Slicing Models for Equiconsistency with Alloy

Marc Thieme[1], Shobhit Singh[1(✉)], Terru Stübinger[1], Romain Pascual[2], and Mattias Ulbrich[1]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
[2] MICS, CentraleSupélec, Université Paris-Saclay, Gif-sur-Yvette, France
Corresponding author: shobhit.singh@kit.edu

Abstract. Model-driven development enables collaborative design across heterogeneous modelling domains, but it also raises the risk of inconsistent models. We study the problem of extracting minimal submodels that preserve cross-domain consistency. Concretely, given two sets of models related by a consistency specification and a model from the first set, we seek an equiconsistent slice, that is, a submodel that preserves the same consistency relationships with models of the second set. Since the definition of equiconsistency quantifies over the complete second set, a direct computation is infeasible (and undecidable in general). Thus, we formulate slicing as a declarative synthesis problem and solve it using counterexample-guided inductive synthesis (CEGIS). The procedure iteratively proposes candidate slices and refines them using counterexamples (models obtained as violation witnesses if a candidate is not a valid slice). Iterating the CEGIS loop converges to equiconsistent, minimal slices within the bounds used by the model finder. We instantiate the abstract equiconsistency slice problem using attributed typed graphs as models, express consistency relations declaratively, and further realize it using relational logic and SAT-based solving. We then obtain an automated synthesis of equiconsistent slices based only on the consistency relation. We evaluate the method on a synthetic dataset to compare three CEGIS implementations: an explicit loop in Alloy, an explicit loop in Alloy*, and a quantified encoding in Alloy*. We highlight their practical trade-offs.

Keywords: Model-Driven Engineering · Slicing · Consistency · Alloy.

1 Introduction

Model-driven development [3, 13] relies on a collection of models, that encodes domain-specific information about the system under development. Since these models describe overlapping aspects of the same system, they must satisfy cross-model consistency relations [27]. As systems become more complex, models grow larger, and consistency checking becomes more expensive. Still, large parts of the models may be irrelevant for a given consistency specification [26].

Model slicing can mitigate this complexity. Originating from static program analysis [6, 7, 20, 30, 32], slicing improves comprehensibility, performance, and debugging. For models, slicing means extracting a submodel that preserves a given

slicing criterion. Model slicing can speed up analyses [26] and help developers focus on the aspects of a model relevant for a given purpose [4]. Still, most existing model slicing work targets behavioral or structural slicing criteria [4, 8, 19].

Inter-model consistency is inherently *relational*: whether a model is consistent is not intrinsic but depends on the other models used in the system description. When accounting for model evolution, i.e., when some models change due to edits through the design process, consistency then needs to be considered with respect to all possible counterpart models. For instance, removing elements of a model might introduce or eliminate consistency partners, making any slicing that considers only the current set of models unsound for consistency analysis.

We address this issue by introducing *equiconsistency slicing*, building on the notion of equiconsistency from [12, 23]. Given a consistency relation between two sets of models $\mathcal{M}_1$ and $\mathcal{M}_2$, two models $m, n \in \mathcal{M}_1$ are equiconsistent if they are consistent with exactly the same models in $\mathcal{M}_2$. An equiconsistent slice preserves this property. Intuitively, an equiconsistency slice retains precisely the information required to determine inter-model consistency. For instance, a consistency condition between two circuit diagrams of different components of a system that share a common databus is that no two elements ever write onto the bus at the same time. To analyze this consistency, all components in the circuit diagrams that have no impact on the contents of the bus can be safely removed without altering the consistency status.

Equiconsistency slicing reduces model sizes while preserving consistency with the related models describing the system under development. It enables analysis of smaller representations without altering consistency and exposes which parts of a model actually influence it. Therefore, slices can help support scalable analysis and the understanding or debugging of consistency relations by isolating the elements that matter semantically.

Computing an equiconsistent slice for $m \in \mathcal{M}_1$ is challenging because it involves universal quantification over the counterpart models in $\mathcal{M}_2$. We formulate the problem as a declarative synthesis task and solve it using the counterexample-guided inductive synthesis (CEGIS) approach [1, 17]. Candidate slices are iteratively refined by using the counterexample generated while trying to prove equiconsistency. Once a valid slice has been obtained, the whole procedure is iterated and converges to a minimal equiconsistent slice.

Practically, we represent models as attributed typed graphs [11], which we encode in Alloy [16]. Adding a description of the consistency relations in Alloy's specification language then provides a bounded implementation, relying on SAT-solving as a backend routine in the CEGIS loop. Within the given scope, the analyzer either finds a counterexample or establishes equiconsistency.

Contributions. This paper contributes (1) a formal notion of equiconsistent model slicing, (2) a CEGIS-based procedure ensuring minimality of the generated slice, (3) an Alloy implementation demonstrating practical feasibility, and (4) an empirical comparison of explicit Alloy, explicit Alloy*, and internal Alloy* implementations, done on synthetic model instances to assess scalability.

2 Related Work

Program Slicing and Semantic Preservation. Program slicing, introduced by Weiser [32], extracts program fragments preserving a variable's value at a given program point. Classical methods use syntactic dependencies, such as control and data dependence graphs [15]. Later work generalized slicing to semantic criteria, e.g., conditioned [10], assertion-based [5], and abstract slicing based on abstract interpretation [14]. In these approaches, the slice preserves satisfaction of a specification rather than execution traces. Our work follows this semantic view but preserves a relational property: consistency with models of another domain.

Model Slicing. Slicing has been transferred from programs to models, including state-based systems [18], UML models [2, 19], and heterogeneous megamodels [25]. Most approaches are dependency-based: a set of model elements defines the criterion, and dependencies determine the slice [4]. Incremental techniques exploit edit histories or model evolution assumptions [24, 28]. In contrast, our slicing criterion is not element-based but property-based. The slice is defined solely by preservation of a logical relation.

Consistency-Preserving Slicing. Constraint-oriented slicing has been studied for UML/OCL models, where slices preserve satisfaction of constraints within a model [26]. Such approaches reason about a single model and its predicates. We instead consider relations between models of different metamodels and preserve the entire set of consistent partner models (equiconsistency [12, 23]).

Synthesis-Based Slicing. Some approaches separate candidate generation from verification and view slicing as a search problem guided by counterexamples [6]. We adopt this perspective and formulate slicing as a synthesis problem solved using counterexample-guided inductive synthesis (CEGIS) [1, 17], where counterexamples are inconsistent partner models.

