Verification and Synthesis of OCL Constraints via Topology Analysis

A Case Study

Jörg Bauer[1], Werner Damm[2], Tobe Toben[2], and Bernd Westphal[2]

[1] Technical University of Munich, 85748 Garching, Germany,
joerg.bauer@in.tum.de
[2] Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany,
{damm,toben,westphal}@informatik.uni-oldenburg.de*

Abstract. On the basis of a case-study, we demonstrate the usefulness of topology invariants for model-driven systems development. Considering a graph grammar semantics for a relevant fragment of UML, where a graph represents an object diagram, allows us to apply Topology Analysis, a particular abstract interpretation of graph grammars. The outcome of this analysis is a finite and concise over-approximation of all possible reachable object diagrams, the so-called topology invariant. We discuss how topology invariants can be used to verify that constraints on a given model are respected by the behaviour and how they can be viewed as synthesised constraints providing insight into the dynamic behaviour of the model.

1 Introduction

The Unified Modeling Language (UML) [?,?] is widely employed for model-driven development of systems. A fundamental strategy of UML is to support a separation of concerns by different diagram types, in particular to separate structural from behavioural aspects. By means of classes and associations, class diagrams determine structural aspects as *possible* connections (or links) between system objects. By means of states and transitions, state machine diagrams determine behavioural aspects of system objects, in particular modifications of current links.

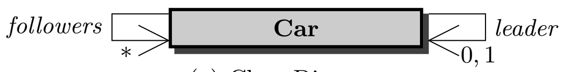
In this article, we address the following problem. Given an executable UML model in form of a class and a state machine diagram, compute (an approximation of) all possible system states (or *object diagrams*) reachable during system run-time. Knowledge about these object diagrams is crucial, because class and state-machine diagrams often allow too many, thus many unintended, object diagrams. Even if one is lucky to have a further annotated model, e.g., annotated by OCL constraints, many unintended object diagrams may arise.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Centre "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

Therefore, we propose a new methodology for computing an over-approximation of *all* reachable object diagrams. While it combines well-established techniques like UML graph grammar semantics and static analysis of graph grammars in a novel manner, it gives the following benefits, on which we shall elaborate in Section 6.

– a *pictorial* overview of all possible object diagrams, whose graphical appeal is one of the major benefits of all graph-based techniques
– the (formal) validation of possibly existing OCL constraints for *every possible* run-time behaviour
– the *synthesis* of OCL constraints, even for non-annotated models (though in few specialised cases only)
– excellent, automatically derivable documentation

We shall now briefly illustrate the problem of unintended object diagrams and its non-triviality with an example. The same example will be used throughout the paper in order to demonstrate the feasibility and usefulness of our methodology.



(a) Class Diagram.

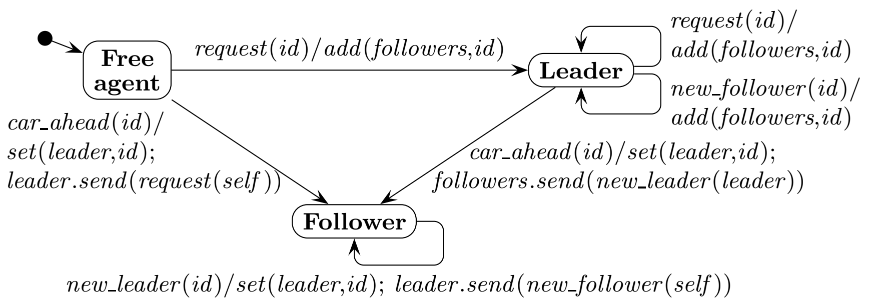

Free agent
request(id)/add(followers,id)
Leader
request(id)/
add(followers,id)
new_follower(id)/
add(followers,id)
car_ahead(id)/
set(leader,id);
leader.send(request(self))
car_ahead(id)/set(leader,id);
followers.send(new_leader(leader))
Follower
new_leader(id)/set(leader,id); leader.send(new_follower(self))
(b) State Machine Diagram.

Fig. 1. Structural and Behavioural model of Car Platooning.

The Problem Illustrated. Consider the task to design a class structure that describes the associations of cars participating in car platooning, i.e. driving in dynamically established convoys (cf. Section 3 for details). In car platooning, a car assumes one of three roles.

(a) It may be part of the tail of a convoy, a so-called follower, then having a link to the platoon leader,
(b) it may be the first car in a convoy, the so-called leader, then having at least one follower, and
(c) it may drive freely as a so-called free-agent, then having neither followers nor leaders.

