On the semantics of EPCs: Faster calculation for EPCs with small state spaces

Nicolas Cuntz
Computer Graphics and Multimedia Systems Group, University of Siegen, Germany
nicolas.cuntz@uni-siegen.de

Jörn Freiheit
Max-Planck-Institute Saarbrücken, Germany
freiheit@mpi-sb.mpg.de

Ekkart Kindler
Computer Science Department, University of Paderborn, Germany
kindler@upb.de

Abstract: One of the main features of *Event driven Process Chains* (*EPCs*) is the non-local semantics of the OR-join and the XOR-join connectors. Simulating this non-local semantics faithfully and efficiently is still a challenging problem. A year ago, we have shown that the semantics of moderately sized EPCs can be calculated in reasonable time by using techniques from symbolic model checking. For larger EPCs, however, this method still takes too long for practical use.

In this paper, we introduce and discuss a new technique for calculating the semantics of EPCs: We combine an explicitly forward construction of the transition system with a backward marking algorithm for checking the non-local constraints. Though this method does not always provide a result, it works for most practical examples and, in most cases, it is much faster than the symbolic algorithm. Basically, the computation time is linear in the size of the resulting transition system. The algorithm works for large EPCs as long as the resulting transition systems are small (where transition systems with millions of states and transitions are still considered to be small), which is true for many practical EPCs.

1 Introduction

Event driven Process Chains (*EPCs*) have been introduced in the early 90ties for modelling business processes [KNS92]. For easing the modelling of business processes with EPCs, the informal semantics proposed for the OR-join and the XOR-join connectors are *non-local*. This non-locality results in severe problems when it comes to a formalisation of the semantics of EPCs [LSW98, Rit00, vdADK02]. These problems, however, have been resolved by defining a semantics for an EPC that consists of a pair of two correlated transition relations by using fixed-point theory [Kin04]. Moreover, these transition systems can be calculated by using techniques from symbolic model checking [CK04b, CK05], which is implemented in an open source tool called EPC Tools [CK04a].

For moderately sized EPCs, this tool calculates the semantics in a reasonable time and, therefore, allows us to simulate moderately sized EPCs. For larger EPCs, however, this technique does not work anymore, even if the set of reachable states of the EPC is small. Therefore, we come up with another algorithm for calculating the semantics of EPCs that is tuned to large EPCs with small state spaces. Unfortunately, this algorithm does not provide a result for all EPCs. For some EPCs, the algorithm might fail. But, it works well for most practical examples. And the computation speed is limited only by the available main memory. Our prototype implementation in Java can calculate transition systems with one million states and eight million transitions in 45 seconds on a Linux machine with 1GB main memory and a 2.4 GHz processor. In combination with another optimisation called chain-elimination, which we introduced for improving the symbolic algorithm already, we can simulate EPCs with more than a billion reachable states.

In this paper, we briefly rephrase the syntax and semantics of EPCs (Sect. 2) before presenting the new algorithm for constructing the transition system of an EPC (Sect. 3). Moreover, we will discuss the efficiency of the algorithm and some improvements (Sect. 4). Finally, we will discuss some practical experiences with the new algorithm (Sect. 5).

2 Syntax and Semantics of EPCs

In this section, we informally introduce the syntax and the semantics of EPCs as formalised in [Kin04], which is a formalisation of the informal ideas as presented in [KNS92, NR02].

2.1 Syntax

Figure 1 shows an example of an EPC. It consists of three kinds of *nodes*: *events*, which are graphically represented as hexagons, *functions*, which are represented as rounded boxes, and *connectors*, which are represented as circles. The dashed arcs between the different nodes represent the *control flow*. The two black circles do not belong to the EPC itself; they represent the current *state* of the EPC: A state, basically, assigns a number of *process folders* to each arc of the EPC. Each black circle represents such a process folder at the corresponding arc.

Mathematically, the nodes are represented by three pairwise disjoint sets E, F, and C, which represent the events, functions, and connectors, respectively. We denote the set of all nodes by $N = E \cup F \cup C$. The type of each connector is defined by a mapping $l : C \to \{and, or, xor\}$. The control flow arcs are a subset $A \subseteq N \times N$. In addition, there are some syntactical restrictions on EPCs, which are not discussed here since they are not relevant for our semantical considerations.

A *state* of an EPC assigns zero or one *process folders* to each arc of the EPC. So a state σ can be formalised as a mapping $\sigma : A \to \{0, 1\}$, which is the characteristic function of the set of all arcs that have a process folder in this state. The set of all states of an EPC will be denoted by Σ.

