Synthesizing object life cycles from business process models — Source link

Rik Eshuis, Pieter Van Gorp

Institutions: Eindhoven University of Technology

Published on: 01 Feb 2016 - Software and Systems Modeling (Springer Berlin Heidelberg)

Topics: Applications of UML, UML tool, UML state machine, Object Constraint Language and Class diagram

Synthesizing Object Life Cycles from Business Process Models

Rik Eshuis and Pieter Van Gorp

Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands
{h.eshuis,p.m.e.v.gorp}@tue.nl

Abstract. Business process models expressed in UML activity diagrams can specify the flow of stateful business objects among activities. Such business process models implicitly specify the life cycles of those objects. To check the consistency of a business process model with an existing object life cycle or to generate or configure software supporting the business process, these implicit life cycles need to be discovered. This paper presents an approach for synthesizing an object life cycle from a business process model in which the object occurs in different states. The synthesized object life cycles are expressed as hierarchical statecharts. The approach makes implicit life cycles contained inside business process models explicit. The synthesis approach has been implemented using a graph transformation tool and has been applied to case studies.

Keywords: statecharts, UML, model transformation.

1 Introduction

Requirements on software systems that support operational business processes are typically captured in business process models. The Unified Modeling Language (UML) [15] has a notation, activity diagrams, that can be used to model business processes. Business process models expressed in UML activity diagrams can specify both the ordering of business activities as well as the flow of stateful business objects among these activities. State changes of these business objects are due to business activities. The actual object life cycles modeling these state changes are typically expressed as hierarchical object-oriented statecharts [5,15].

Business process models and object life cycles are used differently in the engineering process. Business process models are used in requirements engineering while object life cycles are used in the design phase, for instance to generate code or to configure an off-the-shelf software system. Next, business process models and object life cycles have a different scope. A business process model gives a global view of a business process, addressing different objects, while an object life cycle offers a local view, linked to one aspect of the global process view.

There is a need to relate both views, for instance to check consistency between a global and a local view, or to support traceability between requirements expressed in the global view and a design guided by the local view. However, an important obstacle is that object life cycles are only implicitly specified in business process models.

In this paper, we define an automated approach for synthesizing a hierarchical statechart from a UML activity diagram that specifies a business process model referencing a stateful object. This way, the approach discovers an object life cycle that is hidden in a business process model. The synthesized statechart can be used to generate or configure a software system supporting the business process, or to check consistency with existing statechart descriptions [2,11], either due to legacy systems or to specific industrial or governmental standards like ACORD (`http://www.acord.org`) and SCOR (`http://supply-chain.org/scor`).

The synthesis approach consists of two phases. In the first phase, nodes not relevant for the object life cycle are filtered from the activity diagram. In the second phase, the remaining part of the activity diagram is translated into a hierarchical statechart specifying the life cycle of the object referenced by the process model. The approach is fully automated and has been implemented using the graph transformation tool GrGen [4]. Section 5 gives more details on the prototype and our experiences in applying the prototype to case studies.

In this paper we consider UML2 activity diagrams that use object nodes [15], as explained in Section 2. Activity diagrams also support a pin-style modeling of object flows: then input and output objects of each activity are modeled with pins, which resemble parameters. We do not consider the pin style in this paper, but we plan to address it in future work.

The remainder of this paper is structured as follows. Section 2 gives an overview of the approach using a running example. Section 3 discusses the first phase of the synthesis approach, in which action nodes and irrelevant control nodes are filtered from the activity diagram. Section 4 details the second phase, in which a filtered activity diagram is translated into a hierarchical statechart. There we also explain that the translation may fail, since activity diagrams allow fine grained synchronization not expressible in statecharts. Section 5 presents a prototype that implements the approach and discusses our experiences in applying the prototype in case studies. Section 6 discusses related work. Section 7 ends the paper with the conclusion.

2 Overview

To introduce the salient features of the approach, we show in Fig. 1 an example business process model in a UML activity diagram. The process specifies handling an insurance claim. Atomic activities are represented by action nodes (ovals) like Receive. After receiving the claim, the policy and damage are checked in parallel, indicated with the bar symbols. Next, a decision (diamond symbol) is made to either reject the claim or to accept the claim. In that case, the cost are calculated and paid to the client, and in parallel the periodic contribution to be paid by the client is updated. Finally, the client is notified about the decision.

