https://doi.org/10.1007/s00165-021-00555-2
The Author(s) © 2021
Formal Aspects of Computing (2021) 33: 1067–1114

Formal Aspects of Computing

Comprehensive systems: a formal foundation for multi-model consistency management

Patrick Stünkel[1], Harald König[1,2], Yngve Lamo[1] and Adrian Rutle[1]
[1]Høgskulen på Vestlandet, Bergen, Norway
[2]FHDW Hannover, Hanover, Germany

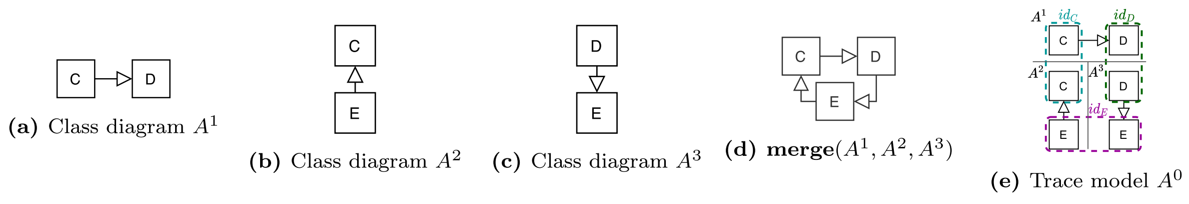
Abstract. Model management is a central activity in Software Engineering. The most challenging aspect of model management is to keep inter-related models consistent with each other while they evolve. As a consequence, there is a lot of scientific activity in this area, which has produced an extensive body of knowledge, methods, results and tools. The majority of these approaches, however, are limited to binary inter-model relations; i.e. the synchronisation of exactly two models. Yet, not every multi-ary relation can be factored into a family of binary relations. In this paper, we propose and investigate a novel *comprehensive system* construction, which is able to represent multi-ary relations among multiple models in an *integrated* manner and thus serves as a *formal foundation* for artefacts used in consistency management activities involving multiple models. The construction is based on the definition of *partial* commonalities among a set of models using the same language, which is used to denote the (local) models. The main *theoretical* results of this paper are proofs of the facts that comprehensive systems are an admissible environment for (i) applying formal means of consistency verification (diagrammatic predicate framework), (ii) performing algebraic graph transformation (weak adhesive HLR category), and (iii) that they generalise the underlying setting of graph diagrams and triple graph grammars.

Keywords: Multi-modelling; Inter-model consistency; Consistency verification; Consistency restoration; Model synchronisation; Multi-directional transformations (MX); Model merging; Model weaving; Graph diagrams; Triple graph grammars; Category theory; Adhesive categories

1. Introduction

Conceptual *models*, i.e. abstract specifications of the system under development, are recognised to be of major importance in software engineering [WHR14]. Representing the whole system in a single (global) model is generally unfeasible [CCP19], hence, different teams design and maintain several models (views) which focus on different aspects of the system. This collection of inter-related models is often referred to as a *multi-model* [BKMW09, SKLR18, DKL19]. A major issue of multi-models is comprehensive *consistency management* [Ste20, FKWVH19, CCP19, SZ01], i.e. keeping the collection of models consistent w.r.t. each other under the ongoing development process to avoid conflicting interpretations of what is being developed.

Correspondence to: Patrick Stünkel, e-mail: past@hvl.no



(a) Class diagram A^1 (b) Class diagram A^2 (c) Class diagram A^3 (d) **merge**(A^1, A^2, A^3) (e) Trace model A^0

Fig. 1. Inconsistent class diagrams, inspired by [DKL19]

Model Synchronisation represents a means for (semi-)automatically restoring consistency in the event of violating model modifications, which is investigated by the cross-disciplinary research domain *Bidirectional Transformations (BX)* [CFH+09]. BX has produced several important results and tools (see [ABW+19] for a recent survey), however, the majority of these approaches are limited to a binary setting, i.e. keeping pairs of models consistent. Stevens [Ste17] recognised this limitation in her outreach to the modelling community leading to an increased momentum in this area [CKSZ19].