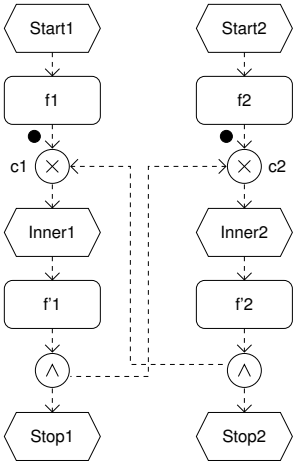

Figure 1: An EPC

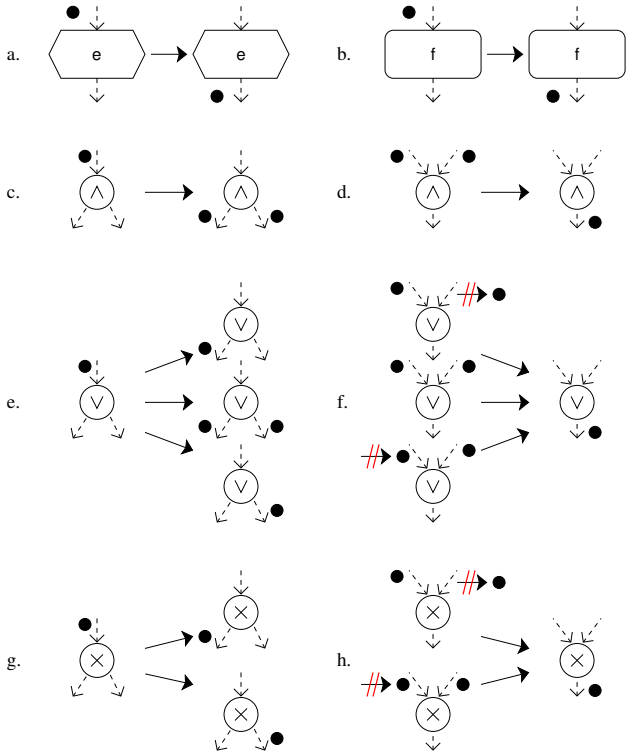

Figure 2: The transition relations for the different nodes

2.2 Semantics

The semantics of an EPC defines how process folders are propagated through the EPC. This can be formalised by a *transition relation* $R \subseteq \Sigma \times N \times \Sigma$, where the first component denotes the *source state*, the last component denotes the *target state*, and the middle component denotes the EPC node that propagates the folders.

For events and functions, the transition relation is very simple: a process folder is propagated from the ingoing arc to the outgoing arc as shown in Fig. 2 a. and b. The semantics of the other nodes is shown in Fig. 2, too. For lack of space, we discuss the details only for the XOR-join connector (case h.): An XOR-join connector waits for a folder on one ingoing arc, which is then propagated to the outgoing arc. But, there is one additional condition: The XOR-join must not propagate the folder if there *is* or there *could arrive* a folder on the other ingoing arc. This additional condition is graphically indicated by the label #•• at the other arc. Note that this condition cannot be checked locally in the current state: whether a folder can arrive on the other arc depends on the overall behaviour of the EPC. Therefore, we call the semantics of the XOR-join connector *non-local*. The

other node with a non-local semantics is the OR-join connector. Its semantics is shown in Fig. 2 f. Again, the label $\not\blacktriangleright\bullet$ at an arc indicates the condition that no folder can arrive at that particular arc anymore. The only difference to the XOR-join connector is that an OR-join may fire also when there are folders on both input-arcs.

Note that, in this informal definition of the *transition relation*, we refer to the transition relation itself when we require that no folders can arrive at some arc according to the transition relation. Therefore, we cannot immediately translate this informal definition into a mathematically sound definition. In order to resolve this problem, we assume that some transition relation P is given already, and whenever we refer to the non-local condition, we refer to this transition relation P. Thus, Fig. 2 defines a mapping $R(P)$: for some given transition relation P, it defines the transition relation $R(P)$. Then, the actual semantics of an EPC could be a fixed-point of R, i. e. a transition relation P with $R(P) = P$. Unfortunately, there are EPCs with many different such fixed-points (Fig. 1 shows an example) and there are EPCs that do not have such a fixed-point at all (Fig. 3 shows an example).

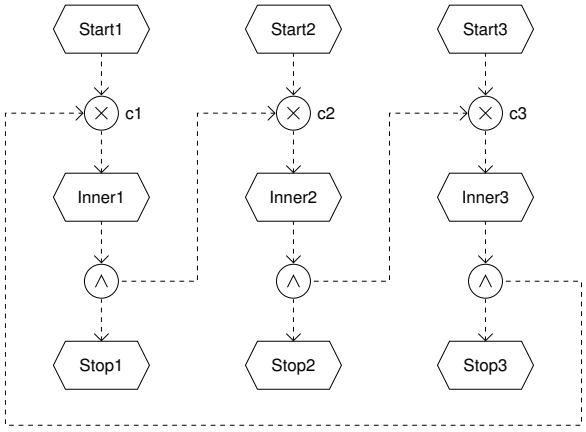

Figure 3: The vicious circle

So, we had to come up with another idea (see [Kin04] for details): The most important property of $R(P)$ is that it is monotonously decreasing in P. This property guarantees that there exists a least transition relation P and a greatest transition relation Q such that $R(Q) = P$ and $R(P) = Q$, where P is called the *pessimistic transition relation* and Q is called the *optimistic transition relation* of the EPC. This pair of transition relations (P, Q) is defined as the semantics of the EPC. In most cases, we have $P = Q$, which means that P is a fixed-point of R. If P and Q are different, there are some ambiguities in the interpretation of the EPC. Therefore, we call an EPC *unclean* if P and Q are different, and we call it *clean* if P and Q are equal.

