Transit Case Study

Eric Verbeek[1] and Robert van der Toorn[2]

[1] Eindhoven University of Technology
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
h.m.w.verbeek@tm.tue.nl
http://www.tm.tue.nl/it/staff/everbeek/
[2] Deloitte Management & ICT Consultants
The Netherlands
RvanderToorn@deloitte.nl

Abstract. One of the key issues of object-oriented modeling is inheritance. It allows for the definition of a subclass that inherits features from some superclass. When considering the dynamic behavior of objects, as captured by their life cycles, there is no general agreement on the meaning of inheritance. Basten and Van der Aalst introduced the notion of life-cycle inheritance for this purpose. Unfortunately, the search tree needed for deciding life-cycle inheritance is in general prohibitively large. This paper presents a comparative study between two possible algorithms. The first algorithm uses structural properties of both the base life cycle and the potential sub life cycle to prune the search tree, while the second is a plain exhaustive search algorithm. Test cases show that the computation times of the second algorithm can indeed be prohibitively expensive (weeks), while the computation times of the first algorithm are all within acceptable limits (seconds). An unexpected result of this case study is that it shows that we need tools for checking life-cycle inheritance.

1 Introduction

1.1 Inheritance of Behavior

One of the main goals of object-oriented design is the reuse of system components. A key concept to achieve this goal is the concept of inheritance. The inheritance mechanism allows the designer to specify a class, the subclass, that inherits features of some other class, its superclass. Thus, it is possible to specify that the subclass has the same features as the superclass, but that in addition it may have some other features.

The Unified Modeling Language (UML) [12, 7, 9] has been accepted throughout the software industry as the standard object-oriented framework for specifying, constructing, visualizing, and documenting software-intensive systems. The development of UML began in late 1994, when Booch and Rumbaugh of Rational Software Corporation began their work on unifying the OOD [6] and OMT [11] methods. In the fall of 1995, Jacobson and his Objectory company joined Rational, incorporating the OOSE method [10] in the unification effort.

J. Cortadella and W. Reisig (Eds.): ICATPN 2004, LNCS 3099, pp. 391–410, 2004.

The informal definition of inheritance in UML states the following: "The mechanism by which more specific elements incorporate structure and behavior defined by more general elements." [12]. However, only the class diagrams, describing purely structural aspects of a class, are equipped with a concrete notion of inheritance. It is implicitly assumed that the behavior of the objects of a subclass, as defined by the object life cycle, is an extension of the behavior of the objects of its superclass. In the literature, several formalizations of what it means for an object life cycle to extend the behavior of another object life cycle have been studied; see [5] for an overview. Combining the usual definition of inheritance of methods and attributes with a definition of inheritance of behavior yields a complete formal definition of inheritance, thus, stimulating the reuse of life-cycle specifications during the design process. One possible formalization of behavioral inheritance is called life-cycle inheritance [5]:

> An object life cycle is a subclass of another object life cycle under life-cycle inheritance if and only if it is not possible to distinguish the external behavior of both when the new methods, that is, the methods only present in the potential subclass, are either blocked or hidden.

The notion of life-cycle inheritance has been shown to be a sound and widely applicable concept. In [5], it has been shown that it captures extensions of life cycles through common constructs such as parallelism, choices, sequencing and iteration. In [3], it is shown how life-cycle inheritance can be used to analyze the differences and the commonalities in sets of object life cycles. Furthermore, in [1], the notion of life-cycle inheritance has been successfully lifted to the various behavioral diagram techniques of UML. Also, life-cycle inheritance has been successfully applied to the workflow-management domain. There is a close correspondence between object life cycles and workflow processes. Behavioral inheritance can be used to tackle problems related to dynamic change of workflow processes [4]; furthermore, it has proven to be useful in producing correct inter-organizational workflows [2]. Finally, life-cycle inheritance has also been successfully applied in Component-Based Software Design (CBSD) [13]. In CBSD, inheritance of behavior is used, in particular, with respect to refinement and evolution of software architectures rather than the reuse of components. Different from the work we already mentioned on Petri nets and inheritance [5], in CBSD not only consider the processes of individual components, but also the interactions between several processes of different components are important. This is different from considering a process of a component in isolation. When we consider a process of a component in a software architecture, there is always an environment which is limiting the behavior of the component. These limitations should be considered carefully, otherwise they may lead to undesired behavior.

1.2 Example: A Requisition Process

We use a requisition process as an example how life-cycle inheritance can be used in the workflow domain. Through this requisition process, employees of some company are able to purchase items they need for doing their jobs. After

submitting a request for requisition, the manager of the requestor is selected to approve the requisition. If the requestor has no manager, then the requisition is rejected immediately. Otherwise, two tasks are started in parallel: we record the fact that the approval of the requisition is forwarded to the selected manager, and we notify the requestor of this forward. After both tasks have been completed, we notify the selected manager that there is a requisition for him to approve. If the manager does not react in a timely manner, s/he is notified again, until s/he either approves or rejects the requisition. If s/he approves the requisition, we verify whether the managers spending limit is sufficient for the requisition. If so, the approval of the requisition is recorded and the requestor is notified. If not, the requisition is forwarded for approval to the next higher manager.

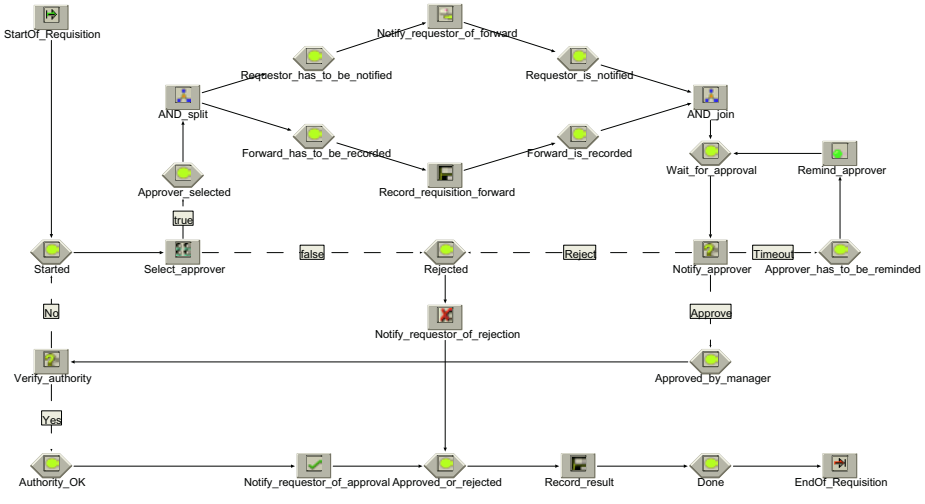

Fig. 1. A process modeled using the COSA Network editor

Figure 1 shows how this requisition process can be modeled using the COSA workflow management system (WFMS). COSA models correspond to Petri nets in a straightforward way: rectangular shapes correspond to transitions, whereas hexagonal shapes correspond to places. In contrast to this, Figure 2 models only a requestor's view on this process using the Protos business process reengineering (BPR) tool. This Protos model also corresponds to a Petri net: rectangular shapes correspond to transitions, whereas circular shapes correspond to places. Using the life-cycle inheritance relation and assuming that tasks correspond to methods, we can answer the question whether this requestor's view matches the process as modeled in COSA.

