Leveraging the Analysis for Invariant Independence in Formal System Models

Nils Przigoda[1] Robert Wille[1,2] Rolf Drechsler[1,2]

[1]*Group for Computer Architecture, University of Bremen, 28359 Bremen, Germany*
[2]*Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany*
{*przigoda,rwille,drechsle*}*@informatik.uni-bremen.de*

***Abstract*—Formal models, based on modeling languages such as UML in combination with constraint languages such as OCL, allow for an abstract description of a system prior to its implementation. But since the resulting models are often rather complex, redundancies in terms of model descriptions which can directly be implied from already existing constraints can easily arise. In particular, OCL invariants are affected by this. In order to efficiently detect those, methodologies for the analysis for invariant dependence have recently been proposed. However, they have severe limitations with respect to scalability, automation, and quality of the determined results. In this work, we aim for leveraging the analysis for invariant independence in formal systems models by addressing these drawbacks. For this purpose, a new methodology based on the exploitation of powerful solving engines as well as a complete analysis scheme is proposed. Experimental evaluations confirm that the proposed solution is significantly faster and leads to a much better quality of the results.**

***Keywords*-UML/OCL Models, Invariant Independence, Model Analysis**

I. Introduction

With increasing complexity of today's electronic systems, researchers started to investigate the integration of modeling languages in the design of hardware systems such as embedded systems [1], [2]. As an example, in the context of hardware/software co-design, systems are specified first on a high level of abstraction, before they are being partitioned into respective hardware- and software-components in a later step. Modeling languages, such as the *Unified Modeling Language* (UML [3]) as one of the best-known representatives, received much attention in this regard.

UML allows for the specification of *formal models*, i. e., a detailed description of a system at a high level of abstraction before precise implementation steps are performed. For this purpose, UML provides appropriate models which hide precise implementation details while being expressive enough to specify a complex system. Within UML, the *Object Constraint Language* (OCL [4]) enables the enrichment of the respective models by textual constraints which adds further information to the description. The usage of OCL makes it possible to define *invariants* which restrict valid system states or describe further properties as well as relations between the specified components.

The resulting models are often designed by large teams and are composed of various components including classes, relations, and constraints, e. g., given by invariants. This usually leads to non-trivial descriptions where redundancies, i. e., model descriptions which can directly be implied from already existing constraints, can easily arise. Those redundancies are often not obvious to the designer but significantly hinder the further design process as they keep the amount of description means to be considered unnecessarily large. In particular, invariants are affected by this.

As a consequence, researchers and engineers started to investigate how corresponding *dependent invariants* can efficiently be identified or how the *independence* of invariants can efficiently be confirmed. Since performing those checks is a cumbersome task, these efforts eventually lead to the consideration of methodologies for the *analysis for invariant independence*. First accomplishments in this regard have already been achieved: Corresponding solutions have been proposed and evaluated in [5], [6] based on the *UML-based Specification Environment* (USE, see [7], [8]) and in [9] based on the theorem prover *Isabelle/HOL* (see [10], [11]). However, these approaches – which, to the best of our knowledge, represent the state-of-the-art in independence analysis – have severe limitations with respect to scalability, automation, and quality of the determined results (this is discussed in more detail later in Section IV).

In this work, we aim for leveraging the analysis for invariant independence in formal systems models by addressing the drawbacks of previously proposed approaches. For this purpose, a fully automatic approach is proposed which efficiently determines either the independence of an invariant or determines the reasons for its dependency. To tackle the underlying complexity, powerful solving engines such as *SMT solvers* (e. g., Z3 [12]) are exploited. By exhaustively analyzing all possible reasons in a clever fashion, the quality of the obtained results is significantly improved. In fact, our approach eventually delivers *all minimal reasons* for all dependencies and, by this, significantly supports the designer in removing the corresponding redundancies.