One way to address the multi-ary ($n \geq 2$) setting is to consider it as a network of well-understood binary synchronisation problems. However, not every multi-ary consistency rule can be factored into binary ones [DKL19]; e.g. the class diagrams A^1, A^2 and A^3 in Fig. 1(a–c) are pairwise consistent but not altogether—since class inheritance is acyclic. Thus, "proper" multi-ary model synchronisation is needed.

According to [KM18] and [KMCD19], the primary approach for verifying the global consistency of structural models is *merging* [SNL+07]: Elements from all models are collected into a new *global artefact* wherein the inter-related elements are identified. This construction is formally well-understood [BCE+06] and is underpinned by the categorical concept of a *colimit object* [DXC11, KD17]. This idea was first proposed by Goguen in the 70s [Gog73]. An example of a merge is shown in Fig. 1d (inter-relations given by sameness of class' names). Indeed, the merged model identifies the violation of inheritance acyclicity. The major drawback of this approach, apart from the additional computational overhead, is that it forgets the origin of elements; e.g. that class `C` was member of A^1 and A^2 but not A^3. This is a problem when consistency rules depend on this membership information.

Model weaving [BBDF+06] is a another approach [FKWVH19, SDZKR18] to multi-ary synchronisation, that avoids the aforementioned "information loss". It treats inter-relations as an entity of its own right, which are stored in a separate *trace model*, see Fig. 1e. The trace model A^0 for the situation in Fig. 1 contains three links (id_C, id_D, id_E) representing the class name identities and they are visualised using dashed lines. Hence, a trace link represents a tuple of element references. Traversing the trace model and looking up element references in the local models A^1, A^2, A^3 allows to discover the inconsistency as well. But the implementation of the respective "check"-function is more involved. Compared to the merge approach, weaving lacks an equally profound formal foundation and as a result there is lack of interoperability between tools implementing this approach [DKPF09]. For example, due to incompatible trace model formats.

Still, the weaving approach highlights the important fact that inter-relations between models and their elements are the most important ingredient in a multi-model. For the remainder of this paper, we call inter-relations on the model level *correspondences* while inter-relations on the model *element* level are called *commonalities*. It is important to note that a commonality may not always be an identity such as in Fig. 1. A commonality can also express other types of relationships such as refinement, dependency, usage and more, see [FKWVH19, TvdBS20]. Aligning models via an additional commonality structure has some tradition. For instance, it is the idea behind *Triple Graph Grammars (TGGs)* [Sch94], a formal and mature BX approach with a focus on Model Driven Engineering (MDE). The TGG approach considers models to have a *graph-like* structure, i.e. there is a common underlying *base modelling language*; we will also stick to this idea of a common base language.

Our contribution in this paper is a novel construction called *comprehensive system*, which forms a formal foundation for multi-modelling. The construction is based on a non-intrusive *linguistic extension* of the base modelling language with commonality specifications. This allows us to work with an arbitrary number $n \geq 2$ of heterogeneously typed (*local*) models as one single (*global*) artefact. We show that comprehensive systems are a more expressive alternative to the (colimit-based) model merging approach and they are able to serve as the formal underpinning for model weaving. Furthermore, we will prove that it is theoretically possible (i) to apply existing means of consistency verification (diagrammatic predicates) on comprehensive systems, (ii) to apply the algebraic *graph transformation (GT)* framework [EEPT06] on comprehensive systems, and (iii) that comprehensive systems generalise the underlying categories of triple graphs and graph diagrams [TA15]—a multi-ary generalisation of the former.

Considering Multi-Model Consistency Management as a three step process comprising the activities *alignment*, *verification* and *restoration*, comprehensive systems are located in the alignment phase. Verification and restoration are enabled by showing that comprehensive systems admit all formal properties that are required to apply mature model management frameworks for verification and restoration that already exist for local models. The results in this paper are foremost of formal analytical nature. A practical evaluation is left for future work.