The process updates stateful object Claim. Each state of Claim is represented by an object node (rectangle). The Claim object can be in multiple states at

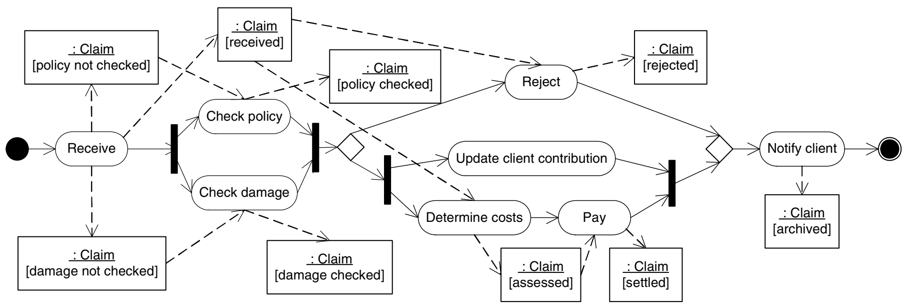


Fig. 1. Activity diagram specifying process for handling insurance claims

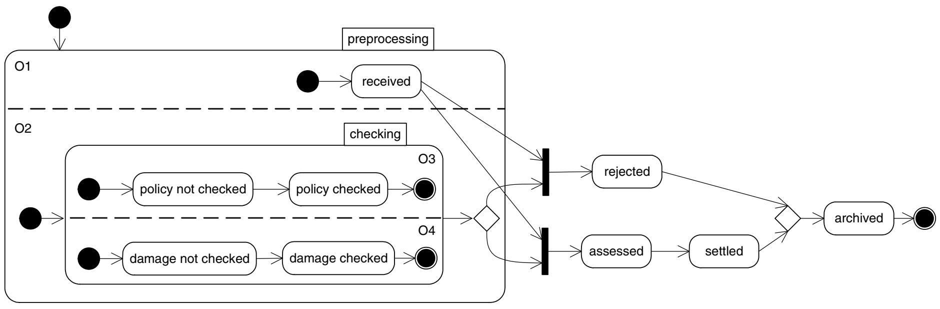


Fig. 2. Statechart specifying life cycle of object Claim in Fig. 1

the same time. For instance, after Receive has completed, the Claim object is in three parallel states: received, policy not checked, and damage not checked. Certain activities change the local state of the Claim object. For instance, Check policy changes the state from policy not checked to policy checked, but does not affect the other states.

Implicitly, the process model specifies the life cycle of the Claim object. Each object node in the activity diagram references a local state of the life cycle. That object life cycle can contain sequence, parallelism, choice, and loops. An example of a choice is the :Claim[received] object node in Fig. 1, which specifies according to the UML2 [15] semantics that object Claim in state received is either input to Reject or to Determine costs but not both.

It is desirable to automatically derive from a process model with an implicit object life cycle an explicit description of that life cycle. Such a description can be used to generate software code or to check consistency with an existing description of the life cycle. Since hierarchical statecharts are the default language for modeling object behavior, the life cycle description should use hierarchical statecharts as well. Fig. 2 shows a hierarchical statechart modeling the life cycle of a Claim object. Note that states received, policy not checked and damage not checked can be active in parallel, just as in Fig. 1. The statechart explicitly models the object life cycle specified implicitly in Fig. 1.

There are two problems that have to be solved in order to derive a hierarchical statechart description from a process model such as the one in Fig. 1. First, some parts of the process model are not relevant for the object life cycle. For instance, the Update client contribution activity does not affect any state of the Claim object, and therefore does not occur in the statechart. These irrelevant parts need to be removed from the process model, but the indirect flows between different object nodes need to be preserved.

Second, statecharts use hierarchical (compound) states, which have no counterpart in process models like activity diagrams. An example of a compound state is preprocessing in Fig. 2 that contains other states like O1 and received. To derive a hierarchical statechart, compound states need to be inferred from the activity diagram syntax. UML 1.5 [14] proposes to translate pairs of fork-join bars to AND states in order to map activity diagrams to statecharts. (A fork is a bar with one incoming and multiple outgoing edges while a join is a bar with multiple incoming and one outgoing edge.) However, in practice many activity diagrams do not satisfy this constraint but can be translated into a hierarchical statechart. For instance, the activity diagram in Fig. 5 contains a fork that matches three joins, but it can be translated to the statechart in Fig. 2.

We define an approach that solves these two problems. Input is an activity diagram specifying the behavior of a stateful object. The approach first filters irrelevant nodes from the activity diagram (Section 3), and next synthesizes a hierarchical statechart from the filtrate activity diagram (Section 4). Both steps are fully automated and do not require any user interaction. Applying the approach to the activity diagram in Fig. 1 results in the statechart shown in Fig. 2, modulo the names of the compound states. The approach may fail, since some activity diagrams cannot be translated into a hierarchical statechart, as we explain in Section 4.3. In that case, diagnostics can be provided giving precise feedback on which part of the process model causes the failure.