The advantages of the proposed solution compared to previous work has been confirmed in an experimental evaluation. It is shown that independence analysis is performed significantly faster and with better results. While previously proposed approaches often determine no explicit reasons for a dependency, our solution guarantees them in a minimal

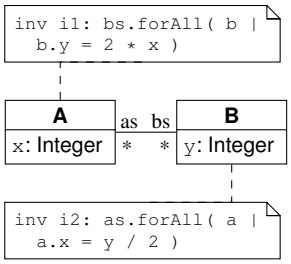
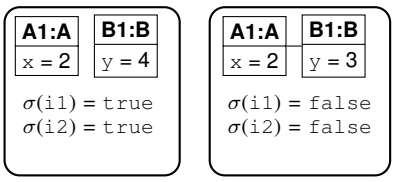


Figure 1: A simple model



(a) Valid (b) Invalid

Figure 2: System states

fashion. As a result, analysis for invariant independence is leveraged with respect to automation, scalability, and quality.

The remainder of this work is structured as follows. The following section provides the basics and notations used in this work. The problem of independence in formal models is introduced in Section III, while a straight-forward approach for that including a discussion of related work is reviewed in Section IV. Afterwards, our solution is described in detail in Section V and results of our evaluation are summarized in Section VI. Section VII concludes the paper.

II. Preliminaries

In order to keep the paper self-contained, this section provides a brief review on UML/OCL and introduces the notation used to describe the respective models and system states.

Definition 1 (Model): A *model* $\mathcal{M} = (\mathcal{C}, \mathcal{R})$ represented in terms of a *class diagram* is a tuple of *classes* $\mathcal{C}$ and *relations* $\mathcal{R}$ (also known as *associations*). A *class* $c \in \mathcal{C}$ may contain attributes and operations. A relation $r \in \mathcal{R}$ describes the connection between two classes $c_1, c_2 \in \mathcal{C}$. Besides that, the model can additionally be enriched by textual constraints which can be provided in OCL. Such textual constraints are called *invariants*. The set of all OCL invariants of a model is denoted by $\mathcal{I}$. Each invariant $i \in \mathcal{I}$ is associated to a class $c \in \mathcal{C}$.

Example 1: Figure 1 shows a model $\mathcal{M}$ composed of two classes $A, B \in \mathcal{C}$ which are connected by a relation $r \in \mathcal{R}$. Invariants `i1` and `i2` restrict the values with which attributes of the classes can be assigned.

Models represent a blueprint for possible instantiations of a system. Formally, instantiations can be represented in terms of *system states* which, depending on the invariants, might be valid or not.

Definition 2 (System states): Let $\mathcal{M} = (\mathcal{C}, \mathcal{R})$ be a model with invariants $\mathcal{I}$. An instantiation of $\mathcal{M}$ is called a *system state*, i. e., for each class $c \in \mathcal{C}$ a corresponding number of objects is derived which altogether satisfy the relations $\mathcal{R}$. A single system state is denoted by σ, while the set of all possible system states is denoted by Σ[1].

A system state satisfies a given invariant $i \in \mathcal{I}$, if the respective expression is satisfied for all objects derived from the associated class $c \in \mathcal{C}$. This is evaluated through a function f defined by:

$$f : \Sigma \times \mathcal{I} \to \mathbb{B}$$
$$(\sigma, i) \mapsto \begin{cases} \texttt{true}, & \text{if } i \text{ holds for } \sigma \\ \texttt{false}, & \text{else} \end{cases}$$

For a fixed state $\sigma \in \Sigma$, the shorthand notation $\sigma(i)$ is used instead of $f(\sigma, i)$. A system state is called *valid* if it satisfies all invariants; otherwise is is called *invalid*.

Example 2: Consider again the model in Figure 1. Two possible system states derived from this model are shown in Figure 2: a valid system state satisfying all invariants is provided in Figure 2a; an invalid system state violating both invariants, `i1` and `i2`, is provided in Figure 2b.

III. Independence in Formal Models

During the design of complex systems, the clear and precise definition of constraints and requirements is crucial. Description means as reviewed in the previous section, in particular invariants, are particularly suited for this purpose. In order to preserve the design understanding, engineers aim to keep the respective specifications indeed complete and comprehensive but, at the same time, as compact as possible. But with increasing complexity of the considered design as well as a rising number of involved engineers, often specifications result which are more complex than necessary.

In fact, invariants are frequently introduced which are supposed to add new constraints and/or requirements but actually are already covered by previously added invariants. This redundancy, which often is not obvious to designers, significantly hinders the further design process as it keeps the amount of invariants to be considered in the following design steps unnecessarily large. Motivated by that, the detection of *dependent invariants* (or the confirmation that all invariants are *independent*) has become an important task in early steps of today's design flows.