3 Explicit calculation of the transition system

We have shown that the pair (P, Q) can be calculated by fixed-point-iteration using techniques from symbolic model checking [CK04b, CK05]. Here, we will introduce another algorithm that calculates a transition system P with $R(P) = P$ in an explicit way. In some cases, however, the algorithm might fail.

3.1 Basic idea

The basic idea of the algorithm is very simple: The transition system will be constructed starting from the initial state of the EPC. From this state, we construct new transitions and new states according to the semantics of the different kinds of nodes of the EPC as illustrated in Fig. 2. In the first phase, however, we completely ignore all non-local connectors, i. e. we ignore the OR-join and the XOR-join connectors. In this *local forward construction phase*, we construct all states and transitions reachable by transitions for local nodes only. This algorithm can be implemented in a fairly standard way as known from other modelling notations such as Petri nets.

When no new transitions and states can be constructed in the local forward phase anymore, we start the second phase. In this second phase, we identify those states in which additional folders can be propagated to the input-arcs of the non-local join connectors. To this end, we start a backward marking algorithm for each non-local join connector. Therefore, we call this phase the *backward marking phase*. For each non-local join connector, this phase works as follows: First, we identify those transitions in the already constructed transition system that propagate an additional folder to one of the input-arcs of the non-local join connectors. If a transition adds a folder, we mark the source state of that transition. Then, we systematically mark all the predecessors of these states by going backward through the transition system. When no new states can be marked in this backward marking phase, the second phase stops. Again, the algorithm for the backward marking can be implemented in a fairly standard way.

In the third phase, we investigate all the states of the transition system again and check whether a non-local connector can be fired in one of its states. Since we have marked all states at which additional folders can arrive at the input-arcs of a join connector, it is now easy to decide 'locally' in a state whether a non-local connector can fire or not: If the state is marked by a non-local connector, the non-local connector cannot fire. After adding all transitions (and possibly new states) for the non-local connectors, we begin a new *round* of the algorithm starting with the *local forward construction phase*.

When a round ends without adding new transitions during the third phase, the algorithm terminates. Since the algorithm only adds states and transitions and since there are only finitely many states, we know that this algorithm will eventually terminate.

The question, however, is whether the constructed transition system meets the requirement $R(P) = P$. The answer to this question is quite simple: During the backward marking phase for each non-local connector n, we check for each newly marked state whether

there is a transition for connector n leaving this state already. If this happens during the construction, we know that this transition is wrong, because its non-local condition is violated. As soon as this happens, we know that the constructed transition system violates $R(P) = P$ (we will see an example for such a situation below). If this does never happen, we know that the resulting transition system P meets the condition $R(P) = P$.

3.2 Example

We illustrate the algorithm by constructing the transition system for the simple technical example EPC shown in Fig. 4. We omitted functions from this EPC in order to make the resulting transition system smaller and better understandable. In order to refer to the arcs of the EPC during the construction process, we have named them, and we have also named all nodes including the XOR-join connectors.

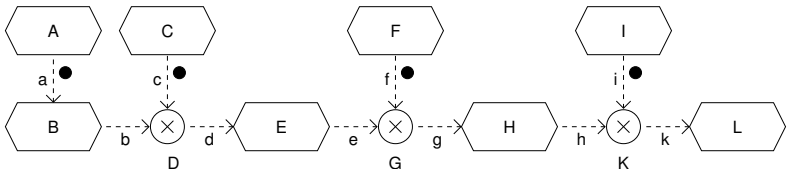

Figure 4: An technical example

We will start the first phase from the initial state of the EPC, which is denoted by $\{a, c, f, i\}$, which is the set of arcs with a folder. Initially, the only local node that can be fired is event B which results in state $\{b, c, f, i\}$. Adding this transition results in the transition system shown in Fig. 5. Note that we cannot add a further transition for a local node to this transition system. So the result of the local forward construction phase is the transition system shown in Fig. 5.

Now, we start the backward marking phase. Clearly, firing event B adds an additional folder to the input-arcs of the XOR-join connector D (viz. the folder at arc b). Therefore, we mark the source state of this transition with D. In order to emphasise its meaning (i. e. connector D should not fire in that state), we represent this mark by a crossed D at the initial node as shown in Fig. 6. Since this node does not have predecessors, the backward marking phase terminates right away – with the result shown in Fig. 6.

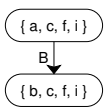

Figure 5: Local forward construction

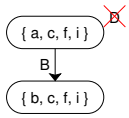

Figure 6: Backward marking

Now, we enter the third phase where we add all possible transitions for the non-local connectors (viz. the XOR-join connectors D, G, and K). The transitions for both connectors G and K can be added to both states as shown in Fig. 7. For the XOR-join connector D, however, we cannot add a transition: The initial state is marked by a D, so we do not add a transition for D in that state; in the second state $\{b, c, f, i\}$, we do not add a transition for connector D because there are folders on both input-arcs, which implies that the XOR-join connector is not enabled in this state. This finishes the third phase of the first round of the algorithm.