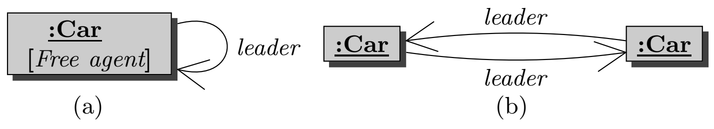


Fig. 2. Unintended topologies.

A viable solution is the class diagram shown in Figure 1(a), as it supports all of the just named three cases. As cars execute concurrently, we've got to employ two unidirectional links to faithfully model transitional situations. For example, during a merge, a follower may have established the link to its leader while the leader has not yet updated its followers.

What the class diagram doesn't say is that these three cases should be the *only* ones. For example, both object diagrams in Figure 2 are legal according to the class diagram but unintended, as a car shall not be its own leader, and cars shall not consider each other to be the leader. Note that from the behavioural model as given by the tiny state machine shown in Figure 1(b), it is neither obvious whether the system remains in the three cases (a)–(c) nor whether the system reaches one of the particular errors shown in Figure 2.

The issue that the class diagram doesn't precisely *say* which object diagrams are wanted can be solved by adding constraints to the class diagram, most naturally in the form of OCL constraints. For example, the constraint

```
Car
(Free agent)                                                    (1)
    implies (leader->isEmpty and followers->isEmpty)
```

formalises case (c) named above, the remaining cases have similar constraints.

But the core problem remains: to analyse whether the system *adheres* to these constraints at run-time. For example, a simple copy-and-paste error during the construction of the state machine could cause *self* to be assigned to the *leader* link in the transition from state *Free agent* to state *Follower*.In this small example, such kind of errors may be excluded by closely considering the actions, but violations of requirements are in general not that obvious.

$$\begin{tikzcd}
\text{UML model} \arrow[r, "\text{UML}", "\text{semantics}"'] \arrow[d, "\text{Section 4}"'] & \text{Reachable object diagrams} \arrow[d, phantom, "\rotatebox{-90}{$\subseteq$}\ \text{Section 6}"] \\
\text{Graph grammar} \arrow[r, "\text{Topology Analisis}", "\text{Section 5}"'] & \text{Abstract clusters}
\end{tikzcd}$$

Fig. 3. Approach. A set of abstract clusters is an abstract description of (a superset of) all reachable object diagrams of a UML model. It is obtained by Topology Analysis from a graph grammar representation of the UML model.

The New Methodology. Our proposal is, assuming a graph grammar UML semantics, to employ a technique called Topology Analysis [**?**] which computes, for a given graph grammar, a concise, finite, abstract description of all graphs possibly reachable by applying the rules of the grammar. The result is, due to the employed abstractions, in general not exact, i.e. it may consider graphs reachable which actually aren't, but it is safe (or sound), i.e. if a graph is reachable in the concrete, then the result of Topology Analysis covers it (cf. Figure 3). As OCL is based on first order logic, most decision problems for OCL are undecidable. On a *finite* characterisation of the reachable object diagrams however, a large fragment of OCL invariants can be evaluated automatically. This does not contradict with the undecidability of OCL as Topology Analysis in general does not compute an exact approximation of the runtime behaviour. Note that neither Topology Analysis nor the specific choice of graph grammar semantics are novel contributions. Rather, our combination and usage of them is unique and beneficial, in particular, in the context of UML/OCL verification. We underpin the usefulness of our methodology by conducting a complex case study.

Structure. Our presentation is structured as follows. Employing the more detailed discussion of the case study in Section 3, we equip a small but relevant fragment of UML with a graph grammar semantics in Section 4. The system described by a UML model is basically a transition system whose states are node and edge-labelled graphs. Objects are nodes which are labelled with the valuation of their attributes and links are edges which are labelled with the association name. The remaining step is then to express actions and event communication in terms of graph grammars. Section 5 recalls the necessary parts of Topology Analysis, in particular the formal definition of topology invariants.

The main contribution of this work is given in Section 6 where we connect topology invariants back to UML models by interpreting them as descriptions of the possible object diagrams. Thereby we gain four things. Firstly, we may evaluate OCL expressions in topology invariants, that is, given a UML model comprising OCL constraints, we can verify that they are satisfied at run-time. Secondly, we can interpret the obtained topology invariant as *synthesised* constraints. A topology invariant may thirdly, in its entirety, provide the developer with an impression of how the system behaves at run-time by giving a concise pictorial overview of reachable object diagrams. This shall in many cases be sufficient to point out subtle design errors. Finally, a subset of the topology invariants may serve for automatically derived documentation. Understanding the intention of data-structures employed in a system necessarily requires object diagrams once a certain model complexity is reached. Generating them automatically eliminates the errors possibly introduced in manual creation of such diagrams. Section 7 concludes and points out further work.