Table 1 shows a possible combination for hiding and blocking the new methods in the COSA model. If we hide a method, we allow it to occur but ignore its occurrences (it's assumed to be internal, not external), whereas if we block

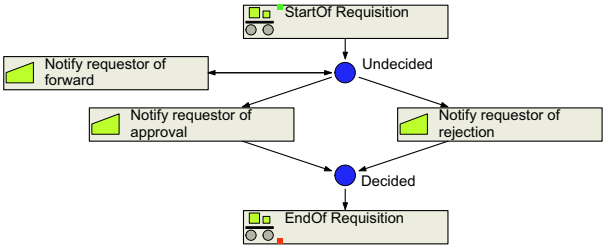

Fig. 2. A process modeled using the Protos tool

Table 1. A possible combination for hiding and blocking new methods

New method	Hide or block?
AND_join	Hide
AND_split	Hide
Notify_approver	Hide
Record_requisition_forward	Hide
Record_result	Hide
Remind_approver	Block
Select_approver	Hide
Verify_authority	Hide

a method, we do not allow it to occur at all. Note that for this example with 8 new methods there would be $2^8 = 256$ possible combinations to hide or block all new methods. Thus, we have to check 256 possible combinations until either (i) we find a combination that results in indistinguishable behavior or (ii) we run out of combinations. In this example, it is straightforward to show that there is no combination that results in indistinguishable external behavior: In the requestor's view, the requestor can get a notification of approval without having received a notification of forward, whereas this is impossible in the COSA model.

1.3 Backtracking Algorithm

The basis of deciding life-cycle inheritance is an equivalence check (to decide indistinguishable behavior), namely a branching bisimilarity check on the state spaces of both object life cycles (the base object life cycle and the potential sub object life cycle). Particularly in the workflow domain, these state spaces can be large (up to millions of states each). Therefore, such a check might be time-consuming despite the fact that efficient algorithms exist to check branching bisimilarity on state spaces [8]. As the example above has indicated, an exhaustive search algorithm for deciding life-cycle inheritance might require many equivalence checks on these state spaces: One check for every possible combination of hiding and blocking the new methods in the potential sub object life cycle, while the number of possible combinations is exponential in the number

of new methods. The combination of the large state spaces and the exponential factor results in an exhaustive search algorithm that is prohibitively expensive in many cases.

In [15], a Petri-net-based backtracking algorithm has been introduced that is based on efficient pruning of the possible combinations. The main goal of this backtracking algorithm is to reduce the number of branching bisimilarity checks, using structural properties of the object life cycles at hand. The first experiments in [15] have shown that this backtracking algorithm does indeed efficiently and effectively reduce the search space. However, in [15] the backtracking algorithm was only compared to the exhaustive search algorithm using toy object life cycles. This paper presents a case study in which both algorithms are compared using a number of real-life object life cycles.

1.4 Overview

Section 2 introduces the case study. Section 3 presents it results and concludes that the life-cycle inheritance relation is absent between some object life cycles. Section 4 discusses how we corrected some of the object life cycles to obtain life-cycle inheritance. Section 5 presents the results after the corrections have been made. Section 6 concludes the paper.

2 The Transit Case

In this section we present the Transit case. The Transit system (Office for Official Publications of the European Communities 2001) is a customs system facilitating the registration and declaration of movements of goods within the Community and EFTA (European Free Trade Area) countries. Within these countries these goods may be moved under certain conditions without payment of the duties and taxes and without having to comply with any other relevant verification measures such as foreign trade requirements. Potential duties and taxes are secured by guarantees which become enforceable in the event of irregularities. The Transit process contains the following steps. A trader in a country (Trader at Destination) buys goods from trader in another country (Trader at Departure). Before the transport of the goods may start, the trader has to declare the goods at the customs office in his country (Customs Office of Departure). If the customs office accepts this declaration, then he has to arrange a financial guarantee which enforces him to fulfill his duties. Next, a customs officer may actually check the goods at the trader premises. If the results are satisfactory, then the goods may be transported. When the goods arrive at their destination in another country, the other trader has to declare the arrival of the goods at the customs office in that country (Customs Office of Arrival). Again the authorities, that are informed by their colleagues in the country of departure, have the possibility to verify the goods (either at their premises or at those of the trader) and if there are no irregularities, then the goods and the financial guarantees are released.

The second author used the Transit case to show how a set of Sequence Diagrams of a system may be used to construct an object life cycle (OLC), in

Table 2. Statistics on the six OLC's of the β case study

OLC	Places	Transitions	Reachable states
transit1	39	24	77
transit2	50	35	88
transit3	54	39	91
transit4	68	51	115
transit5	75	58	143
transit6	99	82	217

terms of a Petri net, which is an integrated system specification that comprises all possible Sequence Diagrams of the system [13]. He started with a simple object life cycle which supports a single Sequence Diagram. Then he extended this object life cycle in a number of iterations into an object life cycle which supports all Sequence Diagrams. In each iteration, he extended the functionality of an object life cycle with a number of new Sequence Diagrams. By using the life-cycle inheritance notion, he checked (manually) each extended object life cycle to make sure that the added functionality did not disturb the already existing functionality. For the remainder of this paper, we refer to this case study as the α case study.

The α case study resulted in figures of six object life cycles, transit1, ..., transit6, which were used by the first author to compare the performance of both the backtracking algorithm and the exhaustive search algorithm. For sake of completeness, we mention that the first author first had to transform these six figures into six object life cycles by hand. Table 2 shows some statistics on these six object life cycles. For the remainder of this paper, we refer to this comparison case study as the β case study. The β case study was performed on a Pentium 4 2.00 GHz computer with 256 Mb of RAM running Windows 2000 SP 3. Please note that this paper presents the results of the β case study but not of the α case study. For the results of the α case study we refer to [13].