We focus in this paper on a single activity diagram with object nodes that references the same object type. If a single activity diagram references multiple object types, then for each object type a version of the activity diagram can be created that references only that object type, by removing from the original activity diagram those object flows and object nodes that do not refer to the object type. If multiple activity diagrams reference the same object type, they can be grouped into a single activity diagram by adding relevant control nodes to connect the different diagrams.

3 Filtering Activity Diagrams

In the first phase of the synthesis approach, nodes not relevant for the object life cycle are filtered from the activity diagram. For the input activity diagram, we require that every action node has one incoming and one outgoing control flow, and every object node has at least one incoming or outgoing object flow. An object node can have multiple incoming or outgoing object flows. The resulting activity diagram contains only object nodes and relevant control nodes and is

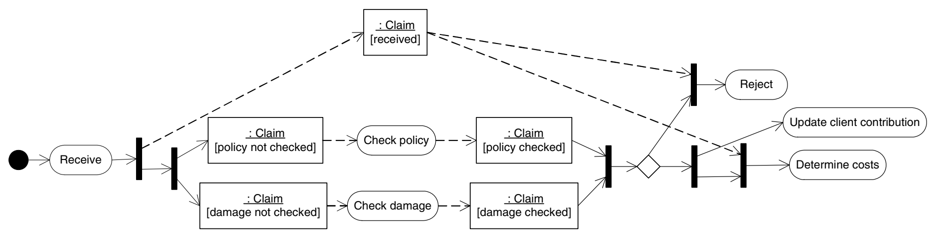


Fig. 3. Part of activity diagram in Fig. 1 after preprocessing

translated into a hierarchical statechart in the second phase, which is explained in the next section.

The filtering phase consists of two stages, which are explained in the sequel of this section. First, the activity diagram is preprocessed and transformed into a normal form. Second, several filtering rules are applied in arbitrary order to the activity diagram. The filtering stage stops if no filtering rule can be applied anymore to the activity diagram.

3.1 Preprocessing

In the preprocessing stage, we transform each activity diagram into a normal form by ensuring that each action node has one incoming and outgoing edge. An activity diagram in normal form has no dangling (object) nodes: each node is on a directed path from the initial to a final node. Preprocessing consists of two steps that performed iteratively in random order until the process model is not changed anymore.

In the first step, we ensure that each object node has at least one incoming and one outgoing edge. If an object node o has no incoming edge, then we take the action node a to which o is input. If a is not unique since o is input to multiple action nodes, this step fails. Otherwise, a is unique and there is a unique control flow that enters a. We change the target of the control flow from a to o. For instance, in Fig. 1 object :Claim[policy not checked] has no incoming object flow. The activity Check policy is targeted by one incoming control flow. The target of this control flow is changed to :Claim[policy checked], as shown in Fig. 3. A symmetrical rule is used for object nodes that have no outgoing edges, such as :Claim[policy checked] in Fig. 1.

However, if a has multiple object nodes as input or output, this preprocessing step is not applicable and the preprocessing fails. For instance, the activity diagram in Fig. 4 cannot be preprocessed; it is not clear whether in the final statechart, state S3 has to be in parallel with state S4 or not. If

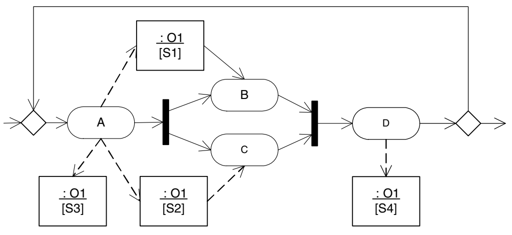


Fig. 4. Ambiguous activity diagram

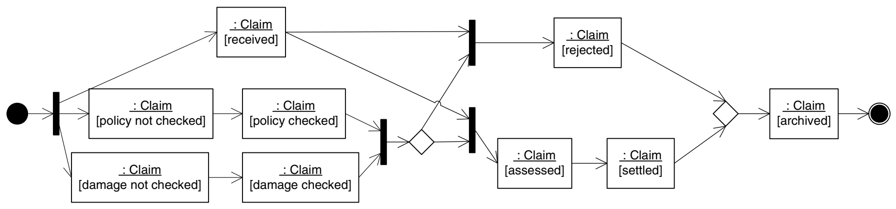


Fig. 5. Activity diagram of insurance claim process (Fig. 1) after filtering

the preprocessing fails for an object node, the user has to rewrite the model into an equivalent model, for instance by adding outgoing edges to the object node.