2 Related Work

As far as we are aware, no other abstract interpretation based approach that aims at solving the problem of *computing all possible reachable object diagrams* exists. There are formal verification techniques like [?,?,?] that are able to prove that a given behavioural UML model *always* adheres to properties specified in variants of temporal logic. Due to the complete nature of that approach, it often becomes infeasible in practice. Moreover, it works on symbolic representations of the reachable object diagrams and does not provide direct, graphical access to them; despite the fact that all of these approaches assume finite upper bounds on the number of objects alive at one point in time, that is, only consider under-approximations of the whole system. Abstract interpretation based methods using aggressive abstractions might be a way out. Besides our and the aforementioned methods, there exist tools like UMLAUT [?], VIATRA [?], and USE [?], which allow the interactive or semi-automatic construction of object diagrams from models. However, this exploration is typically not exhaustive.

Apart from computing reachable object diagrams, we are interested in the verification of OCL formulas. USE and VIATRA may be used for evaluating OCL formulas on class and object diagrams, too. However, they are not able to consider *all* possible diagrams for OCL verification. While exhaustive verification techniques are able to do so, they have the well-known scalability issues.

So far, we have summarised related work aiming at the same goal. Below, we take a more technique-centered approach. There is numerous work on graph grammars semantics for UML. The research around the USE tool and graph grammar based UML semantics by Gogolla and others [?,?] is the one we follow for obtaining a graph grammars semantics. Other approaches might be equally well-suited. The technique of Topology Analysis [?] we employ here, has originally been applied in the context of so-called Dynamic Communication Systems [?], which are basically the essence of object-oriented systems, covering dynamic creation and destruction of objects, dynamically changing topology, and asynchronous communication. Just like in the case of UML graph grammar semantics, we have chosen one approach to graph grammars applications. Related to the abstract rule matching in [?], the authors of [?] describe transformation rules for summary nodes (which however do not stem from graph abstraction).

It may be worthwhile to investigate the applicability of other methods to the problem of approximating object diagrams, e.g., [?,?,?], or even the three-valued logic based techniques employed for the analysis of heap manipulating programs, which originate from [?]. Although we will prove the appropriateness of Topology Analysis in this work, the named approaches may be candidates to replace it.

3 Case Study: Car Platooning

We demonstrate our approach on the notably small, but non-trivial and relevant case study of car platooning (cf. Figure 4). Since the late 80's of the last century, people have investigated systematic ways to improve the throughput of highways and to reduce energy consumption [?]. One particular approach is the so-called car platooning. It assumes that cars are provided with communication equipment supporting a kind of ad-hoc network. Cars are notified about other cars driving in front of them which they may then ask, via the communication network, to form a platoon. If the car in front agrees, the back car becomes a follower in the platoon and reduces the safety distance to a minimum. To remain safe, in particular in case of braking manoeuvres, each platoon has a leader which is responsible for notifying its followers about upcoming braking manoeuvres. In the original design [?], communication happens only between a leader and its followers, but not among followers. We adopt this star-like communication structure in our work.

On a more abstract level, a car can fulfill one of three roles. It can be a *free agent*, a *follower*, or a *leader*. Initially, that is, when entering the highway, a car is a free agent. The roles change along three basic manoeuvres, namely *merge* to join cars into a platoon, *split* to split platoons in half, and *change lane*. In the following, for simplicity, we shall concentrate on the merge manoeuvre; the implementation follows the proposal of [?]. The simplest case of merge involves two cars in role free agent, one approaching the other from the back. If the back car is notified about a car driving in front, it requests a merge by sending an according event with its own identity attached, and accepts the car in front as leader. Its role then changes to follower. On receiving the request, the front car assumes the sender as a follower and changes role to leader. In general, both the front and the back car may actually already be platoons, thereby merging free agents into existing platoons or two platoons into a larger platoon. In case there is a whole platoon in the back instead of only a free agent, the protocol is slightly more complicated as the followers of the back platoon have to change their leader and the new leader has to become acquainted with all new followers. To this end, the back leader sends an event announcing the identity of the new leader to all of its followers. These followers in turn update their leader to the new one and announce themselves as new followers by sending an event carrying their identity to the front leader.