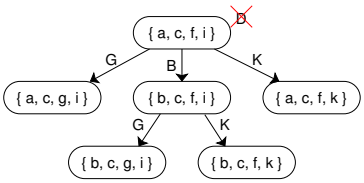

Figure 7: Adding non-local connectors

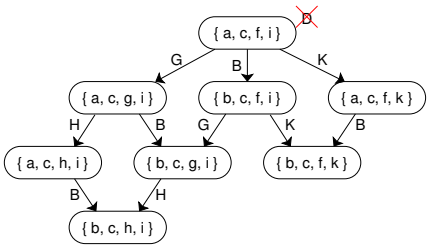

Figure 8: Local forward construction again

Since new transitions have been added during the third phase of the first round, we start another round of the algorithm. We start with the forward construction phase for all local nodes. This gives us the transition system shown in Fig. 8.

Next, we check which transitions propagate an additional folder to the input-arcs of one of the XOR-join connectors. Firing B adds another folder to the input-arcs of XOR-join connector D. So, we mark the source states of the corresponding transitions with a crossed D. Likewise, firing H adds another folder to the input-arcs of XOR-join connector K. So, we mark all the source states of H with a crossed K. The result is shown in Fig. 9. But, the second phase is not finished yet. For all marked states, we must also mark all predecessor states accordingly. The result of this backward marking phase is shown in Fig. 10.

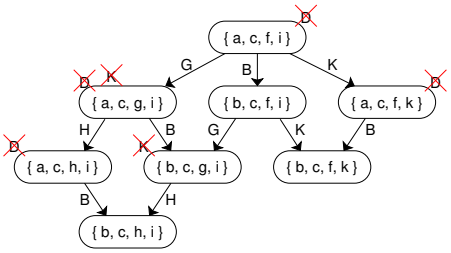

Figure 9: Initialise backward marking

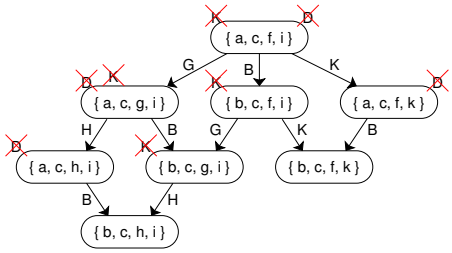

Figure 10: Backward marking

We can see that we have now marked the initial state with a crossed K and we have

constructed a transition K from that state earlier. This shows us that the resulting transition system will not meet the condition $R(P) = P$ and we better had not added transition K to the initial state. This, however, does not bother us right now, we just continue the construction.

We continue our algorithm with the third phase of round two. We add transitions for all enabled non-local connectors. Note that we cannot add transitions for XOR-join connectors D and K since the states are either marked with a crossed D or K, or there are folders on both input-arcs, or there are no folders on the input-arcs at all. We can add a transition only for XOR-join connector G at state $\{b, c, f, k\}$. The result of the second round is shown in Fig. 11.

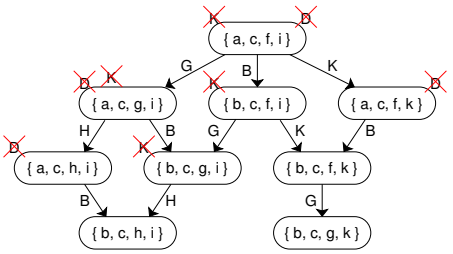

Figure 11: Adding non-local connector

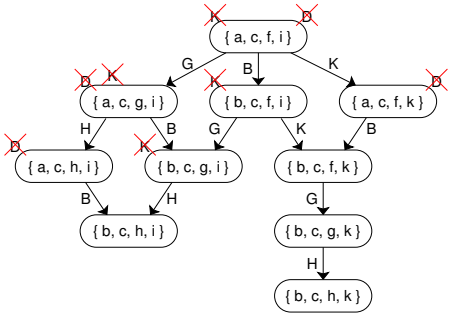

Figure 12: Local forward construction

Since we have added a new transition during the third phase of the second round, we must start another round of our algorithm. In the first phase, we can add only one transition and one state as shown in Fig. 12. Note, that according to the semantics of EPCs, an end event can never fire. Therefore, no transition can be added to the state $\{b, c, h, k\}$.

So, we can proceed with the second phase: We check which transitions add another folder to some XOR-join connector. But, there is none. Note that the new transition adds a folder to the input-arc of XOR-join connector K, but this is the first folder, not an additional folder. So, we do not mark this state.

Next, we check whether we can add another transition for a non-local connector to this transition system. But, this is no longer possible. Note that, in state $\{b, c, h, k\}$, connector K cannot fire due to the folder on its output-arc, which is called a *contact situation*. So, this was the third and last round of our algorithm and the result is shown in Fig. 12. As discussed above, we know that this result is not correct since the transition system has a transition for the XOR-join connector K starting in state $\{a, c, f, i\}$, but this state is also marked with a crossed K now. We will see later that we can improve the algorithm so that this problem is avoided for this EPC.

3.3 Some details

Before discussing this improvement of the algorithm, let us consider some details of the algorithm.