In the second step, we ensure that each action node gets exactly one incoming and one outgoing edge. By constraint, an action node has one incoming and one outgoing control flow. If an action node also has one or more incoming object flows, a bar is inserted that synchronizes the control flow and the object flows. This synchronization denotes that all inputs of activity need to be present before it can start, which is in line with the UML2 standard [15]. For instance, this step ensures that before both Reject and Determine costs in Fig. 3 bars are inserted that synchronize object flow and control flow. Similarly, if an action node has one or more outgoing object flows, a bar is inserted that splits the control flow and the object flows. For instance, in Fig. 3 after Receive an extra bar has been inserted.

Both steps may introduce control nodes that mix object flow and control flow, which is not allowed by the UML standard. However, this is harmless for the synthesis approach, since eventually object flows have to be translated into statechart control flows anyway.

3.2 Filtering

In the filtering stage, irrelevant nodes are removed. All action nodes are irrelevant, since they do not translate into any statechart construct. Furthermore, control nodes that do not influence object nodes are irrelevant. For instance, the rightmost pair of bars in Fig. 1 are irrelevant, since only one of the two parallel branches between the bars references object nodes, not both. Fig. 5 shows the activity diagram that results from filtering the activity diagram in Fig. 1. As Fig. 5 illustrates, control nodes are only included if they influence object nodes.

The actual filtering is realized by applying different filtering rules. A filtering rule eliminates irrelevant nodes from an activity diagram in normal form. The different filtering rules are applied iteratively in arbitrary order. The filtering step stops if no filtering rule can be applied anymore to the activity diagram. After the filtering rules have been applied, the activity diagram contains no action nodes, but only object nodes and control nodes. The resulting activity diagram is input to the translation from activity diagrams to hierarchical statecharts, detailed in the next section.

We use five filtering rules, R1–R5, which are graphically specified in Fig. 6. Reduction rules R4 and R5 resemble transformation rules defined by Hecht and

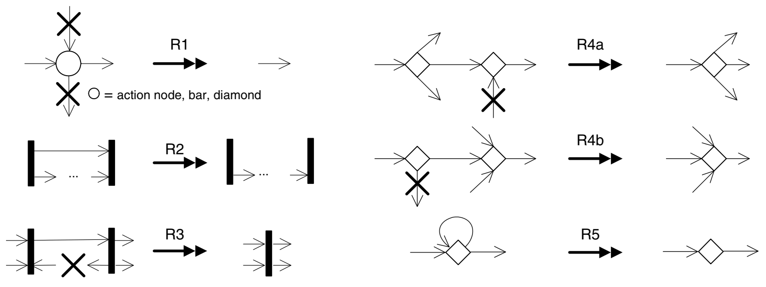


Fig. 6. Filtering rules for activity diagrams

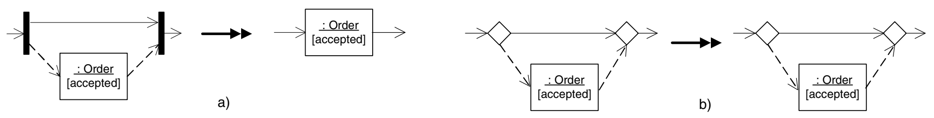


Fig. 7. Example to illustrate different effect of filtering rules

Ullman [7] to test wether a flow graph is reducible [1], i.e. each loop has a single point of entry. Note that flow graphs are sequential, so they do not contain parallelism, as expressed with bars. Moreover, the objective of the reducibility test is to reduce a flow graph to a single node, while the filter approach needs to preserve relevant (object) nodes.

The most interesting feature of the rules is that diamonds and bars are treated in different ways, as illustrated by the example filterings shown in Fig. 7. The models on the right are the result of iteratively applying all filtering rules to the corresponding models on the left. Fig. 7(a) results by applying first rule R3 and next rule R1 two times. In Fig. 7(b), rule R4 is not applicable. We experimented with several other alternative rules for rule R4, for instance a rule for merging two diamonds similar to rule R2. However, such a rules merges the diamonds in Fig. 7(b), which is undesirable as explained before.

4 Synthesizing Statecharts

Output of the filtering phase is an activity diagram containing only object nodes and control nodes. In the next phase, a hierarchical statechart is synthesized from the activity diagram. Basis for the synthesis is an existing, formally defined translation from Petri nets to statecharts [3]. The syntax of Petri nets closely resembles that of activity diagrams. Key difficulty in synthesizing statecharts is the construction of hierarchical (AND and OR) states, which have no counterpart in activity diagram syntax. A synthesized statechart preserves both the structure and behavior of the input activity diagram.

