Eliminating Invariants in UML/OCL Models

Mathias Soeken* Robert Wille*
*Institute of Computer Science, University of Bremen
28359 Bremen, Germany
{msoeken,rwille,drechsle}@informatik.uni-bremen.de

Rolf Drechsler*§
§Cyber-Physical Systems, DFKI GmbH
28359 Bremen, Germany
rolf.drechsler@dfki.de

***Abstract*—In model-based design, it is common and helpful to use invariants in order to highlight restrictions or to formulate characteristics of a design. In contrast to pre- and post-conditions, they represent global constraints. That is, they are harder to explicitly consider and, thus, become disadvantageous when the design process approaches the implementation phase. As a consequence, they should be removed from a design when it comes to an implementation. However, so far only naïve tool support aiding the designer in this task is available.**

In this paper, we present an approach which addresses this problem. A methodology is proposed which iteratively removes invariants from a model and, afterwards, presents the designer with invalid scenarios originally prevented by the just eliminated invariant. Using this, the designer can either manually modify the model or simply take the automatically generated suggestion. This enables to entirely eliminate all invariants without changing the semantics of the model. Case studies illustrate the applicability of the proposed approach.

I. Introduction

Modeling languages, such as the *Unified Modeling Language* (UML) [1] being their most famous representative, received much attention in the past. While they are already applied for years in the early phases of complex software projects, also designers of hardware systems, embedded systems, or hardware/software systems begin to use them in order to keep the increasing complexity of today's designs under control [2]. Recently introduced derivatives such as the *Systems Modeling Language* (SysML) [3] underline this trend.

Depending on the considered application as well as on the feature to be specified, these modeling approaches provide several language means on different levels of abstraction (e.g. in terms of diagram types). Additionally, properties and characteristics of the design can further be refined by textual constraints, e.g. given in the *Object Constraint Language* (OCL) [4]. This includes *invariants*, i.e. global constraints that must be satisfied by all system states, and *pre- and post-conditions*, i.e. local constraints that must be satisfied before and after the execution of an operation, respectively. All this enables the designer to precisely outline the desired structure and behavior of a system in the early stages of a design process.

In the modeling process, invariants are a very helpful concept to express global constraints that need to be satisfied in the system under design. For example, they can be applied to exclude "bad" system states, to express restrictions, or to explicitly highlight characteristics of the design. However, when the design process approaches the implementation phase, invariants are disadvantageous. Typical arguments which do not support the use of invariants during and after the implementation are:

- Programming engineers usually prefer pre- and post-conditions as they explicitly describe the nature of the operation to be implemented. Programming languages such as *Eiffel* [5], *D*, *Spec#*, *.NET Framework*, or *Java* [6] even have syntactical support for pre- and post-conditions.
- Certain invariants often affect only some few and very specific operations. But since invariants represent global constraints, it is often not clear which ones. Consequently, *all* invariants need to be considered during the implementation of *every* operation.
- While pre-conditions (post-conditions) have to be checked only before (after) the execution of an operation, invariants need to be valid at every point in time. This impedes their validation in the implemented system.
- During the execution of safety critical systems, one might want to explicitly avoid entering an invalid or "dangerous" state. Invariants can only confirm that the current system state is already invalid. In contrast, a violated pre-condition can prevent an operation which would lead to an invalid state from being called.
- If the correctness of an implementation should be verified (e.g. by property checking methods [7]), always all invariants have to be considered. But since certain invariants often cover only very specific system states, this leads to an unnecessarily large number of constraints to be checked.

978-3-9810801-8-6/DATE12/©2012 EDAA

As a consequence, invariants should be removed from a design when it comes to an implementation. However, getting rid of invariants is a crucial task which requires a comprehensive design understanding and always bears the risks of inadvertently introducing unwanted or even illegal behavior to the system. Thus, design methods are needed which eliminate invariants without changing the nature and the properties of the original specification.