[1]Note, that we assume a fix number of instantiations and attribute assignments, i. e., the set of all possible system states is bounded. This is a reasonable assumption considering that at least for the concrete implementation of the considered model, finite bounds are applied anyway.

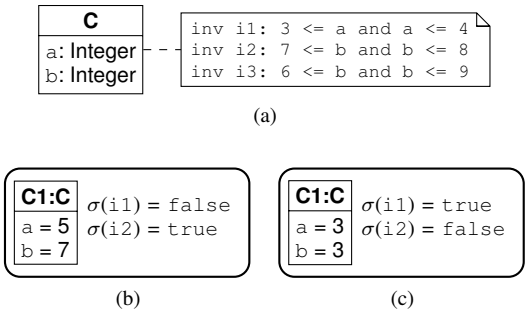


Figure 3: (In)Dependencies in formal models

Definition 3 (Independence): Let $\mathcal{M}$ be a model with invariants $\mathcal{I}$ and Σ the set of all system states derived from $\mathcal{M}$. Then an invariant i_k is called *independent* iff

$$\exists \sigma \in \Sigma : \bigwedge_{i \in \mathcal{I} \setminus \{i_k\}} \sigma(i) \wedge \neg\sigma(i_k). \tag{1}$$

Vice versa, an invariant i_k is called *dependent* iff Eq. (1) does not hold.

In other words, Definition 3 states that an invariant is independent iff it further restricts the set of valid system states compared to $\mathcal{I} \setminus \{i_k\}$, i. e., that at least one system state which was valid under invariants $\mathcal{I} \setminus \{i_k\}$ is not valid anymore under invariant i_k.

Example 3: Consider the model given in Figure 3a composed of a single class with two integer attributes `a` and `b`. The set of valid system states is restricted by invariants `i1`, `i2`, and `i3`. However, invariant `i3` does not further restrict the set of valid system states compared to `i1` and `i2`, since all system states valid under `i1` and `i2` are trivially valid under `i3` as well. Hence, `i3` can be discarded. The remaining two invariants are independent as each of them further restricts the set of valid system states. As an example, Figure 3b shows a system state valid under `i2`, but not `i1`, while Figure 3c shows a system state valid under `i1`, but not `i2`.

Detecting dependent invariants (or showing the independence of invariants) may significantly simplify the design process. Invariants proven to be dependent can be discarded from the model or, at least, do not have to be considered intensely. On the contrary, if all invariants have been proven independent, designers know that each and every constraint indeed restricts the desired design behavior further and shall be considered accordingly. However, analyzing whether a given model is only composed of independent invariants (and if not, determining which invariants are dependent) is a non-trivial task. Consequently, (automatic) methods are applied for this purpose.

Algorithm 1 Analysis for invariant independence

Input: A model $\mathcal{M} = (\mathcal{C}, \mathcal{R})$
Input: A set $\mathcal{I}$ of invariants
1: $D \leftarrow \emptyset$ // set of detected dependencies
2: **for all** $i_k \in \mathcal{I}$ **do**
3: **if** $\exists\ \sigma \in \Sigma : \bigwedge_{i \in \mathcal{I} \setminus \{i_k\}} \sigma(i) \wedge (\neg\sigma(i_k))$ **then**
4: // i_k is independent
5: **else**
6: // i is dependent
7: $D \leftarrow D \cup \{i_k\}$
8: **return** D;

IV. Analysis for Invariant Independence

A straight-forward and simple approach for the analysis of invariant independence is provided by Algorithm 1. The algorithm expects a model $\mathcal{M}$ together with the corresponding set of invariants $\mathcal{I}$ to be investigated. All dependent invariants determined during the analysis are stored in a set D which is returned by the algorithm.