We first explain how the state hierarchy, consisting of AND and OR states, is built from a filtered activity diagram. Next, we explain how the state hierarchy and the filtered activity diagram are used to construct the complete statechart.

Finally, we discuss how the synthesis can fail and how this influenced the definition of the approach.

4.1 Building the State Hierarchy

The state hierarchy is a tree of states. Leaves of the tree are BASIC states while internal nodes are AND and OR states. The tree is visualised by nesting child nodes inside parent nodes. For instance, in Fig. 2 BASIC node policy checked is child of OR node O3 which is in turn child of AND node check.

The AND/OR tree is built in a stepwise fashion, by applying transformation rules to a structure which is a hybrid of activity diagrams and statecharts. The structure contains states and edges resembling activity diagrams, but each state that is source or target of an edge is the root of an AND/OR tree; the states inside this tree are not incident to any edge. Two transformation rules (T1 and T2) are used: one for merging OR states (T1) and one for creating AND states (T2). Each transformation rule reduces edges from the structure but adds state hierarchy. The procedure stops if the structure contains no edges and one state that contains all other states. That state is the root of the state hierarchy. The procedure may fail, in which case the activity diagram cannot be translated into a statechart. Section 4.3 discusses the most common fail case and how this influenced the definition of the overall approach.

We now elaborate the initialization step, in which the initial hybrid structure is created, and the two rules T1 and T2. The next subsection explains how the rules are used to translate an activity diagram into a hierarchical statechart. To simplify the exposition, we do not show the formal specifications of the rules but we apply the rules to the claim processing example.

Initialization. The activity diagram is copied into a new structure. In this structure, each object node is replaced with the BASIC state to which it refers. Then for each BASIC state due to an object node and for each control node, except bars, an OR state is created. The OR state becomes parent of the node for which is created. Finally, the edge relation is for non-bar nodes lifted to their OR parents. So if the activity diagram contains an edge from a non-bar node to another non-bar node, then the structure contains an edge from the OR parent to the other OR parent. If the activity diagram contains an edge incident to a bar, then that part of the edge is not changed. If an edge connects two bars, there

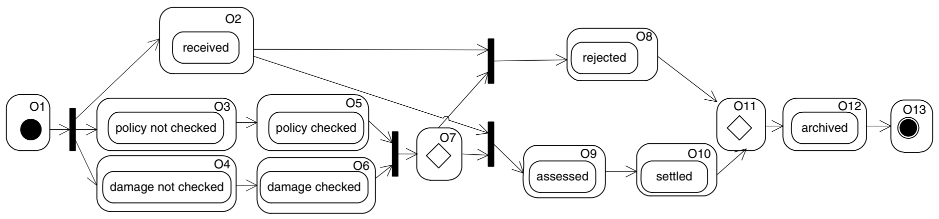
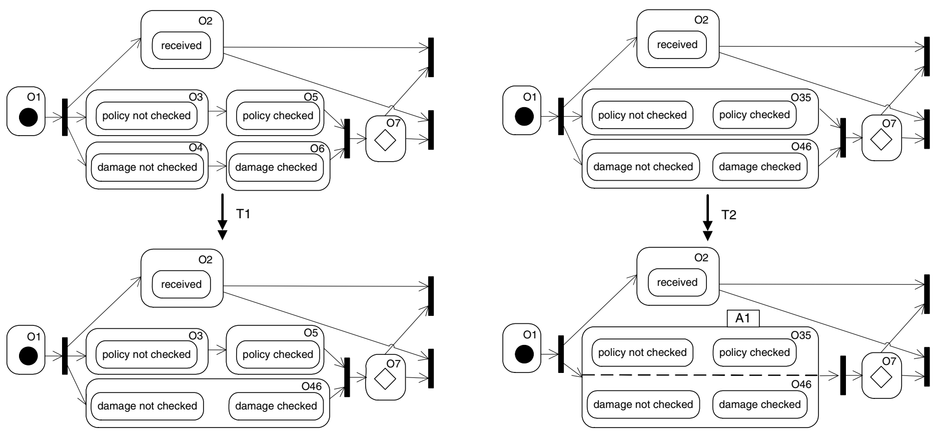


Fig. 8. Initialization of synthesis procedure for filtrate activity diagram in Fig. 5



(a) Example application of rule T1 **(b)** Example application of rule T2

Fig. 9. Example applications of reduction rules on model in Fig. 8

is a cycle (since filtering rule R2 did not remove the edge) and so the activity diagram contains a deadlock, which is undesirable. Fig. 8 shows the initialization of the filtrate activity diagram in Fig. 5.