Changes compared to the conference version This article is an extended version of the paper *Towards Multiple Model Synchronization with Comprehensive Systems* [SKLR20] published in the proceedings of the 2020 edition of the *Fundamental Aspects of Software Engineering* conference. A major change to the conference version is a completely rewritten "state of the art" (Section 3), which provides a more detailed overview of Multi-Model Consistency Management and contemporary tool support. This allows to set the contribution of comprehensive systems in a bigger context and to motivate their use case. Furthermore, the theory part is extended substantially: We proved the fact that comprehensive systems are organized into a category that has the weak adhesive HLR property w.r.t. a suitable class of reflective monomorphisms $\mathcal{M}$ (Corollary 1 in Section 4.4). This opens the door for the application of the well-established GT-framework and represents a substantial extension compared to the conference version.

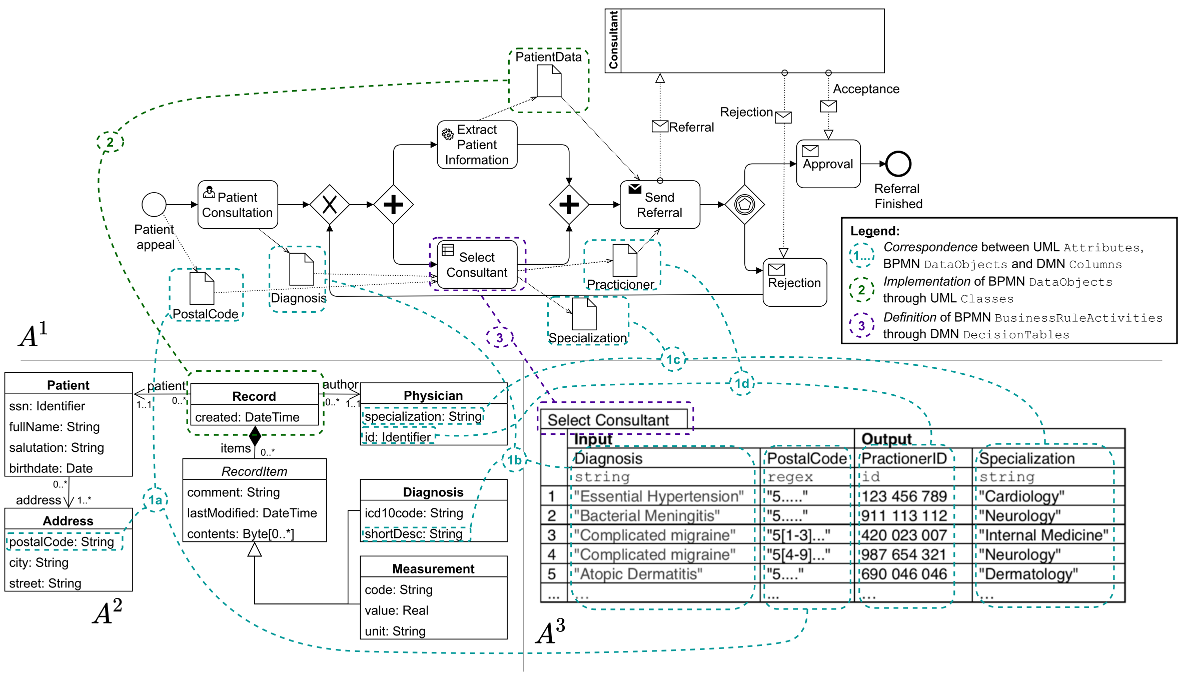
Outline Section 2 introduces a Multi-Model Consistency Management scenario, which will be used as a running example throughout the paper. Section 3 gives an overview of the state of the art of Multi-Model Consistency Management. Section 4 introduces comprehensive systems and their formal properties. Finally, Section 5 concludes the paper with references to related work and future work plans. Moreover, to make this paper self-contained, there is an Appendix, which is divided into two parts. Appendix A contains background on category theory that is required for the proofs in Section 4. Appendix B contains the detailed proofs of the theorems in Section 4.