First, it is assumed that the model does not include dependent invariants, i. e., D is initialized with an empty set (line 1). Then, all invariants i_k in $\mathcal{I}$ are considered separately (lines 2–7). For each i_k, it is checked whether i_k is independent or not. For this purpose, Definition 3 is applied, i. e., it is checked whether there is a system state which is valid under invariants $\mathcal{I} \setminus \{i_k\}$, but violates i_k (line 3). The actual checks can be conducted by approaches for the verification and validation of formal models e. g., as introduced in [7], [13], [14], [15], [16]. These methods generate valid system states for a given model including a set of invariants and can easily be utilized for the purpose considered here. If such a system state was determined, it has been proven that i_k is independent (line 4). In contrast, if no such system state was obtained, the invariant i_k is considered as dependent and, hence, added to the set D (lines 5–7).

Example 4: Consider again the model shown in Figure 3a. As discussed in Example 3, corresponding system states showing the independence can be found for `i1` and `i2` (see Figure 3b and Figure 3c, respectively). In contrast, no system state satisfying `i1` and `i2` as well as violating `i3` can be found. Hence, `i3` is considered as dependent.

Approaches for the analysis of invariant independence as sketched by Algorithm 1 have already been considered in previous work [5], [6], [9]. However, these approaches have severe limitations with respect to scalability, automation, and quality of the determined results.

More precisely, [5], [6] relies on an enumerate approach for system state generation, i. e., when performing the step in line 3 of Algorithm 1, all possible states are considered one after another. Designers may improve this enumeration by additionally providing programs written in *A Snapshot Sequence Language* (ASSL, see [8]) which allows to define

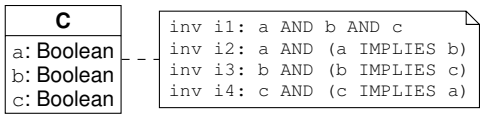


Figure 4: A model containing dependencies

in which fashion the traversal through the search space is performed. Besides the fact that this requires a significant amount of additional manual work, the entire search space has to be enumerated in the worst case anyway.

In [9], the corresponding problem including the entire description of the model to be analyzed is translated into formal semantics for Isabelle/HOL. Then, corresponding theorem provers are applied to determine the respective system states from which either dependence or independence of an invariant can be concluded. While this approach is very powerful with respect generic conclusions (results might be obtained which are universal and e. g., do not rely on bounds for object instantiation), it requires advanced expert knowledge and heavily relies on manual interaction.

Besides that, strictly following Algorithm 1 also leads to rather poor results as illustrated by the following example.

Example 5: Consider the model given in Figure 4. Using Algorithm 1, it is observed that *all* invariants are dependent. Consequently, $D = \{\texttt{i1}, \texttt{i2}, \texttt{i3}, \texttt{i4}\}$ will be returned by the algorithm. While this result indeed is correct (in fact, each invariant could be covered by one or more of the other invariants), precise reasons are not provided. Since simply discarding all invariants obviously is not an option, the designer has to manually check the dependencies in order to remove redundancy in the invariants.

While obviously the example above is rather artificial, cases like this frequently occur in the design of complex systems. The approach in [6] tries to deal with that by first applying Algorithm 1 and, afterwards, trying to identify the reason for it. This is done by considering cases again, for which no system state could be obtained. Then, one invariant after another is *deactivated* until a plausible reason has been obtained (cf. [6, Sect. 2]). An alternative relies on analyzing the set of dependent invariants in detail, but without considering the full set of remaining invariants [6, Sect. 3]. In the example above, this would lead to no result. As a consequence, no reasons for the dependencies would be detected and the designer would probably try to discard `i1` although discarding `i2`, `i3`, and `i4` would not only remove the redundancy, but eventually result in a much more compact model. The approach proposed in [9] makes a more comprehensive analysis but also here neither completeness nor minimality of the determined results are ensured.

Overall, previous approaches do not only suffer from limitations with respect to scalability and automation but, in the worst case, also lead to results where the designer is left with a rather poor understanding about the identified dependencies. For a comprehensive and fast removal of redundancies, an independence analysis is required which (1) does provide a *complete* set of dependent invariants together with (2) a *minimal* list of the *reason* for each dependency.

V. Proposed Solution

In this work, we aim for leveraging the analysis for invariant independence in formal system models by addressing the drawbacks of previously proposed approaches discussed above. We introduce an *automated* approach which does not rely on any manual interaction at all. *Scalability* is improved by utilizing efficient solving engines rather than enumerative methods. Finally, the *quality of the results* is improved by performing a more advanced analysis of the respectively considered invariants which eventually leads to the *complete* determination of dependencies together with their respective *minimal* reasons. In this section, details of the proposed solution are provided. For this purpose, an advanced problem formulation for independence analysis is provided first. Afterwards, the improved algorithm is presented.