Positioning. Existing slicing techniques preserve behavior or predicate satisfaction within a single artifact. We instead introduce relational semantic slicing: the preserved property is a model relation, and slices are synthesized rather than derived from dependency analysis.

3 Equiconsistency Slicing

Model-driven development relies on multiple interrelated models describing different aspects of a system. Each model is an instance of a *metamodel* that determines the admissible structures. Examples include well-formed UML class diagrams, Simulink models, Event-B machines, or compilable Java programs. Following [23], we identify a metamodel with the set of its valid instances.

When multiple models describe the same system, they may create contradictions that hinder its realizability. In that sense, consistency denotes the absence of such contradictions and, thus, appears as a weak form of correctness, representing the notion of "joint realizability" [9]. Following [23], we consider consistency as a relation between models: either they are, or they are not consistent.

Formally, we consider two metamodels $\mathcal{M}_1$ and $\mathcal{M}_2$, together with a *consistency relation* $CR \subseteq \mathcal{M}_1 \times \mathcal{M}_2$, where $CR(m_1, m_2)$ means that m_1 and m_2 are consistent. This abstract description is enough to state the problem addressed in this paper, and will be refined in Sect. 5 for the practical implementation.

3.1 Model Slicing

In practice, large models often contain information irrelevant to cross-model consistency. Engineers, therefore, wish to remove unnecessary parts while preserving the model's consistent behavior. A solution is to slice the model and remove the irrelevant parts. Intuitively, a *slice* of a model is a smaller model that preserves a given property of the original one, called the *slicing criterion* [19]. Model slicing techniques can be broadly classified into two categories. *Syntactic slicing*, originating in program analysis, constructs slices by computing dependency relations between elements and taking their transitive closure, as introduced by Weiser [32]. In contrast, *declarative slicing* defines slices implicitly via a preservation criterion where candidate submodels are validated against a property specification. In this paper, we adopt a declarative slicing approach, where slices are characterized by equiconsistency rather than explicitly computed dependency structures. We define model slicing using a structural reduction relation and an equivalence relation as the slicing criterion.

Definition 1 (Model Slicing). *Let $\mathcal{M}$ be a metamodel, $\preceq$ a partial order on $\mathcal{M}$ and S an equivalence relation on $\mathcal{M}$. A model $m' \in \mathcal{M}$ is a slice of $m \in \mathcal{M}$ with respect to S if $m' \preceq m$ and $S(m', m)$. Additionally, the model is a proper slice if $m' \prec m$, where $\prec$ is the strict partial order induced by $\preceq$.*

In this paper, the order $\preceq$ corresponds to the submodel relation, which will be made explicit in Sect. 5. The remaining question is: Which property must be preserved such that a slice is similar to the original model consistency-wise?

3.2 Equiconsistency

The relevant property for consistency-based slicing is *equiconsistency* [12], which captures the idea that two models are indistinguishable with respect to consistency against all models of the second metamodel.

Definition 2 (Equiconsistency – [12, Def. 2]). *Two models m_1 and m_1' in $\mathcal{M}_1$ are equiconsistent for CR, written $m_1 \sim_1^{CR} m_1'$, if*

$$\forall m_2 \in \mathcal{M}_2,\ CR(m_1, m_2) \iff CR(m_1', m_2)\ . \tag{1}$$

Consequently, a given consistency relation CR induces two equiconsistency relations $\sim_1^{CR} \subseteq \mathcal{M}_1 \times \mathcal{M}_1$ and $\sim_2^{CR} \subseteq \mathcal{M}_2 \times \mathcal{M}_2$, both being equivalence relations on their associated metamodels [12]. In this paper, we will consider equiconsistency on $\mathcal{M}_1$ and make it explicit only when it might be ambiguous. Since CR is assumed fixed, we will then write $\sim$ instead of $\sim_1^{CR}$. Since equiconsistency means

that no model of $\mathcal{M}_2$ can distinguish m_1 and m_1' with respect to consistency, it is the natural candidate for a slicing criterion, which leads us first to consider the following problem:

Equiconsistency Slicing Problem (ESP). Given two metamodels $\mathcal{M}_1$ and $\mathcal{M}_2$, a consistency relation $CR \subseteq \mathcal{M}_1 \times \mathcal{M}_2$, and a model $m_1 \in \mathcal{M}_1$, find an equiconsistent proper slice m_1' of m_1, i.e., find $m_1' \in \mathcal{M}_1$ such that $m_1' \prec m_1$ and $m_1' \sim m_1$.

Finding an equiconsistent slice means finding a model m_1' that is a strict submodel of m_1 and is equiconsistent with m_1. Among all slices of a model, we are particularly interested in *minimal* ones, which leads to the following problem:

Minimal Equiconsistency Slicing Problem (MESP). Given two metamodels $\mathcal{M}_1$ and $\mathcal{M}_2$, a consistency relation $CR \subseteq \mathcal{M}_1 \times \mathcal{M}_2$, and a model $m_1 \in \mathcal{M}_1$, find $m_1' \preceq m_1$ such that $m_1' \sim m_1$ and for all $m_1'' \prec m_1'$, $m_1'' \not\sim m_1$.

Note that we relax the strictness condition in the definition of slice to allow for the possibility that the original model is already minimal, i.e., relaxing the proper slice condition $m_1' \prec m_1$ to $m_1' \preceq m_1$. In that sense, the existence of at least one minimal equiconsistent slice is guaranteed.

3.3 Undecidability of Equiconsistency Slicing

Equiconsistency is a strong property that requires checking consistency against all models of the second metamodel. This universal quantification makes equiconsistency undecidable in general, which hinders the feasibility of equiconsistency slicing. Consider the following decision problem:

Equiconsistency Problem (EP). Given two metamodels $\mathcal{M}_1$ and $\mathcal{M}_2$, a consistency relation $CR \subseteq \mathcal{M}_1 \times \mathcal{M}_2$, and two models $m_1, m_1' \in \mathcal{M}_1$, does $m_1 \sim m_1'$ hold?

Theorem 1 (Undecidability of Equiconsistency). *EP is undecidable.*

Proof (Sketch). Consider an alphabet Σ and $\mathcal{T}, \mathcal{T}'$ two Turing machines over Σ. We can consider the instance of the equiconsistency problem given by

- $\mathcal{M}_1 = \{\mathcal{T}, \mathcal{T}'\}$, $\mathcal{M}_2 = \Sigma^*$,
- $CR \subseteq \mathcal{M}_1 \times \mathcal{M}_2$ such that for any $t \in \mathcal{M}_1$ and $w \in \mathcal{M}_2$, $CR(t, w)$ iff $w \in L(t)$, where $L(t)$ is the language accepted by the Turing machine t,
- $m_1 = \mathcal{T}$, and $m_1' = \mathcal{T}'$.