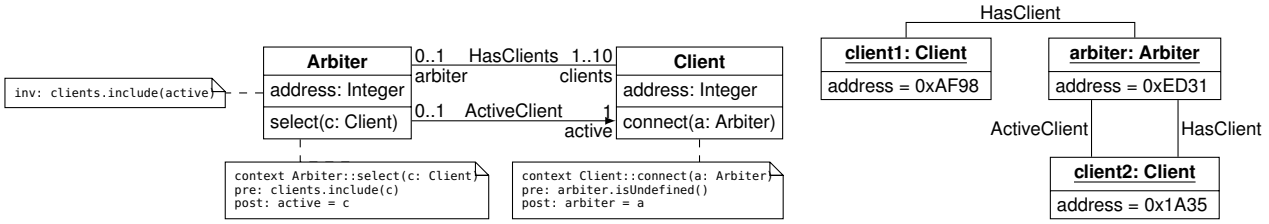
In this paper, we present an approach which aids designers in this task. More precisely, we propose a methodology which iteratively removes all invariants from a given specification. In each iteration, our approach pin-points the designer to invalid scenarios which originally would be prevented by the just eliminated invariant. For this purpose, the designer is guided by given options to substitute the invariant e.g. by adding a new or modifying an existing pre- or post-condition. Moreover, even automatic suggestions for such substitutions are presented and can be incorporated.

By means of a case study the applicability of this methodology is demonstrated. Applying our approach enables to substitute invariants by alternative constraints. Furthermore, our approach ensures that after removing the invariant from the model, the additional constraints prevent the system from reaching invalid states. While we describe and evaluate the proposed approach by means of UML/OCL class diagrams, the general concept can similarly be adapted to other modeling languages. To the best of the knowledge, this is the first nontrivial approach aiding the designer in this task.

II. Background

A. Class Diagrams

A UML *class diagram* is used to represent the structure of a system. The main component of a class diagram is a *class* that describes an atomic entity of the model. A class itself consists of *attributes* and *operations*, where attributes describe the information which is stored by the class and operations define possible actions that can be executed in order to change attributes. Classes can be set into relation via *associations*. The type of a relation is expressed by *multiplicities* that are added to each association-end.



(a) A class diagram (b) An object diagram

Fig. 1. UML class diagram, object diagram, and sequence diagram

Example 1: Fig. 1(a) shows a UML model specifying an arbiter. The class diagram consists of the two classes Arbiter and Client. Both classes have an attribute address of type Integer. An arbiter is connected to a client by two associations expressing two different roles of relationship. First, an arbiter has clients that are connected to it. This is expressed by the association HasClients. As expressed by the multiplicities, each arbiter must be connected to at least one and to at most ten clients, whereas a client can only be connected to at most one arbiter. Second, an arbiter has one special client that is active, expressed by the association ActiveClient. Both classes have an operation, i.e. the arbiter can select a client to be active, and the client can connect itself to an arbiter.

B. OCL Expressions

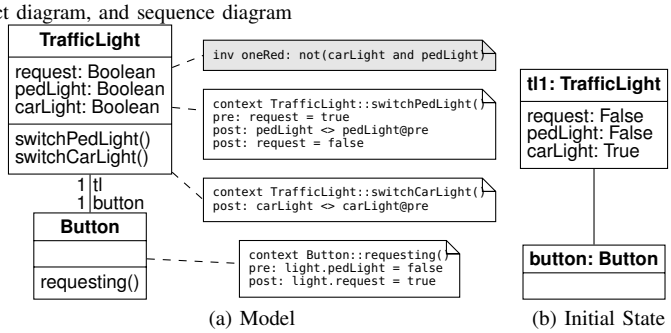
In general, UML class diagrams are not very restrictive. In fact, dependencies and properties of a class diagram can be specified by means of multiplicities at the associations only. In order to express further properties or restrictions, textual constraints provided by OCL can be added to a model. OCL expressions may appear as both, *invariants* or as *pre-* and *post-conditions* of operations. Invariants are global constraints that must be satisfied by all system states. Pre- and post-conditions are considered only locally in the context of an operation call. More precisely, an operation can only be invoked if its corresponding pre-condition is satisfied. Afterwards, the following system state needs to match the operation's post-condition.