First, let us go into the details of the backward marking algorithm in the second phase. This backward marking algorithm can be implemented in a very standard way. We must make sure only that the data structure for representing the transition system under construction allows us to follow arcs backward. There is only one question left: when or where can we stop the backward algorithm? In our example, we stopped at the initial state. In examples with loops, we would stop as soon as no new states can be marked anymore, which can be implemented in a fairly standard way. However, there is one additional condition when the backward marking algorithm should stop: When the backward marking algorithm for a non-local connector n reaches a transition for connector n, the marking algorithm does not proceed beyond this transition. The reason is that this transition removes all folders from the input-arcs of the XOR-join connector n and, therefore, the preceding state needs not to be marked. This condition can be easily included to the standard marking algorithms since the transitions have labels that refer to the corresponding node of the EPC.

Second, we have seen in our example already, that it is not necessary to start the backward marking in the second phase completely from scratch. We can just keep the markings from the earlier rounds and start from that point. This way, the backward marking algorithm will visit each state of the final transition system at most once. Moreover, it is not necessary to run through the complete set of transitions in order to find those transitions that add another folder to the input-arcs of a non-local connector. We can check this during the creation of a new transition (for each non-local connector) and then add the source state of this transition to a list which will start the backward marking algorithm. Since such a list is needed in the backward marking algorithm anyway, this is no extra effort. This way, the first phase and the second phase of the algorithm are entangled in the implementation. Actually, these phases could be run completely in parallel; but we did not implement it in parallel in order to avoid necessary synchronisations.

Only when the first two phases, local forward construction and backward marking, are finished, the third phase will be started adding currently enabled non-local connectors. Again, it is not necessary to run through all states of the constructed transition system again. During the forward construction, we add all newly created states to a list. Then, we need to check only the states from this list. This way, we need to investigate each state of the final transition system only once for the enabledness of a transition of each non-local connector (the standard forward algorithm guarantees that this is true for all local connectors too).

The last important issue is to avoid the construction of duplicate states. This can be easily achieved by standard hashing techniques, and will, therefore, not be discussed here.

3.4 Complexity

All the above issues are important for increasing the efficiency of the algorithm. It is easy to see that, basically, we have to deal with each state and each transition exactly once during the forward construction. Moreover, the backward marking algorithm visits each state and each transition at most once. Since we have backward marking algorithms for each non-local connector, the time complexity of the construction algorithm is about[1] $O(k \cdot n)$, where n is the size of the constructed transition system (number of states and transitions) and k is the number of non-local connectors of the EPC. Since k is quite small in typical EPCs, the complexity of the algorithm (in time and space) is, basically, linear in the size of the transition system. Practical experience shows, that the algorithm is limited only by the available main memory. For our prototype implementation in Java, we could compute transition systems with one million states and eight million transitions in 45 seconds on a Linux PC with 1 GB main memory and a 2.4 GHz processor. The computation significantly slows down as soon as the main memory is exhausted and parts of the transition system need to be swapped to disk.

The nice feature of this algorithm is that its complexity is independent of the actual size of the EPC. As long as the resulting transition system remains small, the algorithm works fine. Recall that small means in the order of millions of states and transitions. This is in contrast to our symbolic algorithm that computes the complete semantics of the EPC by model checking techniques [CK04b, CK05]. This algorithm might not be able to calculate the transition system – even if the EPC has only a few reachable states. Actually, it was this observation that inspired us to think of another way to calculate the transition system of EPCs with small state spaces (see Sect. 5).

Unfortunately, the algorithm does not always give us a result as we have seen in our example. The algorithm will always terminate, but, sometimes, the resulting transition system P does not meet the condition $R(P) = P$, which is detected during the calculation.

4 Improvements

The algorithm as presented in the previous section provides correct results for many practical EPCs. However, there are some EPCs for which the result is not correct. In this section, we will make some improvements so that the algorithm returns correct results at least for well-designed EPCs – and we believe that the improved version works for all EPCs that are constructed from the four workflow constructs.

In order to explain the idea of the improvement, we have another look at our example from Fig. 4. What went wrong during the construction of the transition system for this EPC was that we added the transition for XOR-join connector K too early to the initial state (see Fig. 7 and 10). Without firing any XOR-join connector at all (first two phases of the first round of the algorithm), no folder can ever reach arc e and arc h. Therefore, the transitions

[1] Actually, there is an additional logarithmic factor for the hashing algorithms and in some worst-case scenarios k might be the number of all connectors.

for both XOR-join connectors G and K were added to the initial state at the same time (see Fig. 7). In this example, however, we can easily see that a folder can be propagated to arc h only after XOR-join connector G has had a chance to fire. So, we had better constructed the transition for XOR-join connector G first, then investigated the states reachable from these states, and then checked for connector K.

4.1 Levels

The basic idea of the improvement is to have different levels of non-local connectors. In the third phase of the algorithm, we check the non-local connectors level by level. If we added a transition for a connector on one level, we do not consider the non-local connectors on all subsequent levels in this round.