Equiconsistency in this case becomes the equality of the languages accepted by the Turing machines, which is undecidable by Rice's theorem.

Essentially, the undecidability of equiconsistency comes from the undecidability of membership in CR. The consequence is that any practical algorithm for equiconsistency slicing must rely on restrictions or approximations.

3.4 Finite Approximation

To obtain a computable approximation of equiconsistency, we restrict the set of models of $\mathcal{M}_2$ used to check consistency. This restriction leads to a notion of constrained equiconsistency over a (finite) subset E of $\mathcal{M}_2$. In the implementation, E will correspond to the bounded exploration space of models (see Sect. 5).

Definition 3 (Constrained Equiconsistency). *Let $E \subseteq \mathcal{M}_2$. Two models m_1 and m_1' in $\mathcal{M}_1$ are equiconsistent for CR with respect to E, written $m_1 \sim_E m_1'$, if for all $e \in E$, $CR(m_1, e) \iff CR(m_1', e)$.*

Proposition 1. *The constrained equiconsistency relation $\sim_E$ is an equivalence relation on $\mathcal{M}_1$.*

Constrained equiconsistency is an over-approximation of full equiconsistency, which we can iteratively refine by adding more models of $\mathcal{M}_2$ to the set E. The idea is to add counterexamples that distinguish non-equiconsistent models, which we can find by checking the consistency of the candidate slice against models of $\mathcal{M}_2$. Formally, a counterexample for non-equiconsistent models $m_1, m_1' \in \mathcal{M}_1$ is a model $m_2 \in \mathcal{M}_2$ such that $CR(m_1, m_2) \not\iff CR(m_1', m_2)$. Full equiconsistency holds if and only if no counterexample exists, which naturally suggests an iterative refinement process: start with some finite set E, search for a candidate slice, and enlarge E whenever a counterexample is found. This idea serves as the basis for the CEGIS procedure introduced in the next section.

3.5 Motivating Example

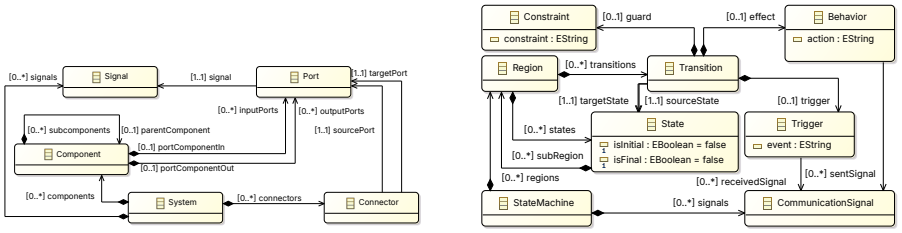
We illustrate equiconsistency slicing on a simplified excerpt of an automotive body comfort system described by Lity et al. [21]. The system is described using two modeling viewpoints: an architectural model capturing hardware structure and a behavioral model describing control logic.

Metamodels and Models. The architectural metamodel *BCSComponents* (Fig. 1a) represents components that communicate via input and output ports. The behavioral metamodel *StateMachine* (Fig. 1b) describes controller logic using regions and transitions labeled by trigger and effect signals. Figures 2 and 3 show example instances of both metamodels. Intuitively, the architecture specifies which signals exist in the system, while the state machine specifies how those signals are produced and consumed.

Consistency Relation. Both models describe the same communication interface. Informally, the architecture determines which signals must appear in the behavior. More precisely, the models are consistent if:

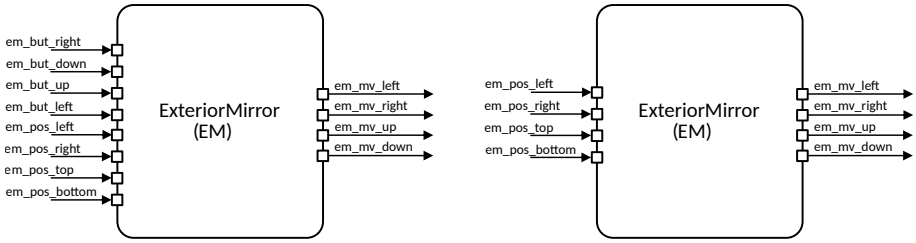
- every input port named x is realized by a transition receiving signal x,
- every output port named x is realized by a transition emitting signal x.

Thus, consistency depends only on the presence of signal names, and not on the state machine's detailed control structure. For instance, the models of Figs. 2a and 3 are consistent, while the models of Figs. 2b and 3 are not, since some signals from the state machine do not appear as a port in the component diagram.



(a) Architecture Metamodel (b) Behavioral Metamodel

Fig. 1: Metamodel: (a) Component diagram, (b) State machine.



(a) Consistent with the model in Fig. 3 (b) Inconsistent with the model in Fig. 3

Fig. 2: Two instances for the metamodel from Fig. 1a (adapted from [21]).

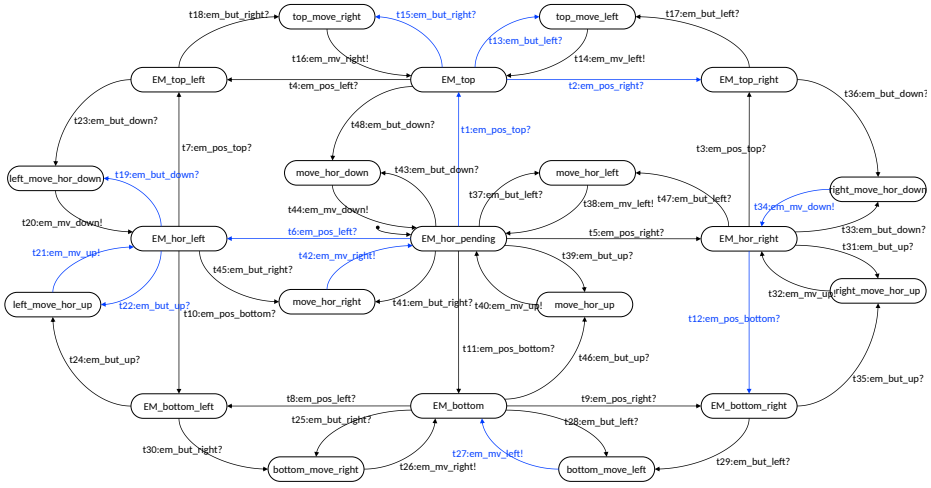


Fig. 3: State machine instance (adapted from [21]).