A. Advanced Problem Formulation

In order to address the drawbacks mentioned above, the original problem formulation for independence analysis (see Definition 3) is enriched by a definition of a *minimal reason for a dependency*.

Definition 4 (Minimal Reason for a Dependency): Let $\mathcal{M}$ be a model with invariants $\mathcal{I}$ and Σ being the set of all system states derived from $\mathcal{M}$. Furthermore, let $i_k \in \mathcal{I}$ be a dependent invariant according to Definition 3. Then, the *reason for this dependency* is a subset $I \subseteq \mathcal{I} \setminus \{i_k\}$ which satisfies

$$\forall\, \sigma \in \Sigma : \bigwedge_{i \in I} \sigma(i) \Rightarrow \sigma(i_k). \tag{2}$$

Furthermore, a reason I is called *minimal*, iff

$$\forall\, J \subsetneq I : \exists\, \sigma \in \Sigma : \bigwedge_{j \in J} \sigma(j) \wedge \neg\sigma(i_k) \tag{3}$$

also holds.

In other words, Definition 4 states that a subset $I \subset \mathcal{I} \setminus \{i_k\}$ of invariants is a reason, if for all system states $\sigma \in \Sigma$ the satisfaction of the invariants I also implies the satisfaction of the invariant i_k. Minimality is guaranteed by the fact that removing just one invariant from a reason I (leading to $J \subsetneq I$) would allow for at least one system state σ where Eq. (2) does not hold (for J instead of I). For such a system state, all the invariants of a subset J are satisfied while i_k is violated.

Algorithm 2 Analysis for dependency of i_k

Global Var.: A set R of minimal reasons (initialized $R \leftarrow \emptyset$)

analyze_dependency(I)
Input: A subset I of invariants
1: $minimal\_reason \leftarrow true$
2: **for all** $J \subset I$ with $|J| = |I| - 1$ **do**
3: **if** $\exists\ \sigma \in \Sigma : \bigwedge_{j \in J} \sigma(j) \wedge (\neg\sigma(i_k))$ **then**
4: // J is not a reason
5: **else**
6: // J is a reason smaller than I, i.e. I is not minimal
7: $minimal\_reason \leftarrow false$
8: analyze_dependency(J)
9: **if** $minimal\_reason$ **then**
10: $R \leftarrow R \cup \{I\}$
11: **return**

Example 6: Consider again, the model given in Figure 4. Instead of providing a simple set $D = \{\texttt{i1, i2, i3, i4}\}$ of dependent invariants as discussed in Example 5, applying the advanced definition additionally provides some evidently minimal reasons, e. g.,

- `i2` is dependent on `i1` (i. e., `i1` $\Rightarrow$ `i2`),
- `i3` is dependent on `i1` (i. e., `i1` $\Rightarrow$ `i3`), and
- `i4` is dependent on `i1` (i. e., `i1` $\Rightarrow$ `i4`).

Obviously, considering these minimal reasons helps the designer much better in removing redundancies from the model. Based on that, it is clear that `i2,i3,i4` can be removed.

B. Determining Minimal Reasons for Dependent Invariants

In order to additionally determine the minimal reasons for a dependent invariant i_k, a more thorough analysis is required. For this purpose, an extension to Algorithm 1 for independence analysis is proposed: Whenever an invariant $i_k \in \mathcal{I}$ has been shown dependent (line 7 in Algorithm 1), another analysis of this particular dependency is performed. To this end, a (recursive) procedure as sketched in Algorithm 2 is proposed.

The algorithm gets the subset of invariants $I = \mathcal{I} \setminus \{i_k\}$ as input and determines all minimal reasons for the dependent invariant i_k[2]. These reasons are stored in a *global* variable R which is initialized as an empty set. Since i_k was shown to be dependent (using Algorithm 1), $I = \mathcal{I} \setminus \{i_k\}$ obviously is a reason for that. It is additionally assumed that I is minimal; stored in a Boolean variable *minimal_reason* (line 1). Then, it needs to be analyzed whether smaller reasons exist or not.