Example 1 (cont'd.): The class Arbiter in Fig. 1(a) is extended by an OCL invariant inv which states that active clients must be in the set of connected clients. Furthermore, the functionality of the operation select is further specified. According to its pre-condition, select can only be invoked if and only if the client c to be selected is connected to the arbiter. The post-condition states that, after the execution of the operation, c has to be the active client. Similarly, the operation connect can only be called by a client, if no connection to the arbiter already exists, i.e. only if connect is undefined. After the execution of the operation, the client has to be connected to the arbiter a.

C. Object Diagrams

Object diagrams represent a precise instantiation of a class diagram. They are also called *system states* or *snapshots*. Therefore, one class diagram can provide the basis for several object diagrams. In the object diagram, classes from the model are instantiated as *objects* whose attributes are assigned precise values and associations are instantiated as precise *links* which connect objects.

An object diagram is called *valid* if all restrictions from the class diagram are met, i.e. both the multiplicities at the associations and the invariants are satisfied. Operations and, therefore, pre- and post-conditions have no effect in object diagrams since only one single state in time is described by them.



(a) Model (b) Initial State

Fig. 2. Simple Traffic Light Preemption

Example 2: A valid object diagram which is instantiated from the class diagram in Fig. 1(a) is outlined in Fig. 1(b). Two clients client1 and client2 are connected to one object arbiter. The attributes can assume 16-bit values.

III. Eliminating Invariants in UML/OCL Models

A. Problem Formulation

As discussed in Section I, design methods are needed which eliminate invariants without changing the nature and the properties of the original specification. To the best of our knowledge, so far only naïve methods address this problem, i.e.

- all invariants are just removed from the design without any further consideration or
- each invariant is added as a post-condition to each operation call.

However, while simply removing all invariants would actually change the semantics of the model and, thus, prevents the implementation from detecting invalid system states, also the second approach is disadvantageous. Here, the post-conditions would become significantly more complex which makes the implementation much harder and inefficient. Furthermore, many invariants are not even affected by each operation.

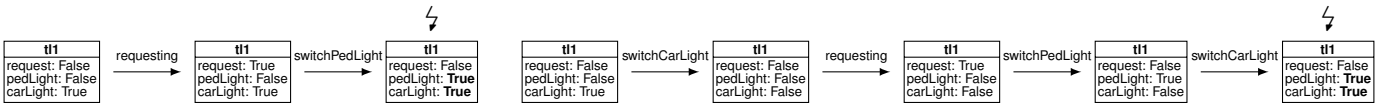
Motivated by this, the following problem is addressed in this paper:

How can invariants efficiently be eliminated from a given model without changing its semantics?

In the remainder of the paper, we introduce a methodology for this problem. For this purpose, an approach is proposed which iteratively considers invariants and creates corner case scenarios based on them. Using this, the designer is presented with options to substitute a considered invariant by adding a new or modifying an existing pre-condition. Alternatively, these substitutions can also be performed automatically.

B. General Idea

The general idea proposed in this paper for eliminating invariants is illustrated by means of the class diagram depicted in Fig. 2(a). Here, a simple traffic light preemption is specified. If the attribute carLight (pedLight) is assigned to



(a) First witness with a depth of 2

(b) Second witness with a depth of 4

Fig. 3. Witnesses for violation of oneRed (For the sake of a better readability, the button objects are omitted from the system states.)