For our example from Fig. 4, we chose three levels: D is on the first level, G is on the second level, and K is on the third level. With these levels, the algorithm works as follows: First, we start the forward construction and the backward marking algorithm as in Sect. 3.2. The result is the same as shown in Fig. 6. Then, we check whether a transition for XOR-join connector D (first level) can be added. This is not possible, so we check for XOR-join connector G (second level). The result is shown in Fig. 13. Since, we added a transition for a non-local connector on the second level, we do not check for transitions for connectors on the third level (connector K) in this round. Rather, we start the next round with the transition system from Fig. 13. The forward construction phase will result in the transition system shown in Fig. 14.

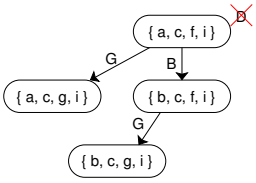

Figure 13: Adding transitions for connector G

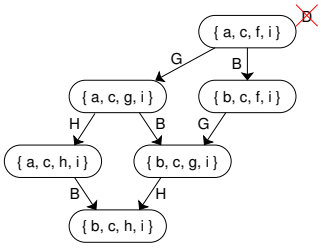

Figure 14: Local forward construction

Next, we start the backward marking phase, which marks the very same states as in our original computation (except for those states that do not occur anymore). The result is shown in Fig. 15.

In the third phase, we check whether transitions for the different levels of non-local connectors can be added. But, it turns out that no non-local transition can be added anymore. So, Fig. 15 shows the final result. This time, there is no transition for a non-local connector starting in a state that is marked. So, the constructed transition system P meets the condition $R(P) = P$, which was called an *ideal* semantics in [Kin04].

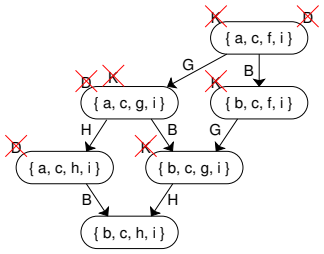

Figure 15: Backward marking algorithm

4.2 Calculating the levels

In our example, we could easily see which non-local connectors should go to which level. In general, however, we need to calculate these levels from the structure of the EPC. Inspired by the above example, we use a simple marking algorithm for the nodes and arcs of the EPC: We start with marking all initial events and set a counter i to 1. Then we proceed with marking the other events and nodes of the EPC as follows (where each node is marked only once):

Events/Functions If an ingoing arc of an event or function is marked, we mark the event or function.

Split connectors If the ingoing arc of split connector is marked, we mark the split connector.

AND-join connectors If all ingoing arcs of an AND-join connector are marked, we mark the AND-join connector.

Arcs If the source node of an arc is marked, we mark the arc.

If no node or arcs can be marked anymore according to the above rules, we proceed as follows:

Fully marked non-local joins First, we check, whether there are unmarked non-local join connectors for which all input-arcs are marked. If this is true, we mark all these connectors and select these connectors for level i. Moreover, we increment the counter i by 1 and start with the first marking phase again

Partially marked non-local joins Otherwise, we check, whether there are unmarked non-local join connectors with at least one marked input-arc. If there is at least one, we mark all these connectors and select these connectors for level i. Moreover, we increment the counter i by 1 and start with the first marking phase again.

Partially marked AND-joins Otherwise, we check, whether there are unmarked AND-join connectors with at least one marked input-arc. If there is one, we mark all these AND-join connectors and start with the first marking phase again.

If no new nodes can be marked anymore, we add all remaining unmarked non-local connectors to the last level and terminate the marking algorithm.

Figure 16 sketches the different phases of the marking algorithm, where the numbers at the different nodes and arcs indicate the level in which the corresponding nodes and arcs were marked. Therefore, the label at each XOR-join connector indicates its level: Connector D is on level 1, connector G is on level 2, and connector K is on level 3.

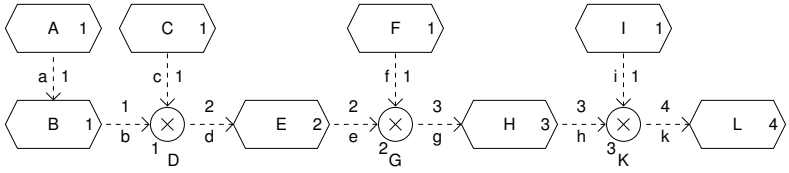

Figure 16: The marking algorithms

4.3 Discussion

The above marking algorithm of an EPC for determining the levels of the different non-local connectors, basically, tries to identify the causal dependencies of the non-local connectors. This order will be considered during the construction of the transition system. The marking algorithm is very efficient; basically, it is linear in the size of the EPC.

Of course, there are EPCs for which the levels calculated by this algorithm do not result in a correct transition system. One example is the vicious circle from Fig. 3; but this is not surprising, because this EPCs does not have a clean semantics at all. We checked our algorithm for several practical examples (e. g. from the SAP/R3 reference models), and it turned out that for almost all examples, our algorithm came up with a correct transition system in virtually no time. There was only one exception, which will be discussed in Sect. 5. Moreover, we believe that for EPCs constructed from properly nested workflow constructs, our algorithm will always result in a correct transition system (a proof of that result, however, is still missing).