For this purpose, all subsets $J \subset I$ which are one element smaller than I are considered (line 2). For each of these subsets, it is checked whether it still is a reason for the dependent invariant i_k (line 3), i. e., whether Eq. 2 from Definition 4 holds. As in Algorithm 1, approaches for the verification and validation of formal models e. g., as introduced in [7], [13], [14], [15], [16] can be utilized for this purpose. This requires a slight re-formulation of the condition: Instead of showing that J is a reason (i. e., the satisfaction of invariants J implies the satisfaction of invariant i_k in *all* system states), it is determined whether J is not a reason (i. e., whether a system state *exists* which satisfies J but not i_k).

If J is indeed not a reason, then the assumption that I is minimal remains valid, i. e., the iteration through all subsets simply continues (line 4). Otherwise, a reason with a smaller number of invariants than I has been determined. Consequently, *minimal_reason* is immediately set to *false* (line 7) and the algorithm is recursively called to analyze further subsets smaller than J (line 8). The process terminates if all subsets have been considered. If no smaller reason has been determined, i. e., *minimal_reason* remained *true* (line 9), the currently considered set I of invariants has proven to be minimal and is accordingly added to R (line 10).

Algorithm 2 completely solves the problem formulated in the previous section. However, a major bottleneck is the check in line 3 which is computationally expensive with respect to both the complexity of each single check but also the quantity of all checks to be conducted. This is addressed as follows:

- *Complexity*
  As discussed in Section IV, previously proposed approaches for the analysis of independence relied on enumerative methods [5], [6] or theorem provers [9] with a significant amount of manual interaction.
  We propose to tackle the complexity by relying on powerful solving engines such as *SMT solvers* (e. g., Z3 [12]). For this purpose, the respective verification problems (is there a system state satisfying or not satisfying certain invariants) are translated into corresponding propositional formulas as described in [16]. Afterwards, an SMT solver is applied to solve the problem. Since they do not enumerately traverse the search space but employ intelligent decision heuristics, powerful learning schemes, and efficient implication methods, they handle the complexity much faster while, at the same time, require no manual interaction.
- *Quantity*
  Strictly following Algorithm 2 might lead to a large number of checks to be conducted – many of those might be redundant. This is avoided by keeping track of previously performed checks: The results of all checks conducted in line 3 are stored in a corresponding global data-structure. If the same subset of invariants is considered again, simply the previously determined result is re-used.

[2]Note, that more than one minimal reason may exist for a dependent invariant i_k.

Table I: Evaluation of the runtime performance

Model name	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{I}\rvert$	$\lvert\mathcal{O}\rvert$	Independent?	[6]	This work
CAB	1	0	3	1	×	< 1 s	< 1 s
class C2	1	0	4	1	×	< 1 s	< 1 s
class C3	1	0	4	1	×	< 1 s	< 1 s
class C4	1	0	9	1	×	< 1 s	3 s
CivStat	1	1	6	4	×	987 s	4 s
				5	×	>12 h	3 s
				6	×	>12 h	7 s
				7	×	>12 h	17 s
				8	×	>12 h	37 s
				9	×	>12 h	80 s
				10	×	>12 h	217 s
Demo1	3	3	4	5	✓	>12 h	3 s
Demo2	3	3	7	5	✓	>12 h	5 s
CarRental	9	12	6	5	×	>12 h	< 1 s
				6	×	>12 h	2 s
				7	×	>12 h	3 s
				8	×	>12 h	5 s
				9	×	>12 h	8 s
				10	×	>12 h	16 s
PerCom	3	4	5	5	✓	>12 h	6 s
Simple-CPU	6	6	9	5	✓	>12 h	9 s

Legend:
$\lvert\mathcal{C}\rvert$: Number of classes $\lvert\mathcal{R}\rvert$: Number of relations $\lvert\mathcal{I}\rvert$: Number of invariants $\lvert\mathcal{O}\rvert$: Maximal number of considered object instantiations per class
Independent?: All invariants in the model are independent (✓) or not (×) [6]/This work: Runtime of the respective approach

Due to the fact that, for a subset J — which has been classified not to be a reason — the determined system state also offers the information that all subsets of J are not a reason as well, this data-structure allows for an efficient and fast processing of the algorithm. By this, the total number of actually performed checks can be reduced significantly.