True, cars (pedestrians) are allowed to go. Otherwise, they are supposed to wait. By the invariant in the class diagram (denoted by oneRed), it should be ensured that the traffic lights for both, cars and pedestrians, are never "green" at the same time. Finally, cars are allowed to pass as long as no pedestrian requests to cross the street, i.e. no pedestrian invokes the requesting operation. This is specified in the respective pre- and post-conditions of the operations. The initial state which is assumed for this example is depicted in Fig. 2(b).

The task is to eliminate the invariant oneRed. For this purpose, we propose the following scheme: First, the invariant oneRed is simply removed from the model. Then, existing methods for automatic model exploration (e.g. [8], [9]) are applied to obtain a sequence of operation calls which actually leads to a system state violating oneRed. The resulting sequence diagrams serve as witnesses pin-pointing the designer to invalid scenarios which are no longer excluded by the invariant. By means of these witnesses, the model can be adjusted.

For example, assume that a sequence as given in Fig. 3(a) is obtained. It witnesses a scenario which originally would be prevented by the invariant. By inspecting this scenario, it can be concluded that switching the pedLight to "green" while the carLight still is "green" should not be allowed. This can be expressed by adding the pre-condition

```
not( not pedLight and carLight )
```

to the operation switchPedLight().

Afterwards, this process is repeated in order to check if further scenarios, originally excluded by oneRed, are possible. This would lead to another witness as e.g. shown in Fig. 3(b). From that, the designer learns that the carLight should not be switched to "green" while the pedLight is still "green". This illegal behavior can be prevented by adding the pre-condition

```
not( pedLight and not carLight )
```

to the operation switchCarLight().

After this iteration, no further scenarios leading to a violation of oneRed can be generated. That is, the model has been adjusted in such a way that oneRed can be removed without affecting the semantics of the specification. Using this scheme, a structural methodology is available which pin-points the designer to scenarios prevented by an invariant to be removed. Based on that, corresponding changes can be conducted on the model. Moreover, adjustments can even be performed automatically. As a result, a model is obtained that does not rely on invariants any longer.

IV. Proposed Methodology

In this section, we describe the proposed methodology in detail. For this purpose, basic notations to formally express the concepts from Section II are provided first.

In the following, $\mathcal{I}$ describes the set of invariants in a model. Further, σ denotes a system state. For an invariant $i \in \mathcal{I}$, $\sigma(i)$ is true if and only if the invariant i is satisfied by the system state σ. Operation calls are denoted by $\omega_1, \ldots, \omega_j$ in the order they are invoked. They lead from an initial system state σ_0 to a sequence of following system states $\sigma_1, \ldots, \sigma_{j-1}, \sigma_j$. Note that operations are not called by classes, but their corresponding objects in the system state.

Given that, the proposed procedure can be formulated as follows:

Algorithm E (*Invariant Elimination*). This algorithm eliminates an invariant $i \in \mathcal{I}$ from a given UML/OCL model without changing the semantics of it.

E1. [Initialization.] Set $\mathcal{I} \leftarrow \mathcal{I} \setminus \{i\}$, i.e. remove an invariant i to be eliminated from the set of invariants.

E2. [Violate invariant.] Try to determine a scenario leading to a system state σ_j with $\sigma_j(\neg i) = 1$, i.e. to a system state which violates the currently considered invariant i.

E3. [Continue or terminate?] If such a σ_j cannot be determined, the invariant is already covered by other constraints. Thus, either continue with the next invariant in Step E1 or, if no further invariants are left, terminate.

E4. [Add pre-condition.] Inspect the determined scenario and add pre-conditions that avoid entering σ_j again. Afterwards, continue with Step E2.

In this methodology, Step E2 and Step E4 are crucial.

In Step E2, a system state has to be determined such that the invariant is violated. Therefore, we propose the usage of approaches for automatic model exploration. Enumerative or simulative approaches are often not suitable for this purpose since they soon reach their limits and can only consider very short sequences of operation calls. Interactive approaches such as introduced in [8] require further manual interaction by the designer and cannot always ensure completeness. Recent advancements in the dynamic verification of UML/OCL models [9] made the proposed methodology practically feasible. Here, formal methods are exploited which enable to efficiently determine sequence diagrams and system states of large sequences of operation calls while still considering the whole search space.