Slicing Problem. Consider the state machine in Fig. 3. Many transitions manipulate signals that appear elsewhere. Removing such redundant transitions does not change whether the behavioral model satisfies the architectural requirements. However, not all reductions are valid. An *equiconsistent slice* therefore preserves exactly the same consistency relationships as the original model: it keeps all information relevant to cross-model consistency, while discarding irrelevant behavioral detail. For instance, it should remain consistent with the model of Fig. 2a and inconsistent with the model of Fig. 2b. The part highlighted in blue in Fig. 3 shows an equiconsistent slice. The resulting model contains only the transitions necessary to witness the required signals; states and redundant occurrences disappear because they do not influence consistency. In this example, we deliberately relaxed the metamodel so that the multiplicity constraints in Fig. 1b are ignored. Depending on the use case, where the metamodel itself may be fixed and not subject to change, the approach allows also encoding these multiplicities to enforce the original metamodel. The goal of this paper is to compute slices automatically from the consistency relation alone, without designing a dedicated slicer for each modeling language or constraint set.

4 Approach: Equiconsistency Slicing via CEGIS

In our declarative approach to equiconsistency slicing, the solution is split into two steps that are iterated alternately: generating a candidate slice and checking its equiconsistency with the initial model. This check requires assessing consistency for *all* possible models of the second metamodel. The universal quantification (see eq. (1)) makes the problem undecidable in general (Thm. 1) and also practically infeasible even for finite metamodels, as the number of possible instances grows exponentially with the bound on model size. We propose to use *Counterexample-Guided Inductive Synthesis (CEGIS)* to iteratively approximate this universal condition with a growing set of counterexamples. Intuitively, this means that checking the equiconsistency of the candidate slice is replaced by searching for a model (in the second metamodel) that violates equiconsistency. Each found counterexample refines the constraints imposed on future candidates. When no counterexample can be found within the bounds, the candidate is considered equiconsistent.

4.1 Oracles

The procedure relies on three decision problems:

1. *Counterexample search:* Given a model m and a slice s, check if there exists a model e such that $CR(m, e) \iff CR(s, e)$. If such a model exists, it is returned as a counterexample.
2. *Counterexample classification:* Given a counterexample e, find if it is consistent with the original model m. This partitions the counterexamples into $E^+ = \{e \mid CR(m, e)\}$ and $E^- = \{e \mid \neg CR(m, e)\}$.

Algorithm 1 Equiconsistency slicing via CEGIS

```
1: procedure EquiconsistentSlice(m)
2:     E⁺ ← ∅, E⁻ ← ∅, s ← m
3:     loop
4:         e ← FindCounterexample(m, s)
5:         if e = ⊥ then return s
6:         if Consistent(m, e) then E⁺ ← E⁺ ∪ e
7:         else E⁻ ← E⁻ ∪ e
8:         s ← SynthesizeSlice(m, E⁺, E⁻)
```

3. *Slice synthesis:* Given E^+ and E^-, compute a slice $s \prec m$ such that

$$(\forall e \in E^+,\ CR(s,e)) \ \wedge\ (\forall e \in E^-,\ \neg CR(s,e)). \tag{2}$$

The slice s is constrained equiconsistent to m with respect to $E^+ \cup E^-$ (Def. 3).

4.2 CEGIS Procedure

The algorithm (Alg. 1) builds the sets E^+ and E^- incrementally: they are both initially empty (meaning any submodel of m is a valid candidate), and each iteration then refines the set of admissible candidates. The algorithm terminates when the checker no longer finds a counterexample, indicating that the slice satisfies the bounded specification.

Partitioning the collected counterexamples into E^+ and E^- effectively caches the consistency status $CR(m,e)$ for each counterexample $e \in E$, and thus relieves the synthesis step from repeatedly referring to the original model m. In practice, this simplifies the synthesis constraints and reduces solver effort. The alternative formulation to (2) would be to require that $CR(m,e) \Leftrightarrow CR(s,e)$ for every $e \in E$.

The CEGIS loop computes an equiconsistent slice, but does not inherently guarantee minimality. To obtain minimal slices, we iteratively restart the procedure using the previously found slice as the new input model. The accumulated counterexamples are retained across runs, allowing the search to progressively shrink the model while avoiding the rediscovery of similar counterexamples. The process stops when no strictly smaller equiconsistent slice can be found.

4.3 Models as Attributed Typed Graphs

The CEGIS procedure is independent of a concrete modeling language. We only assume that models conform to metamodels and admit a notion of submodel and consistency. To make these notions precise, we represent metamodels and model instances as *attributed typed graphs* (ATGs).

A metamodel is given by a finite set of node types, edge types, and attribute types, together with typing functions that describe the admissible sources and targets. An instance model is a finite graph whose nodes and edges are typed by the metamodel and whose elements may carry attribute values. We can then

consider the submodel relation as the partial order $\preceq$, such that s is a *submodel* of a model m if it is obtained by removing nodes, edges, and attribute values from m while preserving typing and incidence. Intuitively, a slice is a structure-preserving restriction of the original model.

The consistency relation CR may be defined by graph constraints, logical predicates, or model transformations. Equiconsistency only assumes that consistency can be decided on concrete model pairs. That is, consistency remains abstract, and the algorithm assumes only the availability of procedures for finding counterexamples, checking the consistency of model pairs, and synthesizing submodels under logical constraints. The following section explains how the required decision procedures are realized in practice for ATGs using Alloy.

5 Implementation

We implemented the proposed CEGIS procedure from Alg. 1 in Alloy [16] and Kodkod [31] for the bounded relational model finder. The three oracles introduced in Sect. 4 (counterexample search, classification, and slice synthesis) are each encoded as satisfiability problems over finite relational structures. This section describes the finite relational representation used to decide the semantic conditions of Sect. 4. The encoding preserves the definitions within a bounded universe, allowing the slicing problem to be reduced to satisfiability.