5 Practical experience

The quest for another algorithm for simulating EPCs and for calculating their semantics was triggered by some example EPCs from the eJustice project (`www.ejustice.eu.com`), which is funded by the European Union. This project aims at a better understanding

of judicial processes. The discussion started with the EPC shown in Fig. 17 as a screen-shot of EPCTools. This EPC models a German payment order procedure. Here, we do not explain the details of this procedure. Figure 17 should give a rough impression of the structure and size of the EPC model only. The EPC consists of about 90 events, 70 functions, and 40 connectors, where exactly 10 connectors are non-local XOR-joins. One important issue of the eJustice project has been to prove the correctness of the legal procedures originally modelled with ARIS. However, proving correctness requires an adequate simulation tool deriving all reachable states based on a well-defined semantics for EPCs. Although the symbolic algorithm of EPCTools complied with this requirement, it could not cope with the size of the models to be simulated. Therefore, we started thinking of another way of simulating this and similar EPCs.

It was quite obvious that this EPC has only very few reachable states. So, we knew that it should be possible to calculate its semantics somehow explicitly. With our new algorithm, we could calculate the semantics in 70 milliseconds, and it turned out that the transition system had 1215 states and 2551 transitions only.

During tests with practical examples from the SAP/R3 reference models of the ARIS Toolset[2], which were given to us as a benchmark by Jan Mendling, we found only one example for which we could not calculate the semantics. This was the process 'Auftragsabwicklung in der Instandhaltung'; 1GB main memory was not enough to calculate the transition system of this EPC. Therefore, we used a technique proposed in [CK04b] called *chain elimination*, which allows us to reduce the EPC. For this reduced system, we could calculate the semantics. However, the semantics was not clean and there were contact situations, which implied that chain-elimination does not provide correct results. Closer investigation of the model showed that there was an OR-split operation that resulted in quite awkward behaviour of the process. Apparently, this behaviour was not really the intended one. So, we replaced the OR-split by an XOR-split, which appeared to be more adequate. Surprisingly, the resulting EPC could be easily simulated – even without using chain elimination, the transition system could be calculated in 20 milliseconds. This experience shows that long simulation times can give rise to the revision of models, which results not only in better models, but also in faster simulation.

Altogether this shows that, for many practical examples, the new algorithm is a big step forward in the faithful and efficient simulation of EPCs.

6 Conclusion

In this paper, we have presented an new algorithm for efficiently calculating the semantics of an EPC. The idea of this algorithm is explicit forward construction of the transition system combined with a backward marking algorithm, which deals with the non-local conditions.

This idea of a backward marking phase is similar to an algorithm proposed for calculating

[2] ARIS Toolset is a registered trademark of IDS Scheer. For more information see `http://www.ids-scheer.com/`