We can capture car platooning on this abstract level in form of UML diagrams as follows. Figure 1(a) shows the rather simple class diagram, comprising only a single, active class *Car* with a possible association *leader* to the leader car and an unbounded, possibly empty association *followers* to the follower cars. The behaviour is given by the state machine shown in Figure 1(b). A newly created car starts off in state *Free agent*, with no links. The identification of cars driving in front is abstracted to reception of an event *car_ahead* carrying the identity of the identified car as parameter. Reception of this event causes a state change

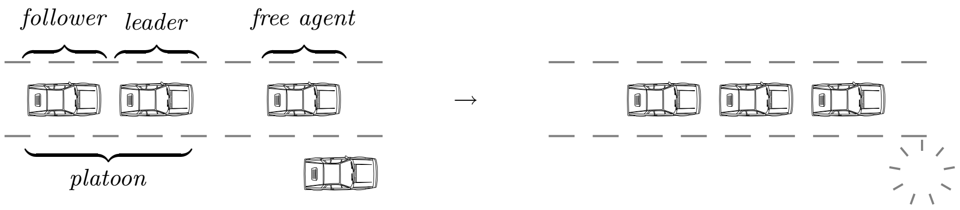


Fig. 4. Car Platooning. A disappearing car and a platoon/free agent merge.

to *Follower* after the leader link has been assigned the received identity and an event *request* with the own identity *self* has been sent to the leader, thereby requesting a merge. A *request* is accepted in state *Free agent* and *Leader*. In both cases, the state changes to *Leader* and the received identity is added to the followers. In state *Leader*, an event *new_follower* announces a new follower when a whole platoon approached from the back and requested a merge. The parameter carried by these events is added to the set of followers. If a platoon approaches a car or platoon in front, this is also announced by the environment with a *car_ahead* message. The back leader changes state to *Follower* after it has set its *leader* link and notified all of its followers of the new leader. Here we assume that the *send* method sends a message to all objects linked as followers. Being a follower, the only expected event is *new_leader* which announces a new leader. The state remains *Follower* after the *leader* link has been changed to the received identity and the new leader has been sent a *new_follower* event announcing the own identity as a new follower.

Note that in the following, we assume an environment which non-deterministically chooses to create instances of class *Car* or to destroy them unless they are in state *Leader*. This models that cars may freely enter and leave the highway. In addition, the environment may send *car_ahead* events to the present instances announcing one of the other present instances as having appeared in front of another car. This can explicitly be added to the model in form of an additional class.

4 Ad-Hoc Graph Grammar Semantics of UML

Our approach as sketched in the introduction is based on abstract interpretation of graph grammars, thus we need a graph grammar semantics for a fragment of UML sufficient to cover our case-study.

Using graphs and graph grammars as a semantical domain for UML as such is not new and rather well-studied, cf. for instance the work summarised in [**?**] and also [**?**], which is more focused on agents than on UML.

In fact, we employ a simplistic variant of the approach proposed in [**?**]. It demonstrates that the particular choice of semantics is not the limiting factor of our approach as we discuss the most relevant features of UML. The semantics is ad-hoc in the sense that it is a minimal setting which is suited to present our approach and we don't intend to provide a formal semantics for each and every syntactical feature of the UML 2.0 standard.

UML Model. Principally following [**?**], for the scope of this paper a UML model is a quadruple $U = (E, C, L, M)$ comprising a finite set E of events, a finite set C of classes, all active, and functions L and M providing classes with associations and state-machines. For each event from E we assume that we're given the information whether it may be sent by the environment or whether it is only used internally in the system, and whether it carries a parameter or not. Given a class $c \in C$, its set of associations $L(c) = \{l_1, \ldots, l_n\}$, $n \in \mathbb{N}_0$, is finite and may be empty. Its state machine $M(c)$ is a quintuple (S, S_0, S_Ω, R, A) comprising a finite set of states S, sets of initial and fragile states $S_0, S_\Omega \subseteq S$, a transition relation $R \subseteq S \times S$, and a transition labelling A assigning each transition $r \in R$ a trigger, a trigger/action pair, or only an action. For the scope of this article, we assume that a trigger is simply an event from E not carrying a parameter, that a trigger/action pair is an event carrying a parameter and an action which manipulates associations and may refer to the parameter, and that plain actions at least comprise association manipulation and event sending. Note that the notion of fragile states is not standard UML, but encodes that cars may non-deterministically be destroyed by the environment we assume (cf. Section 3). For convenience, we assume that states of state machines are disjoint, that is, $S(M(c_1)) \cap S(M(c_2)) = \emptyset$ for classes $c_1 \neq c_2$, which is easily established for any UML model via renaming.