5.1 Relational Encoding of the Slicing Problem

Encoding Metamodels and Models. Attributed typed graphs are interpreted as typed relational structures: each node and edge type becomes an Alloy signature, while source and target mappings are binary relations over them. Instead of the standard additional elements to encode attribution for nodes and edges [11], we represent the association of attribute values as a relation from model elements to values. To ensure decidability of the bounded analysis, attribute domains are interpreted over a finite abstraction domain. Concretely, we use a shared bounded integer universe for all attributes, which yields a finite relational structure while preserving the relational formulation of slicing. A concrete model instance is then encoded using singleton extensions of metamodel signatures for its model elements. Thus, each model corresponds to a fixed relational interpretation of the metamodel signature.

Encoding Slices as Substructures. The structural partial order $\preceq$ used for slicing is the submodel relation, which becomes the subgraph relation for attributed typed graphs (see Sect. 4.3). In the relational setting, it is enforced by inclusion constraints between interpretations. Every relation encoding the slice is required to be a restriction of the relation in the original model. Slice synthesis is then achieved as the search for a sub-interpretation satisfying a semantic preservation condition (the equiconsistency condition) rather than the removal of syntactically dependent elements.

Encoding the Consistency Relation. The consistency relation CR is represented as a logical predicate over relational structures. It is given intentionally, meaning that consistency holds when the relational constraint is satisfied, and the solver can reason directly over candidate models rather than over an explicit enumeration of consistent pairs. In practice, Alloy's relational operators are used to express CR. Thus, checking the consistency of two models boils down to solving a satisfiability problem over the combined relational structure rather than explicitly enumerating consistent partner models. As a result, counterexamples produced during CEGIS are concrete models that explicitly violate equiconsistency within the bounded universe.

5.2 Realizing the CEGIS Oracles

Each oracle in the CEGIS loop becomes a bounded model-finding task, and we use Alloy as the relational solver. It translates relational constraints into SAT and relies on off-the-shelf solvers, providing a concise specification language with an efficient search backend. The size of the counterexamples must be bounded to ensure that the program terminates. In practice, we bound the size proportionally to the original model, and the absence of counterexamples is guaranteed only within the chosen scope. Thus, bounding the search does not guarantee global convergence as a consequence of Thm. 1.

5.3 Execution Strategies

We implemented three variants of the slicing procedure, differing in how the CEGIS loop and the equiconsistency condition are realized. Two of them employ Alloy* [22], an extension of Alloy 4 that, unlike Alloy, can solve constraints with $\exists\forall$ quantifier alternations. It internally employs a CEGIS strategy like the one outlined in Alg. 1, but does not partition counterexamples explicitly into E^+ and E^-. Instead, it invokes the SAT solver incrementally, allowing it to preserve knowledge about previously observed counterexamples as learned clauses in the solver state.

1. *Manual CEGIS*: The first implementation follows the procedure of Alg. 1 closely. Separate solver calls perform equiconsistency check, counterexample classification, and slice synthesis, all encoded in Alloy. The main program orchestrates the CEGIS loop and maintains the sets E^+ and E^-, allowing reuse of the counterexamples across iterations.
2. *Iterative Star*: The second implementation uses Alloy* to encode equiconsistency using a solver-level $\exists\forall$ quantification. The slice is synthesized by solving $\exists s \prec m,\ \forall e,\ CR(m,e) \iff CR(s,e)$ within the chosen bounds. While equiconsistency is internalized, minimality is still obtained procedurally by invoking the slicer repeatedly.
3. *Full Star*: The third implementation extends the previous one by adding an explicit minimality constraint on the slice using an additional $\exists\forall$ quantification. We obtain a single Alloy* specification capturing both equiconsistency and minimality.

The second and third variants require the construction of a finite superstructure containing all admissible model elements within the chosen bounds to allow quantifying over all potential counterexamples. The superstructure ensures the Alloy* analyzer ranges over the intended search space rather than constructing witnesses that trivially satisfy the formula. Note that for the second variant, the drawback is that counterexamples cannot be reused across iterations, as they remain internal to the analyzer.

5.4 Code Generation

We have prototypically implemented an automatic translation that accepts ATGs (given as JSON files) and generates the corresponding Alloy declarations: (1) the metamodel declarations, (2) the model instance declarations, (3) the fixed model instances, (4) the slice declarations, and (5) declarations allowing the CEGIS loop to add counterexamples in the Manual approach. The consistency relation is to be stated as an Alloy predicate directly. Fig. 5 contains the result of this translation for an example.

6 Case Study

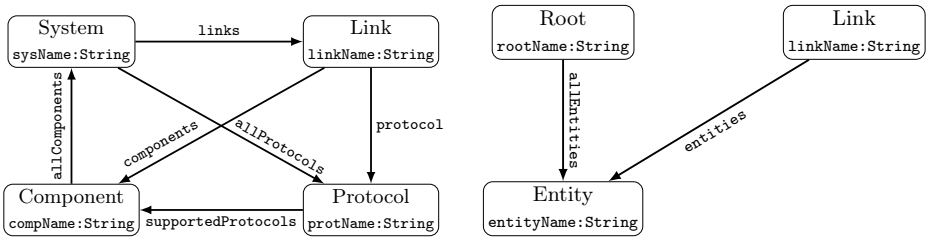
To showcase the approach and principles of the Alloy encoding, demonstrate the feasibility of finding equiconsistent slices, and evaluate and compare the performance of the three approaches outlined in Sect. 5.3, we conducted a case study on small- to medium-sized models. The study uses two metamodels, for which we generated 72 synthetic concrete model instances, which were subjected to the implementations of the three slicing approaches. The test-case models were systematically synthesized for different sizes, different numbers of cycles, and different edge densities. The artifacts for reproducing the analysis are available as supplementary material [29] on Zenodo.

Fig. 4 depicts the two metamodels $\mathcal{M}_1$ and $\mathcal{M}_2$ used in the case study. They are inspired by structural modeling languages but are synthetic in nature. Models in $\mathcal{M}_1$ describe the structure of system composition, including the communication protocols. Models of $\mathcal{M}_2$ express a more abstract view on the compositional nature of the modeled system with fewer details and connections.

The consistency relation CR_{study} used for equiconsistency slicing in this case study requires a correspondence between model elements and associations in the two models. Concretely, a pair $(m_1, m_2) \in \mathcal{M}_1 \times \mathcal{M}_2$ is consistent (i.e. in CR_{study}) when

- for every `System` in m_1 there is an equally named `Root` in m_2 and vice versa;
- for every `Component` in m_1 there is an equally named `Entity` in m_2 and vice versa;
- for every `Link` in m_1 there is an equally named `Link` in m_2 and vice versa.