T1) Merging OR states. If an edge connects two OR states and if there is not a bar such that both states are predecessor or both successor of the bar, then the two OR states can be merged into a new OR state which becomes parent of all children of the two merged OR states. The new OR state replaces the old OR states and the edge connecting the two old OR states is removed from the structure. Note that this does not imply that the edge is not present in the final statechart, since statechart edges are defined separately (see Section 4.2). Figure 9a shows how the rule is applied in the synthesis of the state hierarchy for the structure in Fig. 8: OR nodes O4 and O6 are merged. New node O46 specifies a tree, so the two BASIC states it contains are not connected by an edge.

T2) Creating AND states. Each set of OR states that is input (output) to a bar translates into an AND state, which becomes parent of the OR states. Each pair of OR states in the set should have the same neighboring states. The set of OR states is replaced with a new OR state that becomes input (output) to the bar. The OR state becomes parent of the created AND state. Figure 9b shows an example application of this rule. OR nodes O35 and O46 have the same neighbors and can be grouped under parent AND node A1, which is new. Note that O2 cannot be grouped since it has different successors than O35 and O46. Furthermore, rule T2 is not applicable to the two activity diagram fragments shown in Fig. 9a, since for instance O3 and O4/O46 have different successors.

4.2 Constructing the Statechart

The previous step has resulted in an AND/OR tree of states. We now explain how a hierarchical statechart can be constructed from this tree plus the input

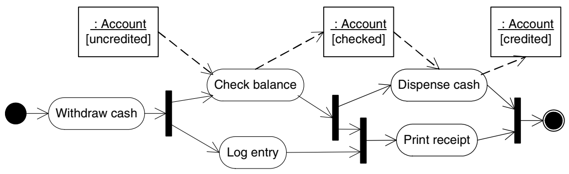


Fig. 10. Process of withdrawing money from ATM

activity diagram. States of the statechart are the states in the AND/OR tree plus additional fork and join pseudo states to represent bars.

Each node in the activity diagram has a unique counterpart in the statechart. If two nodes in the activity diagram are connected by an edge, in the statechart the counterparts of these nodes are also connected by an edge. For instance, object nodes :Claim[policy not checked] and :Claim[policy checked] are connected by an edge in the activity diagram in Fig. 5, so in the statechart BASIC states policy not checked and policy checked are connected by an edge; see Fig. 2.

Since compound statechart states have no counterpart in activity diagram syntax, there are no edges between compound states in the statechart in Fig. 2. We have defined postprocessing rules that rewrite the statechart edges into edges between compound states by eliminating interlevel edges and bars. Applying the translation to the activity diagram of Fig. 5 results in the statechart shown in Fig. 2. The initial and final states inside the AND state preprocessing are due to postprocessing. The names of compound states have been manually defined.

The synthesized statechart can be extended with events as follows (not shown in Fig. 2). Define for each activity A a completion event $cpl(A)$. If A modifies an object to a new state s in the activity diagram, then the corresponding edge that enters s in the statechart before postprocessing can be labelled with $cpl(A)$.

4.3 Discussion

As mentioned at the end of Sections 1 and 2, not every activity diagram can be translated into a hierarchical statechart. Fig. 10 shows a typical example of an activity diagram that cannot be translated directly into statecharts, assuming action nodes map to statechart BASIC states and no filtering rules are applied. After the Withdraw cash action, two parallel branches are started. However, there is a cross-synchronization between the two branches: Print receipt requires that both Check balance and Log entry have been completed. Such a synchronization cannot be expressed in UML 2.3 statechart control flow [15].

However, our approach is still able to synthesize a statechart, since the filtering rules remove the cross-synchronization construct that impedes the translation to statechart. The synthesized statechart contains a sequence of three BASIC states. Our definition of the approach—first filter irrelevant nodes, then synthesize the remaining part into a hierarchical statechart—is motivated by the possible failure cases explained above. An alternative approach would have been to first translate process models into statecharts and then filter irrelevant nodes from the statecharts, so reverse the phases of our approach. Advantage of our approach is that it succeeds in many situations where the other approach fails.

In future work, we plan to analyze problem points for remaining failures and define translation patterns for these cases to enlarge the translation scope.

5 Validation

To evaluate the feasibility of the approach, we have realized a prototype tool that implements the approach and we have tested the tool on several synthetic examples and on two process models that are based on real-life scenarios. In this section, we explain the architecture of the tool and discuss our experiences in applying the approach to the real-life scenarios.