For example, consider the formal representation of the UML model shown in Figure 1. The set of events is $E = \{\mathit{car\_ahead}, \mathit{request}, \mathit{new\_follower}, \mathit{new\_leader}\}$, all carrying parameters and all but *car_ahead* are only used internally. The set of classes is $C = \{\mathit{Car}\}$. The associations of the only class are $L(\mathit{Car}) = \{\mathit{leader}, \mathit{followers}\}$. Its state machine $M(\mathit{Car})$ is (S, S_0, S_Ω, R, A) with states

$$S \supseteq \{\mathit{Free\ agent}, \mathit{Leader}, \mathit{Follower}\}, \tag{2}$$

initial state $S_0 = \{\mathit{Free\ agent}\}$, and fragile states $S_\Omega = \{\mathit{Free\ agent}, \mathit{Follower}\}$.

The semantics of a UML model U is an infinite-state transition system where each state is an object diagram, that is, a set of object instances connected via links. In addition, each object has a sequence of events as event queue. Two such states are in transition relation if and only if the destination state is the outcome of applying an action of an according transition in a state machine of U to a single object in the source state. That is, for convenience we consider a strict interleaving semantics as all classes are active (see above).

As discussed in more detail in [?], this simplistic notion of UML models is not a severe restriction of generality of our proposal as it already captures many essential features by appropriate encodings.

In order to fit into our restricted set of actions, the actual set of states is larger than the ones occurring in Figure 1(b) because the sequential compositions of actions has to be split into atomic actions. For example, the transition from *Free agent* to *Follower* would be split into two transitions by adding an auxiliary state to S (cf. Figure 5). The transition to the auxiliary state is annotated by a trigger/action pair, the action assigns the received identity to the *leader* association. The transition from the auxiliary state is annotated by a plain action, which sends an event to the object denoted by the *leader* association. Note that such operations are semantics preserving in the sense that they neither affect the

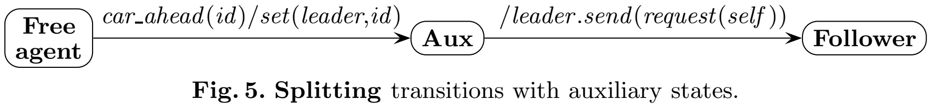


Fig. 5. Splitting transitions with auxiliary states.

reachability of non-auxiliary states nor liveness, that is, whether non-auxiliary states are finally reached. The operations only increase the number of transitions taken during a run-to-completion step.

Furthermore, hierarchical state machines unfold into the flat ones considered here following the well-known procedures (for an example, consider [**?**]). Attributes of finite domains can directly be encoded in an enlarged state set. Similarly, events carrying data of finite domains can be encoded by enlarging the set of events. Methods, unless recursive, can be encoded by "inlining" them into transition annotations. Finally, inheritance can be translated into one class per feature added in a specialisation and a new one-to-one association pointing to the superclass (cf. [**?**]).

Graphs and Graph Grammars. A *graph* is a quintuple (V, E, s, t, l) featuring sets V and E of *nodes* and *edges*, *source* and *target* functions s and t, and a *labelling* function l. Source and target functions map edges to their respective source and target nodes, the labelling function l maps both, nodes and edges, to a label from a finite set of labels.

A *graph grammar* $\mathfrak{G}$ is a finite set of *graph transformation rules.* A graph transformation rule consists of two graphs, a *left graph* L, a *right graph* R, and a relation between them indicating which nodes and edges in L and R correspond to each other. In the rule shown in Figure 6, this correspondence is given implicitly by graphical position. A rule can be *applied* to a graph G if L is a subgraph of G. The result of an application is the replacement of L's occurrence in G with R. For more details, we refer to the textbook [**?**].

Graph Grammar-based UML Semantics. According to the paragraph above, a state of the UML model is an object diagram, where each object is additionally equipped with an event queue. That is, states are graphs where each node represents either an object or an event and each edge a link or possession of an event. Object nodes are labelled with the object's state, event nodes with the event name. Recall from above, that we consider attribute valuations to be encoded into state machine states. Edges to object nodes are labelled by association names, edges to event nodes by the special label μ. Note that, on the level of graphs and within the graph transformation rules, there is no explicit distinction between objects and events, they're both nodes. That is, if we were after an even smaller formal representation of UML models than the one presented above, we could even encode events by having a *class* for each category of events; sending and receiving events would then correspond to creating and destroying instances of these artificial classes.

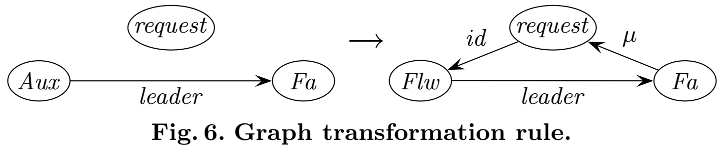


Fig. 6. Graph transformation rule.