3 Performance Results on the β Case Study

Our primary goal with the β case study was to see how the performance of our backtracking algorithm (BA) compares to the performance of an exhaustive search algorithm (ESA). For this reason, we ran both algorithms on every object life cycle (OLC) and its extension, using a prototype of the workflow verification tool Woflan [14, 16] which implements both algorithms. Table 3 shows the results. The computation times mentioned are the computation times of the algorithms

Table 3. Performance results of the β case study

potential sub OLC	Life-cycle inheritance?	Time: BA (in seconds)	(99% conf. interval)	Time: ESA (in seconds)	(99% conf. interval)
transit2	No	7.43×10^{-2}	$\pm\ 2.66\times10^{-4}$	1.69×10^{-1}	$\pm\ 3.52\times10^{-4}$
transit3	Yes	2.19×10^{-2}	$\pm\ 2.40\times10^{-4}$	1.99×10^{-2}	$\pm\ 2.45\times10^{-4}$
transit4	Yes	2.46×10^{-2}	$\pm\ 6.75\times10^{-4}$	2.15×10^{-2}	$\pm\ 2.34\times10^{-4}$
transit5	No	2.40×10^{-3}	$\pm\ 5.43\times10^{-5}$	1.35×10^{0}	$\pm\ 1.07\times10^{-2}$
transit6	No	8.28×10^{1}	$\pm\ 1.30\times10^{-1}$	1.51×10^{2}	$\pm\ 1.95\times10^{-1}$

to check life-cycle inheritance, that is, they exclude set up times and so on. For the remainder of this section, we identify each case by its potential sub object life cycle, because basically we are testing the extension object life cycle and not the extended object life cycle.

The results for cases transit3 and transit4 were to be expected: in both cases, blocking all new methods is a solution. The exhaustive search algorithm always outruns the backtracking algorithm if this is the situation, as is explained in [15]. The results of the other three cases seem normal and quite satisfactory. However, to our surprise, for none of these cases a life-cycle inheritance relation exists between the base object life cycle and the potential sub object life cycle, although this should have been the case.

At this point, it is worth mentioning that, at the moment the β case study was conducted, several researchers knowledgeable in the field of life-cycle inheritance had already took notice of the α case study, and none of them had observed that in three out of five cases the claimed life-cycle inheritance relation was absent. In our eyes, the fact that both these experts and we failed to observe this, shows that one easily underestimates the subtlety of the life-cycle inheritance relation. Thus, we conclude that we need software tools to check for this relation.

4 Diagnosing and Correcting the Processes

Of course, knowing that the α case study would be broken if these inheritance relations were not established in all five cases, we tried to diagnose the absence of these relations, even though, at this point, Woflan does not really support diagnosing errors related to the absence of life-cycle inheritance. Note that diagnosing and correcting these object life cycles was not a goal of the β case study. However, for us, it was necessary to correct the errors, and it was interesting to see to what extent Woflan could be of help.

4.1 Case transit2

Figure 3 shows the original figure[1] for the transit1 object life cycle, whereas Figure 4 shows the original figure for the transit2 object life cycle. Each object life cycle involves four parties who exchange messages. Basically, each party proceeds in the vertical plane, whereas each message proceeds in the horizontal plane. The task S_dec_dat sends the message dec_dat, the task R_dec_dat receives this message, and so on.

In the first iterative step of the α case study, the second author added the behavior related to erroneous message transfers between the two customs offices (the parties in the middle). For example, upon reception of an erroneous aar_snd message, the recipient sends a fun_nck message back to the sender, who will send the aar_snd message again upon reception of this fun_nck message. And so on, until a correct aar_snd message is received. To prevent the sender of the aar_snd message from proceeding while this message has not yet been received, he added a time_out message, although this message was not present in any of the relevant use cases. If a correct aar_snd message is received, the recipient sends a time_out message back to confirm the proper reception of the aar_snd message. Only after this confirmation can the sender of the aar_snd message proceed.

To diagnose why there exists no life-cycle inheritance relation between both object life cycles, we examined the states that were not branching bisimilar to any other state for the situation where the new methods S_time_out and R_time_out are hidden and the new methods S_fun_nck and R_fun_nck are blocked (which seems like a perfect candidate for a life-cycle inheritance relation). In the end, we discovered that the confirmation of proper reception of the arr_adv message is the cause of the absence of the life-cycle inheritance relation. In the transit1 process, the customs office that sends the arr_adv message can proceed to send the des_con message before the other customs office has received the arr_adv message. In the transit2 process, this is impossible, because the customs office that sends the arr_adv message has to wait until the other customs office has confirmed proper reception of this message.

Apparently, the solution chosen by the second author to add the behavior related to erroneous message transfers is not compatible with the life-cycle inheritance relation. This problem is clearly due to the fact that in the use cases, and thus, in practice, the time_out message does not exist. Instead, it seems safe to assume that, in practice, the sender of the arr_adv message will wait for a certain amount of time before proceeding. If the sender does not receive a fun_nck message during that period of time, s/he will proceed. Fortunately, there is a way to capture this behavior, which is shown in Figure 5: The sender will wait for a fun_nck message as long as condition p holds (that is, as long as place p is marked). After this period of time, that is, when condition p does not hold anymore, the customs office that receives the arr_adv message cannot send a fun_nck anymore. Note that this solution makes the time_out message obsolete,

[1] Please note that these figures were the only input for the β case study. For this reason, we did not retouch these figures. As a result, some text might be difficult to read. We apologize for the inconvenience.