Following these schemes, the proposed solution addressed the main drawbacks of state-of-the-art approaches for invariant independence analysis, i. e., scalability, automation, and quality of the determined results. This has also been confirmed in an experimental evaluation whose results are summarized next.

VI. Experimental Evaluation

In order to evaluate the proposed solution, the algorithms described above including its optimizations have been implemented in Xtend as an *Eclipse* plugin. *Z3* [12] has been utilized as SMT solver for the respective checks in line 3 of Algorithm 1 and line 3 of Algorithm 2. As benchmarks, we applied UML/OCL models taken from the *USE* package [7][3]. All experiments have been carried out on an Intel i5 with 2.6 GHz cores and 16 GB memory using a 3.11 kernel Linux.

A. Evaluation of the Runtime Performance

In a first series of experiments, we evaluated the performance of the proposed solution with respect to the required runtime. Since we aimed for a fully-automatic approach for independence analysis, we did not compare our approach to the solution proposed in [9] (which heavily relies on manual interaction and, hence, is not comparable) but to the solution proposed in [6].

Table I summarizes the obtained results. The first columns provide the name of the respective models followed by their number of classes ($\lvert\mathcal{C}\rvert$), their number of relations ($\lvert\mathcal{R}\rvert$), their number of invariants ($\lvert\mathcal{I}\rvert$), their maximal number of considered object instantiations per class ($\lvert\mathcal{O}\rvert$), and an indication whether the invariants in the model are independent (denoted by ✓) or not (denoted by ×) under the respective configuration (Indep?). Note, that the models *CivStat* and *CarRental* have been considered with more than one configuration (i. e., number of object instantiations) and, hence, corresponding results are reported in several rows in Table I. Afterwards, the respective runtime (in CPU-seconds) needed for performing the independence analysis is reported for the approach from [6] as well as the approach presented in this work.

The numbers clearly show that the proposed solution is an improvement compared to state-of-the-art method from [6]. For all models, an independence analysis is performed significantly faster. For most of the considered cases, the previously proposed approach was not able to determine a result within the given time limit of 12 hours while our solution completed all the analyses in just a couple of minutes. At the same time, the proposed approach was even able to determine much better results which is discussed next.

[3]Including models considered in previous work [6] in addition to further models not considered before.

B. Quality of the Results

In a second series of experiments, we evaluated the quality of the obtained results – again including a comparison to the solution from [6]. For this purpose, the results obtained for the five models for which the approach presented in [6] terminated within the given time limit have been investigated in more detail.

Table II provides a corresponding summary, i. e., lists the results of the respective analyses. The right-hand side of the respective terms represent the respectively determined dependent invariants of the model, while the left-hand side represent their reasons. Note, that in the proposed solution all reasons have been shown to be minimal. This is not the case for results obtained by [6]. In cases where [6] did not determine a reason, all remaining invariants (i. e., $\mathcal{I} \setminus \{i_k\}$) have been considered as a reason. In case of the model *class C4*, we omitted an explicit listing of all determined results. Instead only the total number of determined reasons are presented.

Also here, very clear conclusions can be drawn. While the previously proposed approach determines all dependent invariants for the models *CAB*, *class C2*, *class C3*, and *class C4*, no explicit reasons are provided in these cases. As discussed before in Section IV, this may lead to rather poor design decisions, e. g., the removal of invariants which better should remain in the model. In contrast, the proposed approach leads to all minimal reasons from which, as discussed in Example 6 (see Section V-A), redundancy can be removed in a significantly better fashion. Moreover, in case of *CivStat* it can even be observed that the previously proposed solution does not necessarily detect all dependent invariants. In fact, only invariant `i6` was detected as dependent; but invariants `i3` and `i4` are dependent as well in this case.