Step E4 is crucial since here the actual substitution of the invariant is performed. The system state σ_{j-1} as well as the operation call ω_j are particularly important. The operation call ω_j causes to reach the state σ_j which originally was prevented by the currently considered invariant i. Thus, in order to remain the semantics of the model, it has to be ensured that ω_j is not called in state σ_{j-1}. This can easily be done by adding a new pre-condition. Depending on the individual case, this step can be done automatically or interactive.

Automatic Approach. A formal and, therefore, automatable way to avoid that an operation is not called in a state σ_{j-1} is to add an inversion of σ_{j-1} to the pre-condition of the operation corresponding to ω_j. More precisely, all assignments to attributes, links, and operation parameters of σ_{j-1} are taken, conjugated, and finally inverted. Then, the pre-condition is the resulting expression. Since this often leads to very similar expressions, the pre-conditions can usually be optimized afterwards (e.g. by merging identical terms).

Interactive Approach. The interactive approach works almost the same as the automatic approach but additionally exploits the expertise as well as the design understanding of the user. Here, the scenarios as well as the resulting pre-conditions are just considered as a suggestion which (1) pin-points the designer to invalid scenarios originally prevented by

the invariant and (2) provides the designer with options. While in some cases these suggestion can simply be taken over, quite often the designer is able to derive much better constraints out of that.

Example 3: Consider again the class diagram given in Fig. 2(a). The pre-conditions suggested by the automatic approach would be similar to the ones deduced in Section III-B. However, they would further incorporate the value of the request attribute. With a better understanding of the design, one can derive that this attribute is not relevant in order to prevent invalid system states and remove that particular assignment from the suggested expressions. This leads to the pre-conditions given in Section III-B.

V. Case Study

The proposed methodology has been implemented in C++. In order to evaluate the approach, a UML model specifying the abstract functionality of a CPU, i.e. the communication between its different modules such as memory, program counter, and ALU, has been applied. The model was composed of 6 classes, 5 operations, 8 invariants, 41 pre-/post-conditions, and 9 instantiated objects.

For Step E2 of the proposed methodology, i.e. for determining a scenario leading to a system state which violates a currently considered invariant i, the dynamic verification approach introduced in [9] has been utilized. Therefore, the number j of operation calls have to be provided, to which we will refer to as *depth* in the following. Note that, if no scenario can be determined in Step E2, the depth has to be increased up to a reasonably large number where no further violations are expected. Only this ensures that a currently considered invariant is completely substituted by other constraints.

The results of the respective iterations performed by the proposed methodology are summarized in Table I. The model consists of eight invariants, which are to be removed. For the first invariant, a witness with only one operation call (i.e. with depth=1) has been determined first. From that, a pre-condition has been interactively added to the model, i.e. the pre-condition that was suggested by the approach has been modified (denoted by Fix=+1 Pre and Type=Interactive). Afterwards, no further witness could be determined considering one operation call only. Thus, the depth has been increased in order to check for further scenarios. This time, the automatically suggested pre-condition was suitable to be added without modification to the model. Since no further witnesses were found after 2 and after 100 operation calls (the previously defined upper bound), the invariant was classified to be entirely covered by the added constraints and, thus, removed from the model. In a similar way, the remaining invariants have been eliminated. Note that the second invariant has been substituted by a post-condition instead of a pre-condition. This was an individual decision based on the design knowledge of the user. Furthermore, the last two invariants already have been covered by the previously made modifications of the model.

The time needed to eliminate the invariants from the model indicates the efficiency of the methodology. Due to the help of the generated scenarios and the suggested options, the manual modifications of all considered case studies have been performed within a few minutes. The run-time of the approach itself was negligible (i.e. just a few seconds) in most of the cases. Only if larger depths are considered, the run-time might become crucial. The development of more advanced methods to address this is left for future work.