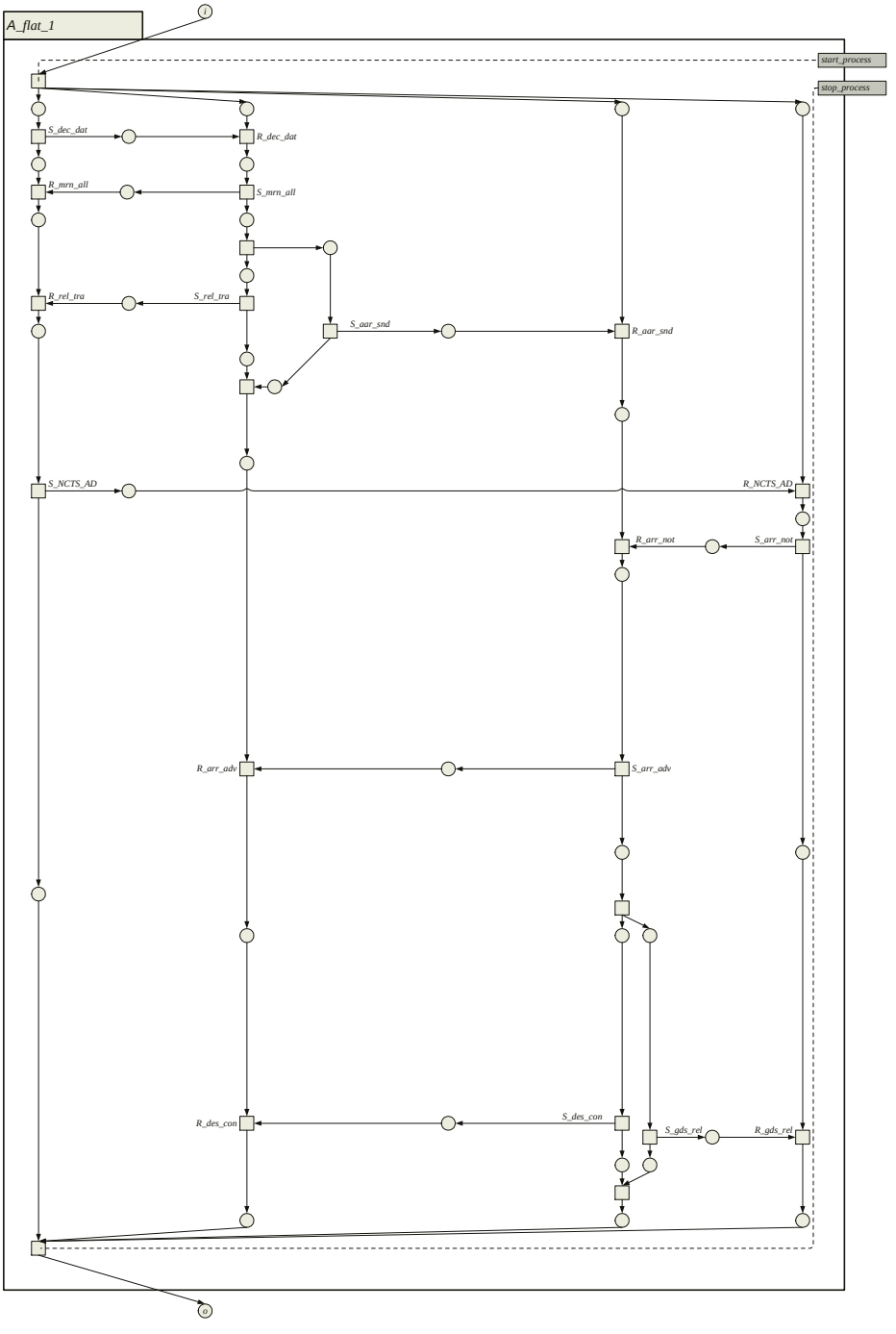

Fig. 3. transit1

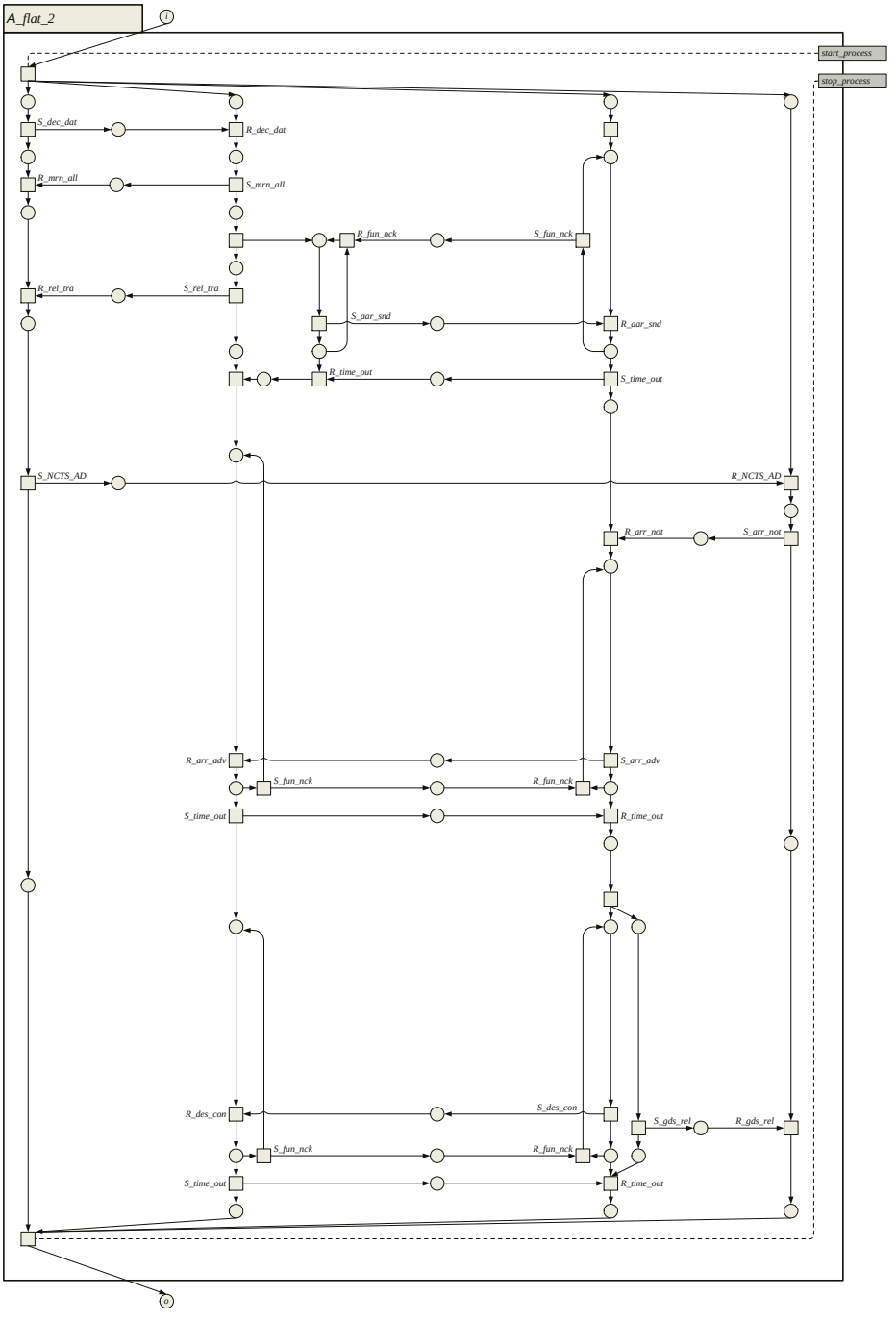

Fig. 4. transit2

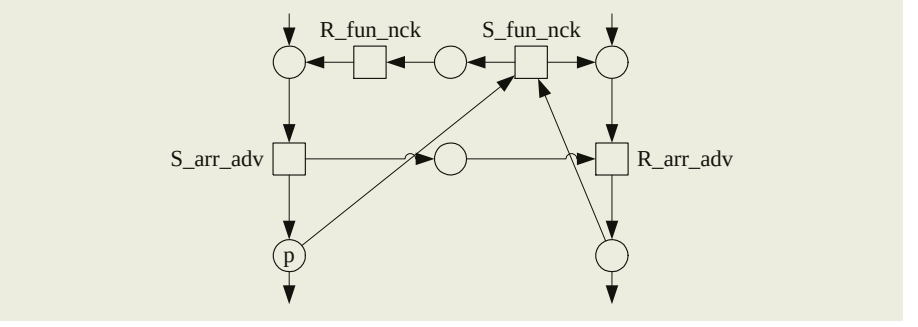

Fig. 5. fun_nck

which is a point in favor, because this message was not a part of the use cases this extension was based on. This led to our first suggestion for correcting the absence of a life-cycle inheritance relation in the transit2 case:

> Hide the methods (tasks) related to the time_out message, remove the place in-between S_time_out and R_time_out tasks, remove the connection from condition p to task R_fun_nck, and add a connection from condition p to task S_fun_nck (as is shown in Figure 5).