Table II: Quality of the results

Model name	[6]	This work
CAB	$i_3 \Rightarrow i_1$	$i_3 \Rightarrow i_1$
class C2	$\mathcal{I} \setminus \{i_1\} \Rightarrow i_1$ $\mathcal{I} \setminus \{i_2\} \Rightarrow i_2$ $\mathcal{I} \setminus \{i_3\} \Rightarrow i_3$ $\mathcal{I} \setminus \{i_4\} \Rightarrow i_4$	$i_4 \Rightarrow i_1$ $i_4 \Rightarrow i_2$ $i_4 \Rightarrow i_3$ $i_1, i_2, i_3 \Rightarrow i_4$
class C3	$\mathcal{I} \setminus \{i_1\} \Rightarrow i_1$ $\mathcal{I} \setminus \{i_2\} \Rightarrow i_2$ $\mathcal{I} \setminus \{i_3\} \Rightarrow i_3$ $\mathcal{I} \setminus \{i_4\} \Rightarrow i_4$	$i_2, i_3 \Rightarrow i_1$ $i_3, i_4 \Rightarrow i_1$ $i_2, i_4 \Rightarrow i_1$ $i_3, i_4 \Rightarrow i_2$ $i_1 \Rightarrow i_2$ $i_2, i_4 \Rightarrow i_3$ $i_1 \Rightarrow i_3$ $i_2, i_3 \Rightarrow i_4$ $i_1 \Rightarrow i_4$
class C4	# 9	# 96
CivStat	$i_3, i_4 \Rightarrow i_6$	$i_3, i_4 \Rightarrow i_6$ $i_1, i_4, i_6 \Rightarrow i_3$ $i_1, i_3, i_6 \Rightarrow i_4$ $i_2, i_3, i_6 \Rightarrow i_4$

The right-hand side of the respective terms represent the respectively determined dependent invariants of the model, while the left-hand side represent their reasons. For *class C4*, only the total number of dependencies are listed.

VII. Conclusion

In this work, we leveraged the analysis for invariant independence in formal system models by addressing the main drawbacks of previously proposed solutions: scalability, automation, and quality of the determined results. For this purpose, a new methodology based on a complete analysis scheme as well as based on the exploitation of powerful solving engines has been proposed. The experimental evaluation showed that this enabled an *automatic* analysis of independence within seconds or minutes, while previously proposed solutions were not able to determine a result within hours. At the same time, the quality of the results is improved. For the first time, minimal reasons for dependencies have been determined, while previous approaches often provide no or only incomplete and non-minimal reasons.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001, the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1 and a research project under grant no. WI 3401/5-1, the Graduate School SyDe funded by the German Excellence Initiative within the University of Bremen's institutional strategy as well as the Siemens AG.

References

[1] Y. Vanderperren, W. Müller, and W. Dehaene, "UML for electronic systems design: a comprehensive overview," *Design Automation for Embedded Systems*, vol. 12, no. 4, pp. 261–292, 2008.

[2] G. Martin and W. Müller, *UML for SOC Design*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[3] J. Rumbaugh, I. Jacobson, and G. Booch, Eds., *The Unified Modeling Language reference manual*. Essex, UK: Addison-Wesley Longman Ltd., 1999.

[4] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *TAP*, ser. Lecture Notes in Computer Science, C. Dubois, Ed., vol. 5668. Springer, 2009, pp. 90–104.

[6] M. Gogolla, L. Hamann, and M. Kuhlmann, "Proving and visualizing OCL invariant independence by automatically generated test cases," in *TAP*, ser. Lecture Notes in Computer Science, G. Fraser and A. Gargantini, Eds., vol. 6143. Springer, 2010, pp. 38–54.

[7] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, 2007.

[8] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *Software and System Modeling*, vol. 4, no. 4, pp. 386–398, 2005.

[9] A. D. Brucker and B. Wolff, "Semantics, calculi, and analysis for object-oriented specifications," *Acta Inf.*, vol. 46, no. 4, pp. 255–284, 2009.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[11] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. New York, NY, USA: Cambridge University Press, 1993.

[12] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[13] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *ICST Workshops*. IEEE Computer Society, 2008, pp. 73–80.

[14] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation," in *MoDELS*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer, 2007, pp. 436–450.

[15] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS*, ser. Lecture Notes in Computer Science, O. Grumberg and M. Huth, Eds., vol. 4424. Springer, 2007, pp. 632–647.

[16] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*. IEEE Computer Society, 2010, pp. 1341–1344.