The graph grammar of U is then the set of graph transformation rules obtained for the state machine transitions in U. For example, the rule shown in Figure 6 is actually the rule corresponding to the second half of the transition from state *Free agent* to *Follower*. If there are objects in state *Aux* and *Free agent* and if the former knows the latter by link *leader*, then an event node *request* carrying the identity of the former object as a parameter may be sent to the latter. Note that the latter link is labelled with μ as it points to an event, that is, it can be read as pointing to the head of the message queue.

5 Topology Analysis

The technique we employ to compute the possible object diagrams for a given UML model is called *Topology Analysis* (TA) [**?**]. The subject of TA are graph grammars for directed node- and edge-labelled graphs, that is, finite sets of graph transformation rules. For a given graph grammar, TA yields a finite over-approximation, called *topology invariant*, which (abstractly) describes all graphs possibly generated by the graph grammar when applied to a finite set of initial graphs. Technically, topology invariants are obtained by an abstract interpretation [**?**] of graph grammars in the abstract domain of *abstract clusters*. An instance of an abstract cluster is any graph that can be abstracted to it by *partner abstraction*. Partner abstraction of a graph in turn is the quotient graph with respect to *partner equivalence*. Intuitively, two nodes of a graph are partner equivalent if and only if they are similar and if they have similar edges to (sets of) similar nodes, where being similar means having the same label.

More formally, let $G = (V, E, s, t, l)$ be a graph. Two nodes $v_1, v_2 \in V$ are partner equivalent if and only if they have the same label, i.e. $l(v_1) = l(v_2)$, and if for all edge labels a, the nodes reachable from v_1 and v_2 via an edge labelled with a and the nodes reaching v_1 and v_2 via an edge labelled with a have the same label, i.e.

$$\mathit{out}_G(a, v_1) = \mathit{out}_G(a, v_2) \text{ and } \mathit{in}_G(a, v_1) = \mathit{in}_G(a, v_2) \tag{3}$$

where

$$\mathit{out}_G(a, v) = \{l(v') \mid \exists\, e \in E : (s(e), t(e)) = (v, v') \land l(e) = a\} \tag{4}$$

and analogously for incoming edges.

Based on partner equivalence, the partner abstraction $\alpha(G)$ of G is obtained in two steps. Firstly, for each connected component C of G, compute the quotient graph with respect to partner equivalence. Doing so, mark equivalence classes containing more than one node as *summary nodes*. Secondly, summarise isomorphic quotient graphs, that is, keep only one of them. The quotient graphs are called abstract clusters.

As mentioned above, Topology Analysis is an abstract interpretation of a given graph grammar $\mathfrak{G}$ in the domain of abstract clusters. Beginning from the empty abstract cluster, $\mathfrak{G}$ is applied iteratively until a fix-point is reached, which

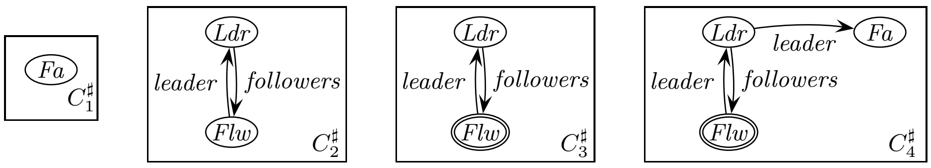


Fig. 7. Abstract clusters. Doubly outlined nodes are summary nodes.

is guaranteed to exist as the abstract domain is finite (cf. [?]). The fix-point is called topology invariant of $\mathfrak{G}$ and denoted by $\mathscr{G}_G$.

Lemma 1 (Soundness of TA [?]). *Let $\mathfrak{G}$ be a graph grammar. If graph G is obtained from the empty graph by applying $\mathfrak{G}$, then $\alpha(G) \subseteq \mathscr{G}_G$.*

Figure 7 shows four abstract clusters of a topology invariant for a graph grammar $\mathfrak{G}$. By Lemma 1, they indicate that the graphs obtainable from the empty graph by applying $\mathfrak{G}$ iteratively may comprise any number of instantiations of abstract clusters and any combination thereof. An instantiation of an abstract cluster is a concretisation in the sense of abstract interpretation, that is, any graph abstracted to the respective abstract cluster. For example, Figure 7 indicates that there may be any number of nodes labelled "*Fa*" (by abstract cluster $C_1^\sharp$), and any number of connected components with two nodes, one labelled "*Ldr*" and the other one "*Flw*" (by abstract cluster $C_2^\sharp$), and any number of connected components with one node labelled "*Ldr*" and at least two nodes labelled "*Flw*" and connected as indicated by abstract cluster $C_3^\sharp$, etc. That is, a topology invariant is an *over-approximation.* It is an abstract description of the set of all possible graphs obtainable from $\mathfrak{G}$, which doesn't miss an obtainable graph but possibly covers more. This kind of approximation is an inherent feature of abstract interpretation based methods and is the price to pay for efficiency. Due to the high complexity of the original problem, we must lose some information somewhere.