4.2 Case transit5

Figure 6 shows the transit4 object life cycle, whereas Figure 7 shows the transit5 object life cycle. In the transit5 object life cycle, a dummy task has been added (labeled GO), and tasks have been added for receiving the message arr_not, for sending the message time_out, and for sending and receiving the messages lar_req and lar_rsp. Note that the messages arr_not and time_out were already present in the transit4 object life cycle. As a result, we cannot hide or block the new methods that receive the message arr_not or send the message time_out. However, it is obvious that we have to hide or block the new methods labeled S_time_out: In the transit4 object life cycle we cannot send a time_out message after having send a mrn_all message, whereas in the transit5 object life cycle we can. Recall that in the previous subsection we already pointed out that the time_out message is obsolete for the extension from object life cycle transit1 to object life cycle transit2. If we would remove the tasks sending and receiving these messages in object life cycle transit2 (and, hence, from the object life cycles transit3, transit4, transit5, and transit6), the method S_time_out would not exist in the transit4 object life cycle, and, hence, we could hide or block it. Thus, applying the suggestion for correcting the previous error might also correct this one. This led to the following suggestion to correct the absence of a life-cycle inheritance relation in the transit5 case:

> Apply the suggestion for correcting the absence of a life-cycle inheritance relation in the transit2 case.