5.1 Architecture

We have decided to implement the rules in Sections 3 and 4 as graph transformation rules using the general purpose graph transformation engine GrGen [4]. This engine provides a scalable implementation for state-of-the-art matching and rewriting constructs and provides especially useful support for visual debugging. Fig. 11 shows the overall architecture of the resulting implementation. The figure's left-hand side shows that that the tool reads activity diagrams expressed in XMI syntax according to the UML 2.3 standard. Such XMI code is generated by mainstream tools like MagicDraw or the Eclipse UML 2 plugin. The rectangle at the top of the figure represents the GrGen platform [4].

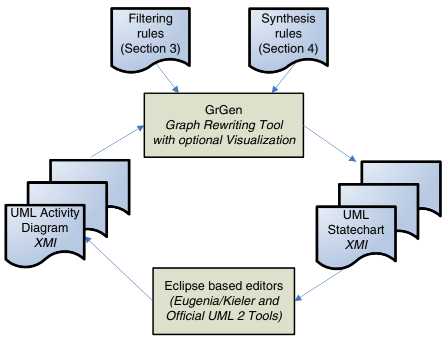

Filtering rules (Section 3)
Synthesis rules (Section 4)
GrGen
Graph Rewriting Tool with optional Visualization
UML Activity Diagram *XMI*
UML Statechart *XMI*
Eclipse based editors
(Eugenia/Kieler and Official UML 2 Tools)

Fig. 11. Implementation architecture

The right-hand and bottom side of Fig. 11 shows that our transformation implementation produces output models that can be consumed by many interesting tools, including Eclipe-based editors. The implementation produces hierarchical statecharts expressed in UML XMI that conforms to the official UML 2.3 standard. Moreover, the implementation produces XMI based on a very simple UML metamodel. The generated XMI can be used among others as input for a statechart simulator that we have developed using state-of-the-art tool Eclipse modeling technology. The architectural strength is that the rules from Sections 3 and 4 do not have to be changed for such extensions.

5.2 Case Studies

To further evaluate the feasibility of the approach, we applied the prototype to two real-life industrial processes that we modeled ourselves: ordering and delivery of bikes, and the handling of dermatology patients. As with the running example, the control flow of the two process models is block-structured (each bar that starts parallel branches has one matching bar that synchronizes the branches) but by preprocessing the object flows, extra bars are introduced that destroy the block-structure (cf. Fig. 3). Table 1 shows that the prototype constructs for each activity diagram a hierarchical statechart in a less than a second. We have shown before that the synthesis procedure defined in Section 4 runs in polynomial time [3] and that the GrGen-implementation scales well for large input models [16].

Table 1. Characteristics of the cases

Case study	# action nodes	# object nodes	# bars	# diamonds	# control flows	# object flows	preproc. & filter (ms)	synthesis (ms)
Bikeshop	15	12	7	4	33	15	297	577
Dermatology	17	5	4	10	39	6	250	280

It would be interesting to let trained professionals design activity diagrams with object flows. An obstacle might be the peculiar semantics of object nodes in combination with action nodes. For instance, inserting in Fig. 1 an object flow from :Claim[settled] to activity Notify client means according to the UML2 semantics that :Claim[settled] is needed for Notify client to start which contradicts the control flow specification that allows that activity Pay is not executed (if Reject is executed instead). This is the reason why many object nodes have either no incoming or no outgoing edge. Such UML modeling issues might make it difficult for professionals to construct activity diagrams with object nodes.

6 Related Work

Our work is most closely related to research on relating object life cycles and business process models (i) and to approaches for synthesizing statecharts (ii).

(i) There are a few approaches [8,9,13] that consider the relation between business process models that reference business objects and object life cycles of these objects. All these other approaches consider flat finite state machines whereas we consider hierarchical statecharts. Moreover, some approaches [9,13] generate a process model from a set of object life cycles, where each object life cycle is specified by a flat finite state machine. Whereas we study the reverse direction: how can an object life cycle be generated from a process model?

Kumaran et al. [8] give an algorithm for deriving sequential, flat state machines from a business process model with data flow. The algorithm focuses on discovering business entities of which the life cycles have to be derived, but in activity diagrams these entities are already modeled explicitly as objects. Next, the algorithm does not use filtering rules, which are essential to enable the discovery of implicitly specified object life cycles from business process models.

(ii) There are several works that study how to generate a statechart from a set of scenarios specified as either MSC or LSC, e.g. [6,17,18]. The constructed statechart satisfies the scenarios, i.e., each scenario is playable with the statechart. An important difference between scenarios and activity diagrams is that scenarios reference statechart events but not states, whereas activity diagrams reference statechart states (object nodes) but not events.