Moreover, if two model elements are associated in m_1 via edges in `allComponents` or `components`, then the equally named model elements in m_2 must be associated via edges in `allEntities` or `entities`, and vice versa.



(a) Metamodel $\mathcal{M}_1$. (b) Metamodel $\mathcal{M}_2$.

Fig. 4: Metamodels used in the empirical evaluation.

The study is designed such that some elements of the metamodel are irrelevant for consistency. The presence of `Protocol` nodes in m_1 does not influence its consistency with any $m_2 \in \mathcal{M}_2$. Hence, minimal equiconsistent slices for m_1 produced by the slicing implementations omit `Protocol` nodes, all unnamed nodes, and all edges except those in `allComponents` and `components`.

6.1 Excerpts of the Study Encoded in Alloy

Fig. 5 shows excerpts from the generated Full Alloy* specification. Metamodels $\mathcal{M}_1$ and $\mathcal{M}_2$ are encoded as abstract signatures. The model instance to be sliced is encoded by listing its model elements as singleton signatures. Attribute relations are defined by enumerating all attribute values. The excerpt shows a model instance representing an unnamed system with two components named 0 and 1. The figure also shows the consistency relation CR_{study} encoded as an Alloy predicate. Identifiers suffixed `_m1` and `_m2` refer to the predicate's parameter list (omitted for space reasons). The predicate first introduces abbreviations and then checks relational inclusion and equality. For example, the relations `sysName` (resp. `rootName`) capture the mapping from systems (resp. roots) to names, and `sys2root` relates equally named system/root pairs. The first checked clause ensures that every named system in `sysName` has an equally named root in `rootName`.

6.2 Results

To evaluate and compare the scalability of the different implementations, we generated a test bench of 72 problem instances, with the configuration of the sliced model varying in size (10–15/30–45 nodes), density (low/high), and cyclicity (acyclic/cyclic). For each configuration, three random instances were added to the test bench. Experiments were conducted on an AMD EPYC 7742 CPU and 1 TiB of RAM, running Ubuntu 24.04 (kernel 6.17.0-14-generic), with multiple experiments running in parallel. The SAT solver used was SAT4J shipped with the Alloy Analyzer version 6. Alloy* is a fork of Alloy 4.2. The Java Virtual Machine was assigned a maximum heap size of 100 GB.

```
// Signatures for metamodels M1 and M2
abstract sig Link_1, Protocol_1, System_1, Component_1 {}
abstract sig links_1 { source: System_1, target: Link_1 }
abstract sig allProtocols_1 { source: System_1, target: Protocol_1 }
abstract sig allComponents_1 { source: Component_1, target: System_1 }
abstract sig components_1 { source: Link_1, target: Component_1 }
abstract sig protocol_1 { source: Link_1, target: Protocol_1 }
abstract sig supportedProtocols_1 { source: Protocol_1, target: Component_1 }
sig linkName_Source_1 in Link_1{linkName_1: set{v: Int | -8 < v ∧ v < 8}}
sig protName_Source_1 in Protocol_1{protName_1: set{v: Int | -8 < v ∧ v < 8}}
sig sysName_Source_1 in System_1{sysName_1: set{v: Int | -8 < v ∧ v < 8}}
sig compName_Source_1 in Component_1{compName_1: set{v: Int | -8 < v ∧ v < 8}}

abstract sig Link_2, Entity_2, Root_2 {}
abstract sig allEntities_2 { source: Root_2, target: Entity_2 }
abstract sig entities_2 { source: Link_2, target: Entity_2 }
sig linkName_Source_2 in Link_2{linkName_2: set{v: Int | -8 < x ∧ x < 8}}
sig entityName_Source_2 in Entity_2{entityName_2: set{v: Int| -8 < x ∧ x < 8}}
sig rootName_Source_2 in Root_2{rootName_2: set{v: Int | -8 < x ∧ x < 8}}
// Sample of the singleton signatures for the original model instance
one sig S1 extends System_1 {}
one sig C1, C2 extends Component_1 {}
one sig a0 extends allComponents_1 {} { source = S1 ∧ target = C1 }
one sig a0 extends allComponents_1 {} { source = S1 ∧ target = C2 }
fact { compName_1 = C1->0 + C2->1 }
...
// The signatures for the slice
sig Link_S in Link_1 {}
sig Protocol_S in Protocol_1 {}
sig System_S in System_1 {}
sig Component_S in Component_1 {}
sig links_S in links_1 {} {
    source in System_S ∧ target in Link_S }
sig allProtocols_S in allProtocols_1 {} {
    source in System_S ∧ target in Protocol_S }
sig allComponents_S in allComponents_1 {} {
    source in Component_S ∧ target in System_S }
sig components_S in components_1 {} {
        source in Link_S ∧ target in Component_S }
sig protocol_S in protocol_1 {} {
    source in Link_S ∧ target in Protocol_S }
sig supportedProtocols_S in supportedProtocols_1 {} {
        source in Protocol_S ∧ target in Component_S }
sig linkName_Source_S in linkName_Source_1 { linkName_S: set linkName_1 }
sig protName_Source_S in protName_Source_1 { protName_S: set protName_1 }
sig sysName_Source_S in sysName_Source_1 { sysName_S: set sysName_1 }
sig compName_Source_S in compName_Source_1 { compName_S: set compName_1 }

pred consistent [ /* params for model 1 (..._m1) and model 2 (..._m2) */ ] {
let allComponents = ((~source).(allComponents_m1 <: target)),
    allEntities = ((~source).(allEntities_m2 <: target)),
    entities = ~((~source).(entities_m2 <: target)),
    sys2root = sysName_m1.~rootName_m2,
    components2entity = compName_m1.~entityName_m2,
    link2link = linkName_m1.~linkName_m2
    {   sysName_m1 in sys2root.rootName_m2
        rootName_m2 in ~sys2root.sysName_m1
        compName_m1 in components2entity.entityName_m2
        entityName_m2 in ~components2entity.compName_m1
        linkName_m1 in link2link.linkName_m2
        linkName_m2 in ~link2link.linkName_m1
components2entity.entities=(components2entity.univ<:components_m1).link2link
sys2root.allEntities = (sys2root.univ <: ~allComponents).components2entity }}
// Not shown: fill the M2 signatures with nodes & edges for a complete graph
run{ all Link_m2: set Link_2, Entity_m2: set Entity_2, Root_m2: set Root_2,
    // Edges must be incident to nodes that actually belong to the graph
    allEntities_m2: set Root_m2.(~source:>allEntities_2) & target).Entity_m2,
   .., Link_m2 : Link_2 -> { x: Int | -7 ≤ x and x ≤ 7 },... |
consistent[/* signatures for slice (..._S) and signatures for m2 */]
iff consistent[/* signatures for original model (..._1) and for m2 */]
} for 0 but 4 Int
```

Fig. 5: Alloy excerpts encoding the case study using the Full Star CEGIS procedure.