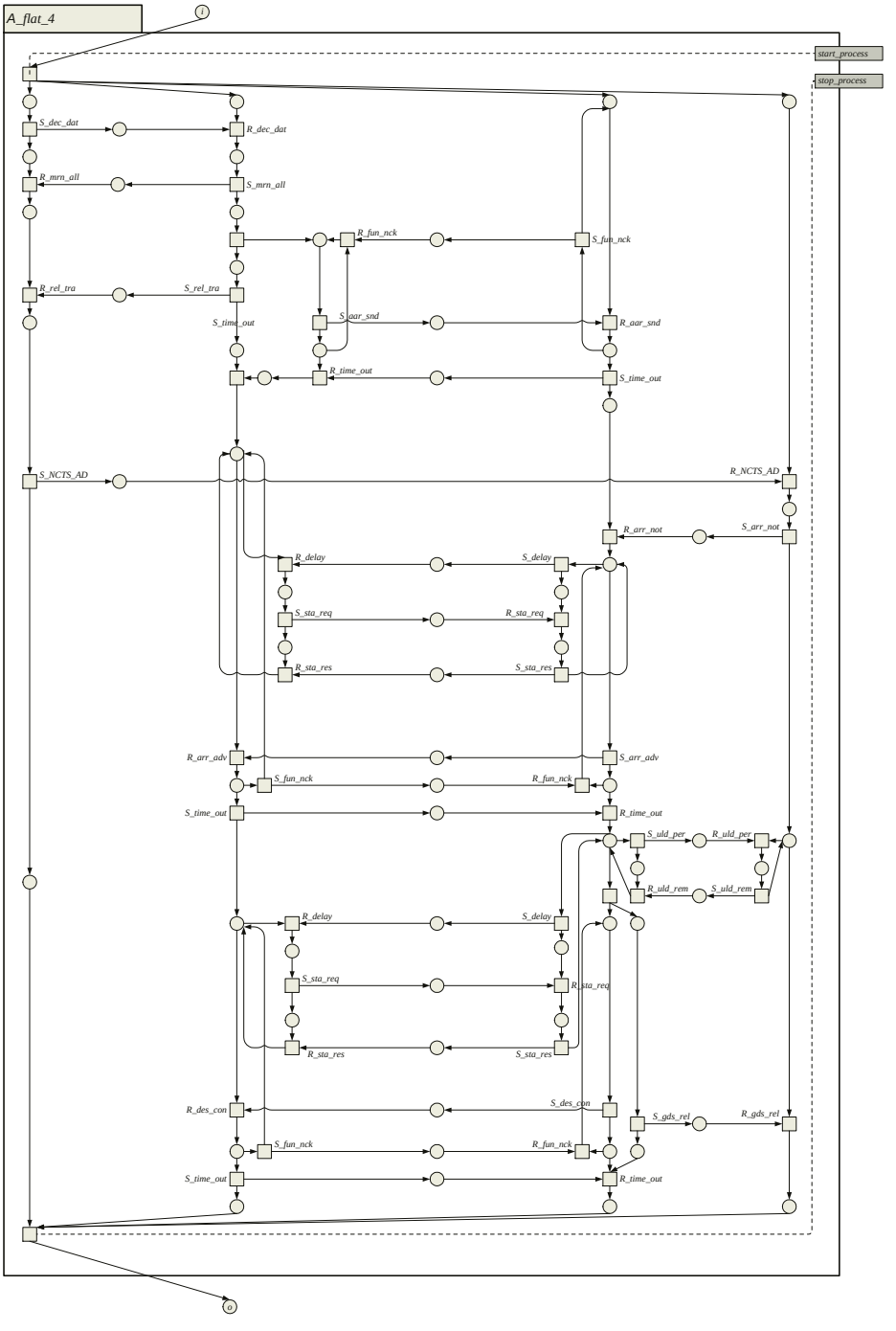

Fig. 6. transit4

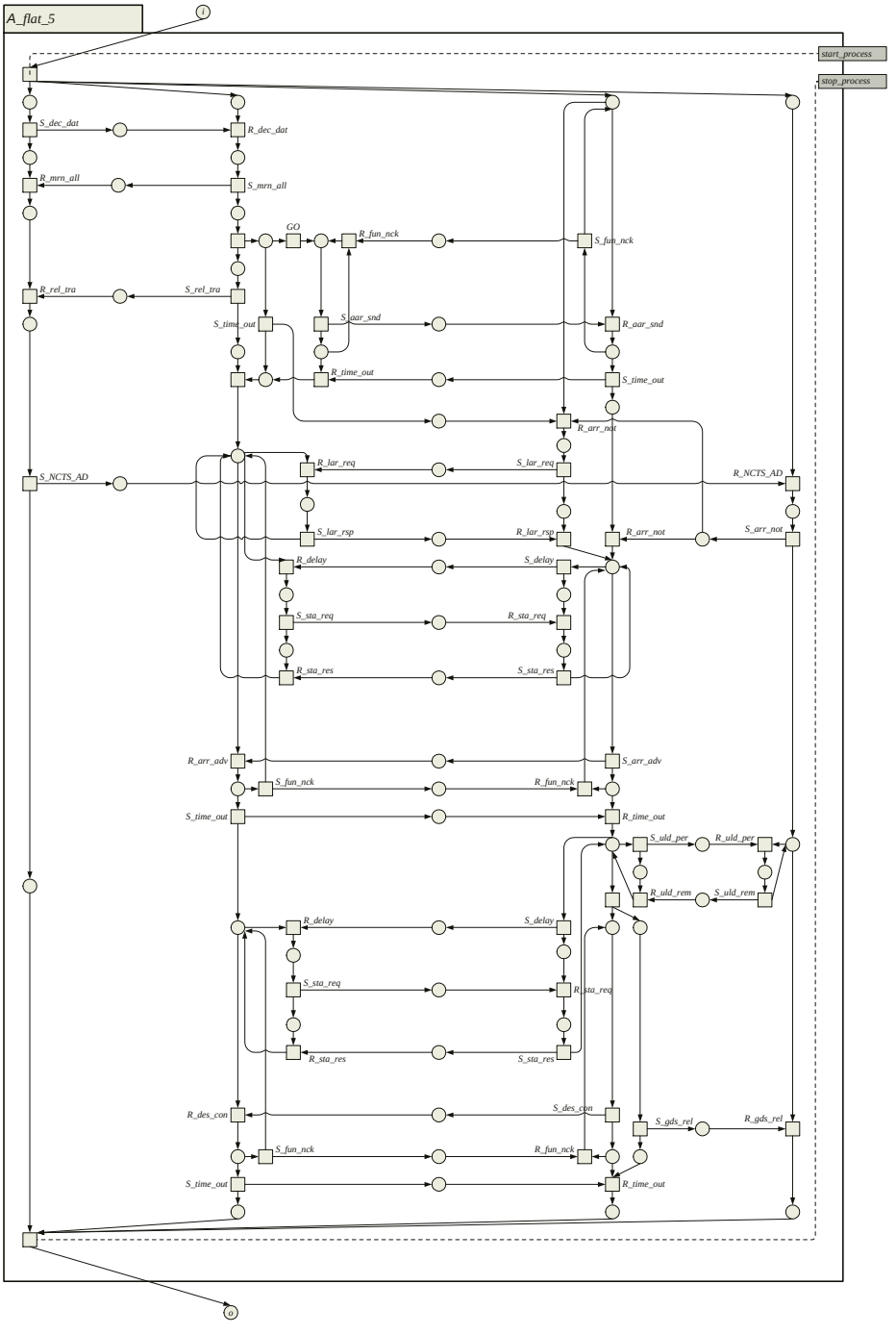

Fig. 7. transit5

Table 4. Statistics on the six OLC's of the γ case study

OLC	Places	Transitions	Reachable states
transit1	39	24	77
transit2	47	35	156
transit3	51	39	171
transit4	65	51	214
transit5	72	58	242
transit6	96	82	338

4.3 Case transit6

Figure 8 shows the transit6 object life cycle. At first sight, it seems that, if we block all new methods, this object life cycle has to be branching bisimilar to the transit5 object life cycle of Figure 7. However, observe that two methods have been added that appear to have no labels (the two input tasks of the lower place labeled S_can_dec). Because these methods appeared to have no labels, we assumed them to be internal tasks. Thus, they had to be hidden. However, this leads to incompatibility with object life cycle transit5 because both these tasks are non-dead in the transit6 object life cycle (because we cannot block method R_delay), and executing one of these tasks leads to a locking problem if all new methods are blocked (because the condition S_can_dec cannot be falsified). This led to the conviction that, when modeling the transit6 object life cycle definition, we misinterpreted the text S_can_dec for a conditions name, where it was meant as a label for both methods we believed to be unlabeled, and thus to the following suggestion for correcting the absence of a life-cycle inheritance relation in the transit6 case:

Label the methods we believed to be unlabeled with the label S_can_dec.

Table 4 shows the statistics of the six object life cycles after we applied the above mentioned suggestions and corrected the object life cycles accordingly. For the remainder of this paper, we refer to this case study on the corrected object life cycles as the γ case study.

5 Performance Results on the γ Case Study

On the corrected object life cycles, we again ran a comparison between the performance of the backtracking algorithm and the performance of the exhaustive search algorithm. Table 5 shows the results. Note that the life-cycle inheritance relation is present in all five cases. The results for the transit2, transit3, transit4,