6 Reachable Object Diagrams

The abstract clusters shown in Figure 7 are actually a fragment of the topology invariant of the graph grammar representation of the UML model shown in Figure 1. While the graph grammar has been obtained (and improved) manually for this case study, the computation of topology invariants is completely automatic [?]. To keep the number of abstract clusters well manageable, we've assumed a maximal event queue length of 1 during the analysis, which is not a principal restriction of the approach (cf. [?]).

Recall from Section 4 that graphs are used to represent object diagrams, and a topology invariant is an over-approximation of the reachable object diagrams of the UML model we started from. The information represented by a topology invariant can be exploited in many ways, most prominently the following.

6.1 Constraints Verification

The most sophisticated use is to give OCL expressions a semantics on abstract clusters. As abstract clusters are basically graphs, the starting point for such a semantics will be an OCL semantics on graphs as provided by [**?**]. The problem with abstract clusters is that they abstract from certain information in order to remain finite, first of all the number of instances. For example, abstract cluster $C_3^\sharp$ doesn't indicate the *number* of followers a leader may have. That is, one has to be careful when evaluating collection comprehension expressions of OCL, for example `self.followers` which yields a set. The size of this set has to evaluate to the indefinite value `oclUndefined` to remain sound, while `notEmpty` evaluates definite on the same set, i.e. a constraint requiring that an object in state *leader* has at least one follower holds in all abstract clusters shown in Figure 7. That is, the information lost by the abstraction has the effect that some expressions evaluate to `oclUndefined`, while some remain definite values. As OCL is a three-valued Kleene logic, the indefinite value is correctly treated through all arithmetical and logical expressions. Table 1 sketches the treatment of OCL concepts; the only untreatable feature are time expressions (see discussion below).

A system-wide OCL expression like (1) from the introduction is then evaluated for all abstract clusters in the topology invariant. For the considered model, we've established constraint (1) by (manual) evaluation in all abstract clusters. In contrast, the following constraint, which explicitly excludes the unintended topology from Figure 2(b), cannot be excluded by topology invariants.

```
Car
(Leader) implies (leader->leader <> self)
```
(5)

Close inspection of the model unveils that the state machine is too simple to ensure this property. The reason is that two cars may subsequently be announced to each other as driving in front. As there is no further negotiation, they both continue to set their leader link to each other, ending up in the object diagram shown in Figure 2(b). The error can be eliminated by adding further negotiation employing additional acknowledge events. For the corrected version, the topology invariant, and thus the corrected model, satisfies constraint (5).

Given such an interpretation of OCL in abstract clusters, the constraint verification can be conducted automatically for the constraints found in the model as well as for interactive query of constraints.

attribute access (‘`.`’)	node label
association navigation (‘`.`’)	possibly undefined
collection comprehension (‘`collect`’)	possibly undefined
collection operations (‘`->count`’)	possibly undefined

arithmetic expressions (‘`+`’, ‘`-`’)	possibly undefined (indirect)
logical expressions (‘`<>`’, ‘`and`’)	possibly undefined (indirect)
typing, meta-level (‘`oclType`’)	only implicitly
time expressions (‘`@pre`’)	not considered

Table 1. Abstract semantics of OCL constructs.

However, we cannot prove arbitrary properties to hold for any model. This is related to the *property preservation* properties of Topology Analysis. A property is preserved by an analysis, if the fact, that it evaluates to true on every concrete model, implies that it holds true of any abstract model as well. Property preservation is often used to *exclude* undesired behaviour by applying it in its counterpositive form. Whenever something does not hold for a topology invariant, it will not hold for any object diagram of the model. Topology Analysis, for instance, "preserves graphs". If the abstraction of a certain graph does not occur in the topology invariant, then it will not occur in any object diagram. Topology Analysis doesn't preserve all properties. This is the case for all temporal properties, that is, it won't be possible to support the OCL time expression `@pre`, but also for others. A detailed account of property preservation can be found in [**?**].