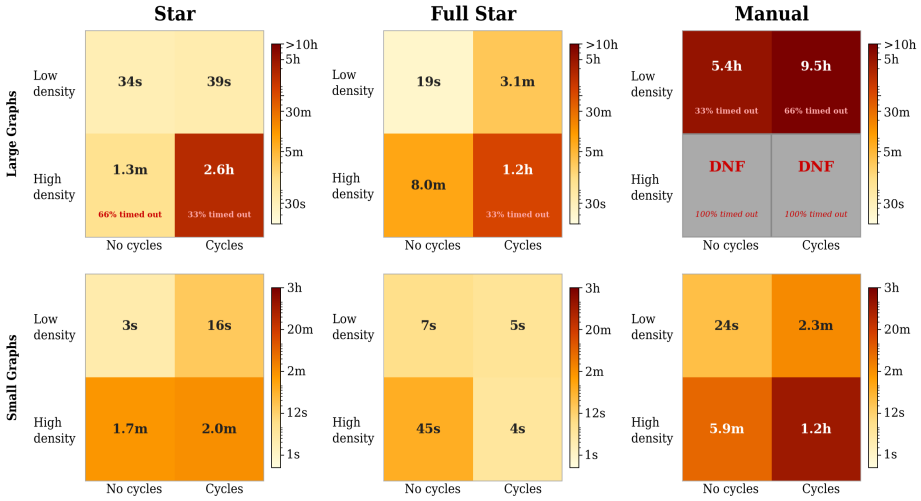


Fig. 6: Runtime Heatmaps: Effects of Density and Cycles (average of completed runs only, DNF is "did not finish within 10h")

Fig. 6 compares the three implementations. Manual CEGIS is consistently the slowest: on large, low-density acyclic graphs, it exceeds 5 hours, while Iterative Star and Full Star finish in 32 s and 19 s, respectively. None of the large high-density instances terminate within the time bound for Manual CEGIS.

Density has the strongest impact on runtime (Fig. 6). On small graphs, high density multiplies runtime by 6× (Full Star), 38× (Iterative Star), and 15× (Manual CEGIS). Cycles have a comparatively moderate influence. On large graphs, cycles increase Full Star runtime by about 10×, while Iterative Star changes only marginally (1.2×). The combination of high density and cycles causes timeouts for Manual CEGIS on all large instances. Under these conditions, Iterative Star also times out in most cases, and only Full Star remains reliable.

From small to large graphs (acyclic, low density), runtime increases by about 3× for Full Star, 12× for Iterative Star, and nearly three orders of magnitude for Manual CEGIS (24 s to over 5 h). This suggests that explicit orchestration of the synthesis loop scales poorly compared to analyzer-internal quantification. Iterative Star is the most memory-efficient implementation. On large graphs without cycles and low density, it peaks at 1.7 GB on average, compared to 5.3 GB for Full Star and 46 GB for Manual CEGIS. Manual CEGIS execution is dominated by CNF construction (about 90% of runtime), generating 100–800× more clauses than Full Star (up to 266 million), explaining runtime and memory behavior.

We also applied our approach to the motivating example from Fig. 3, an extremely sparse graph comprising 95 nodes and 226 edges. On this instance, the Manual procedure performed best and completed in 36 minutes (of which 35 minutes were spent in the SAT solver) after two iterations. Iterative Star timed out, and Full Star produced a spuriously empty result, likely because of

an internal out-of-memory error in the solver. One reason for this difference may be that in this example the slice was very small (all nodes of the state chart were to be dropped), and Manual seems to favour more aggressive reductions, resulting in fewer procedure iterations (3.1 on average vs 7.1 for Star).

6.3 Discussion and Limitations

Encoding of Attributed Typed Graphs. The implementation uses a relational encoding for SAT solving rather than mirroring the theoretical ATG formulation. Attributions are represented as relations, and data values belong to bounded integer intervals. This design keeps the encoding compact and allows uniform reasoning over structure and data. The trade-off is that extending the prototype to richer ATG variants (e.g., complex data domains) would require adapting the encoding rather than directly reusing the current representation.
Performance limitation. Our implementation demonstrates feasibility, but the runtime remains too high for interactive use, partly due to the repeated analyzer calls. Nevertheless, the approach remains useful for generating meaningful test cases (representative slices or counterexamples) to support the verification and validation of the consistency relation itself, or of additional tools that act on it.
Encoding constraints. Relational encoding has a significant impact on performance. In particular, deeper signature hierarchies increase compilation time during the translation to SAT, and larger universes degrade solving time. We mitigated this by encoding attribute values over a shared value domain, reducing the number of atoms introduced into the analyzer. These observations suggest that analyzer-oriented modeling decisions have a stronger influence on performance than the abstract algorithmic structure of the slicing procedure.
Constructing the quantified set for Alloy.* Alloy* evaluates quantified formulae over a fixed universe, requiring a superstructure of all potential counterexamples. By comparison, Alloy uses bounded instance search over scoped signatures, finding counterexamples by varying these scopes. Consequently, Alloy* relies on an explicit over-approximation of candidate models, making the procedure less direct than in the standard Alloy setting.

7 Conclusion

We addressed the problem of slicing heterogeneous models linked by consistency relations. Unlike program slicing, the goal is not to preserve observable behavior but distinguishability for a consistency predicate. Thus, dependency-based slicing techniques do not apply directly; instead, we adopt a declarative approach.

We define equiconsistency slicing as a synthesis problem: given a model and a consistency predicate, we search for a minimal submodel that remains equiconsistent. This yields a language-agnostic approach independent of both the modeling and the specification formalism. The problem is solved using a CEGIS loop combining counterexample search, classification, and slide synthesis. Our empirical

evaluation stresses several practical observations. First, quantified solving outperforms explicit CEGIS in most of the use cases with few outliers as exception. Second, when models are represented as graphs, structural density dominates runtime more than raw size.