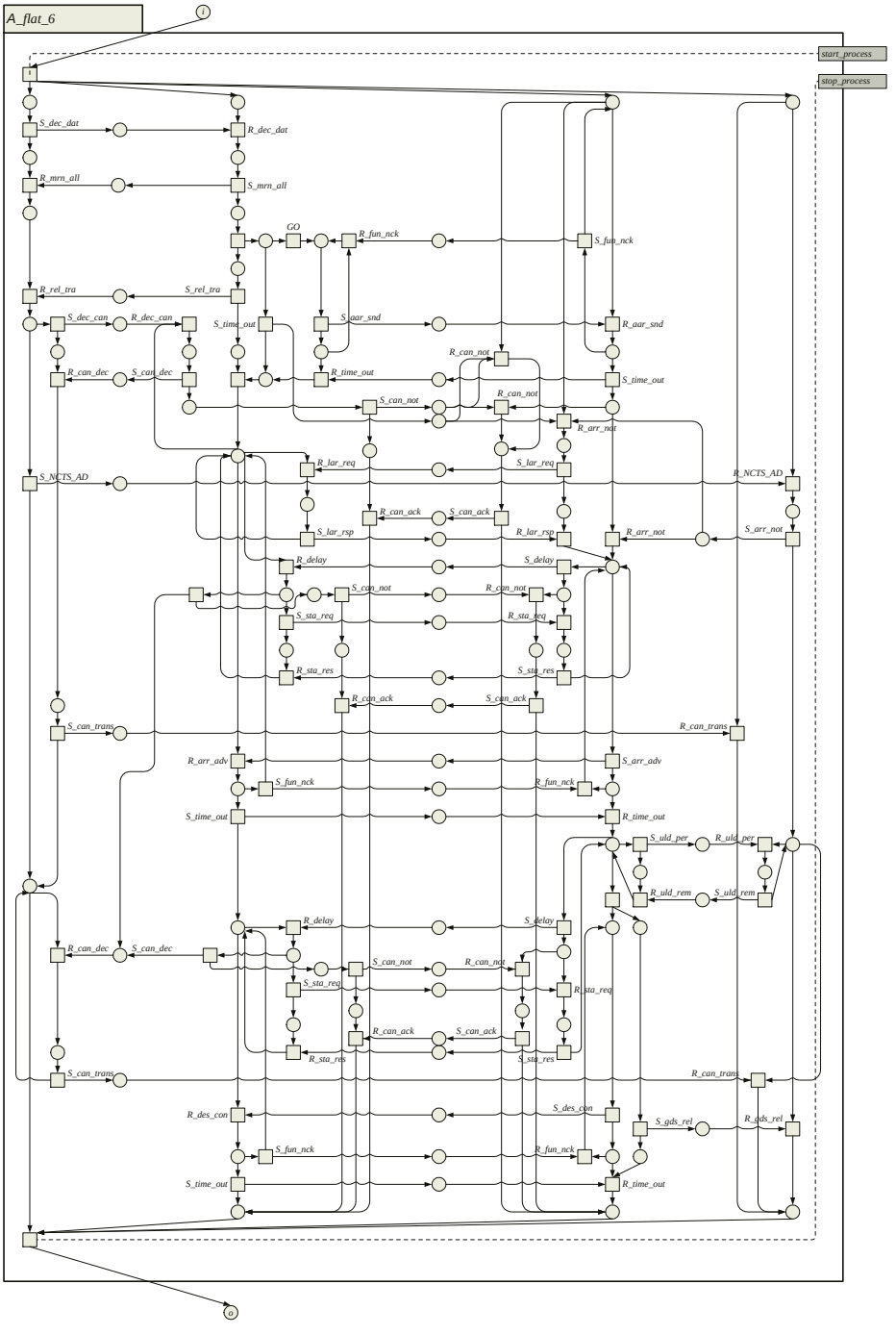

Fig. 8. transit6

Table 5. Performance results of the γ case study

potential sub OLC	Life-cycle inheritance?	Time: BA (in seconds)	(99% conf. interval)	Time: ESA (in seconds)	(99% conf. interval)
transit2	Yes	1.34×10^{-2}	$\pm\ 4.45\times10^{-4}$	1.25×10^{-2}	$\pm\ 4.50\times10^{-4}$
transit3	Yes	3.68×10^{-2}	$\pm\ 4.13\times10^{-3}$	3.44×10^{-2}	$\pm\ 1.78\times10^{-4}$
transit4	Yes	4.78×10^{-2}	$\pm\ 7.16\times10^{-4}$	4.66×10^{-2}	$\pm\ 3.41\times10^{-4}$
transit5	Yes	9.79×10^{-2}	$\pm\ 7.17\times10^{-4}$	1.92×10^{-1}	$\pm\ 1.64\times10^{-3}$
transit6	Yes	1.25×10^{-1}	$\pm\ 5.01\times10^{-4}$	1.19×10^{-1}	$\pm\ 6.54\times10^{-4}$

and transit6 case can be explained by the fact that blocking all new methods is a solution. As mentioned before, the exhaustive search algorithm always outruns the backtracking algorithm if this is the case, because it does not have to compute the constraints.

In the transit5 case, the backtracking algorithm clearly outperforms the exhaustive search algorithm, although the latter algorithm only needs to check branching bisimilarity twice: Hiding new method GO and blocking the other new methods is a solution, and, by chance, method GO is at the bottom level in our search tree. Thus, after the branching bisimilarity check with all new methods blocked fails, the next check is with method GO hidden and all others blocked. It is clear that the position of the GO method in the search tree effects the computation time of the exhaustive search algorithm: After we repositioned label GO at the middle of the tree, the exhaustive search algorithm took $4.85\times10^{-1}\pm1.05\times10^{-3}$ seconds, after we repositioned it at the top, it took $3.17\times10^{0}\pm1.17\times10^{-2}$ seconds. An extreme example illustrating that the position of labels in the search tree can have a dramatic effect on the computation time, is the case when using process transit1 as base WF-net and process transit6 as potential sub WF-net. For this case, the backtracking algorithm took $2.73\times10^{-2}\pm2.10\times10^{-4}$ (method GO at the bottom), $2.71\times10^{-4}\pm2.01\times10^{-4}$ (at the middle), and $2.71\times10^{-2}\pm3.25\times10^{-4}$ (at the top) seconds, but the exhaustive search algorithm took $3.19\times10^{-2}\pm1.78\times10^{-4}$ (at the bottom), $1.45\times10^{2}\pm2.78\times10^{-1}$ (at the middle), or approximately 2.38×10^{6} (at the top) seconds (that is, almost four weeks). We did not measure the latter number, because it simply takes too much time; instead, we extrapolated the previous result in the following way:

- The search tree contains 28 levels (28 new methods), where level 1 is the top level, level 15 is the middle level, and level 28 is the bottom level;

- Thus, finding the solution when method GO is positioned at the middle takes 8193 ($2^{28-15}+1$) branching bisimilarity checks, which took approximately 145 seconds;
- Because finding a solution when method GO is positioned at the top takes 134,217,729 ($2^{28}+1$) branching bisimilarity checks, this will take approximately 2.38×10^6 ($((2^{27}+1)/(2^{13}+1))\times145$) seconds.

Note that we assume that the branching bisimilarity check is the dominant factor with regard to the computation time and that the branching bisimilarity checks, when method GO is positioned in the middle, are representative for all these checks.

6 Conclusion

The main first conclusion of the β and γ case studies is that the backtracking algorithm can outrun the exhaustive search algorithm by orders of magnitudes (for example, a fraction of a second instead of almost four weeks). In contrast with this, the exhaustive search algorithm might outrun the backtracking algorithms too, but usually this is only the case when blocking all new methods yields a life-cycle inheritance relation, and the difference is clearly within acceptable limits.

We can also try to control the exhaustive search algorithm by using intermediate steps, as both case studies shows. As mentioned, when trying to check for life-cycle inheritance between object life cycles transit1 and transit6 in case method GO is at the top level of the search tree, the exhaustive search algorithm takes about four weeks to compute. However, using four intermediate steps, the combination of five exhaustive search algorithms takes approximately 3.29 (0.015 + 0.0344 + 0.0466 + 3.17 + 0.119) seconds. Thus, divide and conquer might also be a good technique to lessen performance problems with the exhaustive search algorithm. However, this might not always be possible.

Another conclusion is that, although the authors of [15] did it for different reasons, it seemed to be a wise decision to try to block a new method before we try to hide it: It seems that, in general, the majority of the new methods needs to be blocked. (Note that the case study performed in the ATPN 2003 [15] also supports this conclusion).

A third conclusion is that we really need a software tool to check whether a life-cycle inheritance relation exists between two object life cycles. In the α case study, a number of knowledgeable people did not detect that in certain cases the claimed life-cycle inheritance relation was absent. Only after we checked this with Woflan, this became apparent.

