this chapter can be used to match to structure of collaborations, but in order to come to an useful match, more has to be taken into consideration.

In ebXML, activities in a Binary collaboration are either nested collaborations or Business Transaction Activities. Neither of these two are atomic, so in order to use the algorithm, the following conversions have to be done:

- For each activity A that contains a nested collaboration, the nesting must be removed. This is done by connection the all incoming transitions of A to the first normal node in the nested diagram. Each transition in the nested diagram that leads to an end state should be connected to all outgoing transitions of A.
- If a Business Transaction Activity (BTA) consists of the exchange of more than one document, that BTA must be replace by a sequence of BTA's, each consisting of the exchange of exactly one document. If the replace BTA contains pre conditions, these pre conditions are added to the first BTA in the sequence. If the BTA contains post conditions, these post conditions are added to the last BTA in the sequence.

Besides the matching of structure, also content has to be matched. The following chapter treats the matching of content in ebXML.