Our approach still has inherent limitations. Guarantees are bounded and therefore relative to a finite domain, and the method relies on the availability of a consistency oracle, which may be expensive or undecidable in richer languages. Moreover, the workflow is currently batch-oriented and not yet incremental. These limitations suggest several research directions, including symbolic encodings for unbounded reasoning (e.g., SMT-based implementations), support for richer consistency languages such as rule-based constraints or transformations, and improved analyzer integration to make the presented approach more suitable for practical modeling environments.

Overall, this work shows that equiconsistency slicing can be understood as a synthesis problem, demonstrating how model synthesis enables language-agnostic techniques for heterogeneous models and could, for instance, be adapted to repair inconsistencies.

References

1. Abate, A., David, C., Kesseli, P., Kroening, D., Polgreen, E.: Counterexample Guided Inductive Synthesis Modulo Theories. In: CAV. pp. 270–288 (2018). doi: 10.1007/978-3-319-96145-3_15
2. Ahmadi, R., Posse, E., Dingel, J.: Slicing UML-based Models of Real-time Embedded Systems. In: MODELS. pp. 346–356 (2018). doi: 10.1145/3239372.3239407
3. Ambler, S.W.: The Object Primer: Agile Model-Driven Development with UML 2.0. 3 edn. (2004). doi: 10.1017/CBO9780511584077
4. Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., Tratt, L.: State-based model slicing: A survey. ACM Comput. Surv. **45**(4), 53:1–53:36 (2013). doi: 10.1145/2501654.2501667
5. Barros, J.B., da Cruz, D., Henriques, P.R., Pinto, J.S.: Assertion-based slicing and slice graphs. Form. Asp. Comput. **24**(2), 217–248 (2012). doi: 10.1007/s00165-011-0196-1
6. Beckert, B., Bormer, T., Gocht, S., Herda, M., Lentzsch, D., Ulbrich, M.: Using relational verification for program slicing. In: SEFM. pp. 353–372 (2019)
7. Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., Korel, B.: Theoretical foundations of dynamic program slicing. Theor. Comput. Sci. **360**(1), 23–41 (2006). doi: 10.1016/j.tcs.2006.01.012
8. Blouin, A., Moha, N., Baudry, B., Sahraoui, H., Jézéquel, J.M.: Assessing the use of slicing-based visualizing techniques on the understanding of large metamodels. Inf. Softw. Technol. **62**, 124–142 (2015). doi: 10.1016/j.infsof.2015.02.007
9. Bowman, H., Steen, M., Boiten, E., Derrick, J.: A Formal Framework for Viewpoint Consistency. Form. Methods Syst. Des. **21**(2), 111–166 (2002). doi: 10.1023/A:1016000201864
10. Canfora, G., Cimitile, A., De Lucia, A.: Conditioned program slicing. Inf. Softw. Technol. **40**(11), 595–607 (1998). doi: 10.1016/S0950-5849(98)00086-X

11. Ehrig, H., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graph Transformation. In: Graph Transform. LNCS, vol. 3256, pp. 161–177 (2004). doi: 10.1007/978-3-540-30203-2_13
12. Färber, H., Pascual, R., Stübinger, T., Ulbrich, M.: Observable Semantics for Characterising Consistency Between Heterogeneous Models. In: SEFM. pp. 110–128 (2026). doi: 10.1007/978-3-032-10444-1_7
13. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Syst. J. **45**(3), 451–461 (2006). doi: 10.1147/sj.453.0451
14. Halder, R., Cortesi, A.: Abstract program slicing on dependence condition graphs. Sci. Comput. Program. **78**(9), 1240–1263 (2013). doi: 10.1016/j.scico.2012.05.007
15. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990). doi: 10.1145/77606.77608
16. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002). doi: 10.1145/505145.505149
17. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE. pp. 215–224 (2010). doi: 10.1145/1806799.1806833
18. Korel, B., Singh, I., Tahat, L., Vaysburg, B.: Slicing of state-based models. In: ICSME. pp. 34–43 (2003). doi: 10.1109/icsm.2003.1235404
19. Lano, K., Kolahdouz-Rahimi, S.: Slicing of UML Models Using Model Transformations. In: MODELS. pp. 228–242 (2010). doi: 10.1007/978-3-642-16129-2_17
20. Lee, W.K., Chung, I.S., Yoon, G.S., Kwon, Y.R.: Specification-based program slicing and its applications. J. Syst. Archit. **47**(5), 427–443 (2001)
21. Lity, S., Lachmann, R., Lochau, M., Schaefer, I.: Delta-oriented software product line test models – the body comfort system case study. Tech. rep., TU Braunschweig (2013)
22. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. Form. Methods Syst. Des. **55**(1), 1–32 (2019). doi: 10.1007/s10703-016-0267-2
23. Pascual, R., Beckert, B., Ulbrich, M., Kirsten, M., Pfeifer, W.: Formal foundations of consistency in model-driven development. In: ISoLA. LNCS (2024)
24. Pietsch, C., Ohrndorf, M., Kelter, U., Kehrer, T.: Incrementally slicing editable submodels. In: ASE. pp. 913–918 (2017). doi: 10.1109/ASE.2017.8115704
25. Salay, R., Kokaly, S., Chechik, M.: Heterogeneous Megamodel Slicing for Model Evolution. In: ME@MODELS. pp. 50–59 (2016)
26. Shaikh, A., Wiil, U.K., Memon, N.: Evaluation of Tools and Slicing Techniques for Efficient Verification of UML/OCL Class Diagrams. Adv. Softw. Eng. **2011**(1), 370198 (2011). doi: 10.1155/2011/370198
27. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: Handbook of Software Engineering and Knowledge Engineering, pp. 329–380 (2001). doi: 10.1142/9789812389718_0015
28. Taentzer, G., Kehrer, T., Pietsch, C., Kelter, U.: A Formal Framework for Incremental Model Slicing. In: FASE. pp. 3–20. LNCS (2018). doi: 10.1007/978-3-319-89363-1_1
29. Thieme, M.: Slicing Models for Equiconsistency with Alloy - Replication Package (2026). doi: 10.5281/zenodo.18706490
30. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**(3), 121–189 (1995)
31. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: TACAS. p. 632–647 (2007)
32. Weiser, M.: Program Slicing. IEEE Trans. Softw. Eng. **SE-10**(4), 352–357 (1984). doi: 10.1109/TSE.1984.5010248