At the moment, Woflan does not provide any diagnostic information related to life-cycle inheritance, except perhaps for a branching bisimulation relation in case of a seemingly perfect hiding and blocking scheme. Using such a scheme, we were able to correct one absent life-cycle inheritance relation, but it took considerable effort. The other two cases in which the life-cycle relation was absent were diagnosed by simply accepting the fact that this relation was absent and

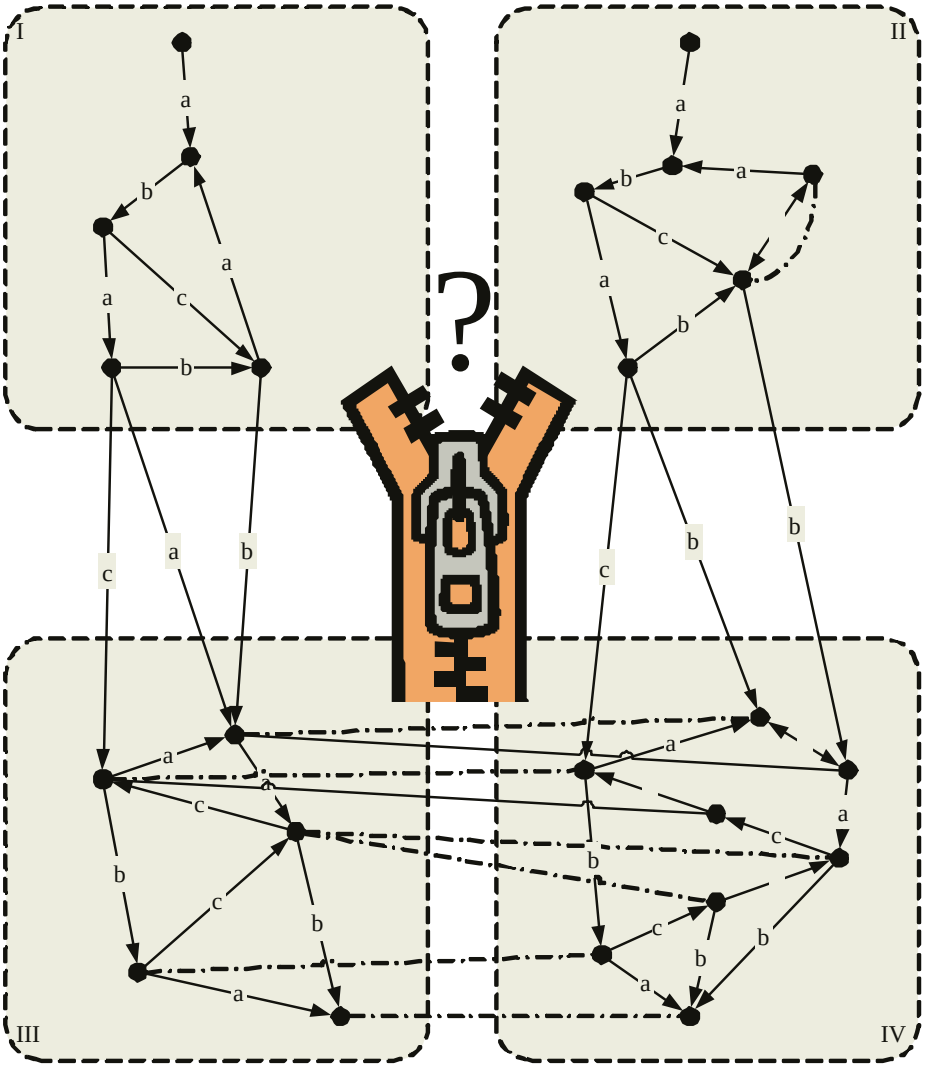

Fig. 9. The 'zipper' scheme

to look for possible causes. However, it would be nice if Woflan could give some guidance when trying to diagnose the absence of a life-cycle inheritance relation. A possible idea would be to have the designer of the process definitions specify a hiding and blocking scheme, and to report back (in some way) the boundary between branching bisimilar states and non-branching bisimilar states. By definition, the successful terminal states are branching bisimilar, and, obviously, the initial states are not. Somewhere between the initial states and the successful terminal states a boundary exists that separates branching bisimilar states

from non-branching bisimilar states. Note that, by definition, arcs can only cross this boundary from non-branching bisimilar states to branching bisimilar states. Apparently, for some reason, on this boundary the branching bisimulation gets lost, which makes it very interesting from a diagnosis point of view. Figure 9 visualizes this 'zipper' scheme. We conclude with a remark concerning a possible positive relation between the computation time needed to check a property and the existence of violations of these properties (errors), that is, in case of errors, the computation time increases. If we rank the computation times for the exhaustive search algorithm as presented by tables 3 and 5 from high to low, then we observe that the cases where no life-cycle inheritance relation is present rank first, second, and fourth. If we do the same for the backtracking algorithm, then they rank first, fourth, and tenth (last). Although the transit5 case of the β case study ranks last, the other cases seem to suggest that the computation time increases in the presence of errors. Our experience is that such a correlation also exists for the soundness property (see [4]). From [4, 5] it is clear that an object life cycle corresponds to a sound workflow net. As a result, life-cycle inheritance is only defined on sound workflow nets. Thus, before checking life-cycle inheritance, we need to check soundness. We used Woflan [16] to check whether the object life cycles where sound. Empirical data on these soundness checks are not presented in this paper, but these data suggest that there exists a positive correlation between the existence of soundness errors and the computation time needed to check soundness.

References

1. W.M.P. van der Aalst. Inheritance of dynamic behaviour in UML. In D. Moldt, editor, *MOCA'02, Second Workshop on Modelling of Objects, Components, and Agents*, pages 105–120, Aarhus, Denmark, August 2002. University of Aarhus, Report DAIMI PB - 561. http://www.daimi.au.dk/CPnets/workshop02/moca/papers/.
2. W.M.P. van der Aalst. Inheritance of interorganizational workflows to enable business-to-business E-commerce. *Electronic Commerce Research*, 2(3):195–231, 2002.
3. W.M.P. van der Aalst and T. Basten. Identifying commonalities and differences in object life cycles using behavioral inheritance. In J.-M. Colom and M. Koutny, editors, *Applications and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 32–52, Newcastle, UK, 2001. Springer, Berlin, Germany.
4. W.M.P. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.
5. T. Basten and W.M.P. van der Aalst. Inheritance of behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.
6. G. Booch. *Object-Oriented Analysis and Design: With Applications*. Benjamin/Cunnings, Redwood City, California, USA, 1994.
7. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1998.

8. J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M.S. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638, Warwick University, England, July 1990. Springer, Berlin, Germany.
9. Object Management Group. Omg unified modeling language. http://www.omg.com/uml/.
10. I. Jacobson, M. Ericsson, and A. Jacobson. *The Object Advantage: Business Process Reengineering with Object Technology*. Addison-Wesley, Reading, Massachusetts, USA, 1991.
11. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewoord Cliffs, New Jersey, USA, 1991.
12. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1998.
13. R.A. van de Toorn. *Component-Based Software Design with Petri Nets: an Approach Based on Inheritance of Behavior*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, January 2004. (forthcoming).
14. H.M.W. Verbeek and W.M.P. van der Aalst. Woflan 2.0: A Petri-net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer, Berlin, Germany, 2000.
15. H.M.W. Verbeek and T. Basten. Deciding life-cycle inheritance on Petri nets. In W.M.P. van der Aalst and E. Best, editors, *24th International Conference on Application and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 44–63, Eindhoven, The Netherlands, June 2003. Springer, Berlin, Germany.
16. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnozing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.