6.2 Constraints Synthesis

In addition to evaluating given OCL expressions in abstract clusters, we can in some cases translate abstract clusters back to OCL. This is tightly related to the *property reflection* properties of the underlying Topology Analysis. Often, property reflection is much harder than property preservation. A property is reflected, if the fact that it holds on a topology invariant implies that it holds on every object diagram represented by it.

Topology analysis reflects only few properties. Again, we refer to [**?**] for a detailed account. Among the reflected properties are, for instance, edges that do not exclusively involve summary nodes. For example, the abstract clusters in Figure 7 indicate that

```
Car
(Follower) implies (leader->followers->includes(self))
```
(6)

might be a valid constraint of the considered model.

If some additional and automatically checkable technical requirements are fulfilled as well, then such a constraint can be synthesised (automatically). Constraints obtained by this approach may yield valuable, highly condensed insights into the behaviour of the model, comprehensible for every developer trained in OCL. And even hardly comprehensible constraints, for example due to size or nesting, may serve as indicators for regression if they become violated after changes to the model.

Again, we must stress, that only few properties are reflected and, often, it will not be possible to synthesise constraints. However, the fact that is is possible—sometimes even automatically—seems like an important contribution.

6.3 Graphical Appeal for Debugging and Documentation

One of the major benefits of a graph-based approach like Topology Analysis, is its graphical appeal. Our method lends itself for two major purposes: early error detection (debugging) and documentation.

Given the developer's intuition of how the expected object diagrams look, it should in many cases be possible to identify unwanted object diagrams. Experience with implementing our case study shows, that running Topology Analysis already at early design stages, often reveals subtle mistakes. This is mainly owed to the graphical nature of the outcome.

Finally, abstract clusters could give hints for good object diagrams to be used in a system's documentation. As obvious with the minimal UML model example, the class diagram alone is typically not sufficient to understand a model's behaviour at run-time. To this end, good documentation typically comprises characteristic object diagrams. Given a set of good candidates, the only remaining task is to show that they're not spurious, as Topology Analysis is in general not exact (cf. Section 5). We're confident that this task can efficiently be automated employing formal verification techniques. The observation with formal verification tools, in particular the ones employing search-based techniques similar to the SPIN model-checker [**?**], is that they're in average orders of magnitude faster for so called "drive to configuration" tasks than for verification tasks. Tasks of the former kind confirm the reachability of certain "good", or desired system states, while verification establishes satisfaction of temporal properties or the absence of "bad" states for the whole state space. Applying SPIN to UML has been demonstrated, for instance, by Schaefer and others in [**?**].

7 Conclusion

We have proposed a new methodology for approximating all possible object diagrams given a structural and behavioural UML model. Our methodology relies on well-established techniques from the areas of UML graph grammar semantics and graph graph grammar verification. It combines these approaches in a novel fashion. On top of a graphical overview of all possible object diagrams, we expect benefits like OCL constraint verification and synthesis, early error detection, debugging and automated documentation. In fact, the case study presented in this work shows the general feasibility and relevance of the application of Topology Analysis to UML models and fully meets our expectations. Moreover, most of the results were obtained automatically.

As the results presented here are only a case-study, further work clearly consists of fully elaborating this approach. This involves further case studies, thus more experimental results, more automation, and, perhaps, the exploration of other available graph grammar UML semantics and other graph grammar verification methods. In more detail, the formal connection between the specific UML semantics chosen and the specific graph grammars serving as input for Topology Analysis must be established more formally. This may give rise to more automation, too. Furthermore, the abstract interpretation of OCL expressions on abstract clusters has to be fully elaborated. Our first approach as reported in Section 6 clearly indicates the feasibility, but also shows that there is work to be done in order to pass all information from the abstract clusters through to the level of OCL, that is, to obtain a best abstract interpretation.

While we did not experience any scalability problems during our case study, Topology Analysis might be rather costly or imprecise in general, which is not surprising given the complexity of the task. It may thus be beneficial to abstract as early as possible, that is, on an as high language level as possible, for instance, on model level directly rather than on graph grammar level as we propose in our methodology. That is, one should investigate whether there are possibilities to abstract from behaviour of the UML model, for example, certain arithmetics on attributes that don't affect the topology. This will improve the overall scalability of the methodology considerably.

Finally, a promising idea to improve precision was outlined in Section 6: employ formal verification technology but only for the limited (and typically orders of magnitude less expensive) use-case of falsification to confirm the validity of each abstract cluster. This could be conducted after termination of Topology Analysis, on the final topology invariant, or possibly even during the iterative computation constituting the analysis itself. Complementary, the existing criteria for exactness given in [**?**] can possibly be lifted to the level of UML models.