Overall, we can conclude that the proposed methodology is very helpful in the process of invariant elimination. Instead of considering *all* invariants of a given model in *each* single operation, we are able to entirely remove them from the model, while at the same time a significantly lower number of special and local constraints is added. While a naïve method, i.e. adding each invariant as a post-condition to each operation, would lead to $8 \cdot 5 = 40$ local constraints in case of the *CPU* benchmark, our approach generates just 14.

TABLE I
Case Study Results

Inv.	Depth	Fix	Type	Time	Inv.	Depth	Fix	Type	Time
I1	1	+1 Pre	Inter.	0.00	I5	1	+1 Pre	Inter.	0.00
	1	—		0.00		1	+1 Pre	Inter.	0.00
	2	+1 Pre	Auto.	0.00		1	+1	Inter.	0.00
	2	—		0.00		1	—		0.00
	100	—		11.80		100	—		4.59
I2	1	+1 Post	Inter.	0.00	I6	1	+1 Post	Inter.	0.00
	1	—		0.00		1	—		0.00
	2	+1 Post	Inter.	0.01		100	—		0.43
	2	—		0.00					
	100	—		19.60					
I3	1	+1 Pre	Inter.	0.00	I7	1	—		0.00
	1	+1 Pre	Inter.	0.00		100	—		0.43
	1	+1 Pre	Inter.	0.00					
	1	—		0.00					
	100	—		4.61					
I4	1	+1 Pre	Inter.	0.00	I8	1	—		0.00
	1	+1 Pre	Inter.	0.00		100	—		0.47
	1	+1 Pre	Inter.	0.00					
	1	—		0.01					
	100	—		4.65					

Finally, this approach further helps in model understanding as designers are directly pin-pointed to scenarios which are prevented by invariants.

VI. Conclusions

We presented an approach that assists designers in the elimination of invariants from a model. While invariants are a helpful concept during the modeling phase, they become cumbersome in the implementation phase. Our approach iteratively removes invariants from the model by substituting them with alternative local constraints, such as pre-conditions. The semantics of the model is thereby preserved. Depending on the situation, the designer can apply the suggested pre-conditions of the approach automatically or modify them in an interactive step.

In case studies we illustrated that our approach is able to reduce the number of additional constraints in comparison to naïve methods. Furthermore, the necessary exploration of the system states can be done in a reasonable run-time. In future work, we plan to improve the performance of the automatic approach, particularly in scenarios requiring larger depths. Furthermore, the quality of the automatically generated modifications is subject for further consideration.

Acknowledgments

This work was supported by the German Research Foundation (DFG) (DR 287/23-1).

References

[1] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*, Jan. 1999.
[2] G. Martin and W. Müller, *UML for SOC Design*. Springer, Jul. 2005.
[3] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., Feb. 2008.
[4] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML*. Addison-Wesley Longman, Mar. 1999.
[5] B. Meyer, J.-M. Nerson, and M. Matsuo, "EIFFEL: Object-Oriented Design for Software Engineering," in *European Software Engineering Conference*, ser. Lecture Notes in Computer Science, H. K. Nichols and D. Simpson, Eds., vol. 289, Sep. 1987, pp. 221–229.
[6] G. T. Leavens and Y. Cheon, "Design by Contract with JML," Sep. 2006, available at http://www.eecs.ucf.edu/ leavens/JML/jmldbc.pdf.
[7] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
[8] J. Tenzer and P. Stevens, "GUIDE: Games with UML for interactive design exploration," *Knowledge-Based Systems*, vol. 20, no. 7, pp. 652–670, Oct. 2007.
[9] M. Soeken, R. Wille, and R. Drechsler, "Verifying Dynamic Aspects of UML Models," in *Design, Automation and Test in Europe*, Mar. 2011, pp. 1077–1082.