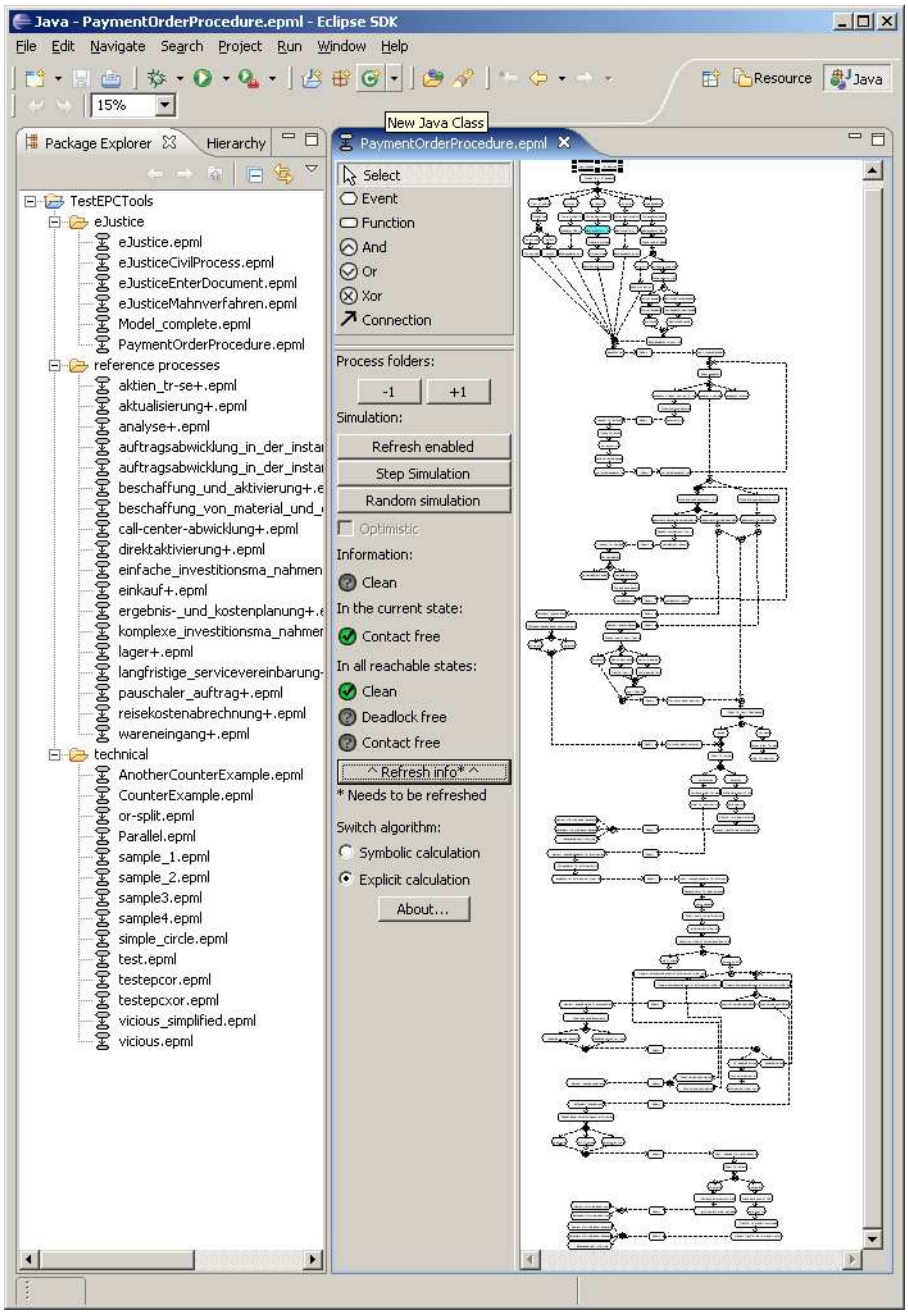

Figure 17: The eJustice example

the enabledness of the OR-join connector in YAWL [WEvdAtH05]; in YAWL, however, only the OR-join connector has a non-local semantics. Moreover, the backward marking algorithm considers only local nodes and the YAWL algorithm does not work in several rounds. Therefore, or algorithm is more adequate for faithfully calculating the semantics of EPCs.

The performance of our algorithm presented in this paper is much better than our symbolic algorithm presented in [CK04b, CK05] – as long as the set of reachable states and transitions are 'small' (i. e. in the magnitude of millions of states and transitions, and, in combination with chain elimination, even billions of states and transitions). The disadvantage of our new algorithm, however, is that in some cases there might be no result at all. Moreover, the new algorithm calculates some ideal transition system. The new algorithm cannot answer the question whether the calculated transition system is the only such transition system or whether there could be more. Anyway, the new algorithm nicely complements the existing algorithm and allows us to simulate (and to analyse) many more EPCs. Experience shows that we could simulate most practical examples based on this algorithm.

The results and measurements come from experiments with a prototype implementation of the new algorithm based on EPCTools. A polished version of this prototype was released in November 2005 as a beta version of EPCTools 2.0. This version supports the explicit and the symbolic simulation algorithm as well as the chain elimination optimisation. It is open source (under the GNU Public License Agreement) and can be downloaded from [CK04a]. What is still missing in this version of EPCTools is functions for analysing the EPC models and for checking some properties. Currently we are investigating and implementing some analysis functions based on the calculated transition system, which will be used for evaluating practical EPCs from the SAP/R3 reference models and from the eJustice project.

References

[CK04a] Nicolas Cuntz and Ekkart Kindler. The EPC Tools Project: Home Page. http://www.upb.de/cs/kindler/research/EPCTools, 2004.

[CK04b] Nicolas Cuntz and Ekkart Kindler. On the semantics of EPCs: Efficient calculation and simulation. In M. Nüttgens and F. J. Rump, editors, *EPK 2004, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten*, pages 7–26, October 2004.

[CK05] Nicolas Cuntz and Ekkart Kindler. On the semantics of EPCs: Efficient calculation and simulation (Extended Abstract). In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management, Second International Conference, 3rd International Conference, BPM 2005, LNCS* 3649, pages 398–403. Springer, September 2005.

[Kin04] Ekkart Kindler. On the semantics of EPCs: Resolving the vicious circle. In J. Desel, B. Pernici, and M. Weske, editors, *Business Process Management, Second International Conference, BPM 2004, LNCS* 3080, pages 82–97. Springer, June 2004.

[KNS92] G. Keller, M. Nüttgens, and A.-W. Scheer. Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten (EPK). Technical Report Veröf-

fentlichungen des Instituts für Wirtschaftsinformatik (IWi), Heft 89, Universität des Saarlandes, January 1992.

[LSW98] P. Langner, C. Schneider, and J. Wehler. Petri Net Based Certification of Event driven Process Chains. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998, LNCS* 1420, pages 286–305. Springer, 1998.

[NR02] Markus Nüttgens and Frank J. Rump. Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In *PROMISE 2002, Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen, GI Lecture Notes in Informatics* P-21 , pages 64–77. Gesellschaft für Informatik, 2002.

[Rit00] Peter Rittgen. Quo vadis EPK in ARIS? *Wirtschaftsinformatik*, 42:27–35, 2000.

[vdADK02] Wil van der Aalst, Jörg Desel, and Ekkart Kindler. On the semantics of EPCs: A vicious circle. In M. Nüttgens and F. J. Rump, editors, *EPK 2002, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten*, pages 71–79, November 2002.

[WEvdAtH05] Moe Thandar Wynn, David Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-Join in Workflow Using Reset Nets. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005, 26^{th} International Conference, LNCS* 3536, pages 423–443. Springer, June 2005.