There are two important differences with our work. First, these approaches translate the complete control flow of the scenarios to a statechart. In our approach, we translate object flows to a statechart. Since an activity diagram is a mixture of object flow and control flow, the object flows need to be filtered from the activity diagram. This step is missing in the scenario-based synthesis approaches. The filtering phase is a key element of our approach to discover a hidden object life cycle from a process model.

Second, only a limited set of statecharts can be synthesized from scenarios, compared to the statecharts constructible with our approach. In most approaches [6,18], each scenario-based statechart consists of communicating sequential state machines, so there is one top-level AND state that contains sequential finite state machines. Whereas in our work, constructed statecharts can have concurrency at arbitrary levels of nesting, not just the top level.

A recent approach [17] studies synthesis of hierarchical state machines from UML 2.0 interaction diagrams, which contain activity diagram constructs to specify complex concurrent behavior. However, the interaction diagrams are required to be (block-)structured [10]: each fork matches with a join and pairs of matching nodes are properly nested and loops with multiple exits are not allowed. Consequently, the synthesized statecharts are also block-structured. Whereas the translation defined in Sect. 4 takes as input unstructured activity diagrams and constructs statecharts that can be unstructured, for instance containing loops with multiple exits or unbalanced forks and joins as in Fig. 5.

7 Conclusion

An approach for synthesizing an object life cycle from a business process model, making the implicit life cycle contained in the process model explicit, has been proposed. The approach is fully automated and has been implemented with the graph transformation tool GrGen [4]. Though the approach has been defined for UML activity diagrams, it is also applicable to models expressed in languages like the Business Process Model and Notation (BPMN) [12], provided the models use similar constructs as activity diagrams. Synthesized hierarchical statecharts can be used to generate or configure a software system supporting the business process, or to assess consistency with existing statechart descriptions [2,11].

Future work is to enlarge the scope of the translation in several ways. First, we will define translation patterns that can deal with frequently occurring activity diagram constructs that prohibit a translation into statecharts. Second, pin-style object flows in activity diagrams will be considered. Third, we will define a translation for BPMN models that cannot be expressed as activity diagrams.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)

2. Engels, G., Küster, J.M., Heckel, R., Groenewegen, L.: A methodology for specifying and analyzing consistency of object-oriented behavioral models. In: Proc. ESEC / SIGSOFT FSE, pp. 186–195 (2001)
3. Eshuis, R.: Translating Safe Petri Nets to Statecharts in a Structure-Preserving Way. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 239–255. Springer, Heidelberg (2009)
4. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006)
5. Harel, D., Gery, E.: Executable object modeling with statecharts. IEEE Computer 30(7), 31–42 (1997)
6. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. Int. Journal of Foundations of Computer Science 13(1), 5–51 (2002)
7. Hecht, M., Ullman, J.: Characterizations of reducible flow graphs. J. ACM 21, 367–375 (1974)
8. Kumaran, S., Liu, R., Wu, F.Y.: On the Duality of Information-Centric and Activity-Centric Models of Business Processes. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 32–47. Springer, Heidelberg (2008)
9. Küster, J.M., Ryndina, K., Gall, H.: Generation of Business Process Models for Object Life Cycle Compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 165–181. Springer, Heidelberg (2007)
10. Liu, R., Kumar, A.: An Analysis and Taxonomy of Unstructured Workflows. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 268–284. Springer, Heidelberg (2005)
11. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of statecharts specifications. In: Proc. ICSE, pp. 54–64. IEEE Computer Society (2007)
12. OMG. Business Process Model and Notation (BPMN) Specification, Version 2.0. Object Management Group (2011), `http://www.bpmn.org`
13. Redding, G., Dumas, M., ter Hofstede, A.H.M., Iordachescu, A.: Generating business process models from object behavior models. IS Management 25(4), 319–331 (2008)
14. UML Revision Taskforce. OMG UML Specification v. 1.5. Object Management Group, 2003. OMG Document Number formal/2003-03-01, `http://www.uml.org`
15. UML Revision Taskforce. UML 2.3 Superstructure Specification. Object Management Group, 2010. OMG Document Number formal (May 05, 2010)
16. Van Gorp, P., Eshuis, R.: Transforming Process Models: Executable Rewrite Rules versus a Formalized Java Program. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 258–272. Springer, Heidelberg (2010)
17. Whittle, J., Jayaraman, P.K.: Synthesizing hierarchical state machines from expressive scenario descriptions. ACM Trans. Softw. Eng. Methodol. 19(3) (2010)
18. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: ICSE, pp. 314–323 (2000)