2. Use case

Our running example stems from the *healthcare* domain and models a *patient referral* process. A referral is "the act of sending a patient to another physician for ongoing management of a specific problem with the expectation that the patient will continue seeing the original physician for co-ordination of total care" [Seg92]. It is an important and recurring process in the healthcare domain. Hence, ICT-support is desirable [WK19]. Furthermore, its design is far from trivial as it involves multiple actors (software vendors, government officials, hospitals and physicians) and aspects (data structures, behavior, interfaces, policies, etc.). A small excerpt of models involved in this design is shown in Fig. 2 (ignore the dashed lines for the moment).

There is a *process* model A^1 denoted in *Business Process Model and Notation (BPMN)* [Obj14], a *data* model A^2 denoted as a *Unified Modelling Language (UML) class diagram* [Obj15], and a *decision* model A^3 denoted in *Decision Model and Notation (DMN)* [Obj19]. The model in A^1 represents a simplified version of the one in [WK19] and specifies the behavioural aspect from the viewpoint of the referring physician: The process is triggered by a patient's appeal beginning with an introductory consultation. Afterwards, information about the patient and its medical history is extracted while in parallel a consultant is selected via a business rule. The patient information is then sent to the consultant. The consultant can either approve the referral or reject it. In the latter case, another consultant has to be found. If a consultant accepts the referral, the process is finished.

This process model is related to the other models: The domain-specific behaviour of the *"Select Consultant"* `activity` in A^1 is specified in the `decision table` model A^3, which for a given combination of `values` in `input side columns`, assigns combinations of `values` in `output side columns`. The `data objects` (represented by file symbols) in A^1 are implemented by respective `classes` or `attributes` in A^2. There are many more examples of such relations in practice [FKWVH19, TvdBS20]: identity, usage, dependency, refinement, and so on. Hence, there is a plethora of names for this concept. For example, traces [DKPF09], corrs [Sch94], morphisms [Ber03], mappings, cross-reference links [dLGKH18].



Select Consultant

	Input		Output	
	Diagnosis	PostalCode	PractionerID	Specialization
	string	regex	id	string
1	"Essential Hypertension"	"5....."	123 456 789	"Cardiology"
2	"Bacterial Meningitis"	"5....."	911 113 112	"Neurology"
3	"Complicated migraine"	"5[1-3]..."	420 023 007	"Internal Medicine"
4	"Complicated migraine"	"5[4-9]..."	987 654 321	"Neurology"
5	"Atopic Dermatitis"	"5...."	690 046 046	"Dermatology"
...	...	...	...	...

Fig. 2. Example models A^1, A^2 and A^3 and their commonalities

As mentioned in the introduction, we use *commonality* [KG19] as the umbrella term for all kinds of structural relationships between elements of disparate models. In Fig. 2, we depict commonalities using dashed lines.

During the development process, models A^1, A^2 and A^3 will be modified by different parties, which can lead to inconsistencies, e.g. changing the name of the "Select consultant" table in A^3 without changing it in A^1. Consistency is often described by the absence of *inconsistencies* and an inconsistency, according to [SZ01], is "a state in which two or more overlapping elements of different software models make assertions [...] which are not jointly satisfiable". Thus inconsistencies are manifold, ranging from the rather simple name inconsistency, mentioned above, over structural inconsistencies (e.g. acyclic inheritance in Fig. 1) to more complex behavioural and interaction inconsistencies, see [TvdBS20]. In the most abstract sense, we consider consistency as a system state that is induced by the validity of so-called *consistency rules* [Egy07]. For our example, assume the following consistency rules:

CR1 Every `business rule activity` in A^1, must be *defined* by a decision `table` in A^3 with the same `name`.

CR2 The `input side columns` (`output side columns`) of a `table` in A^3 must *correspond* to `data objects` that are `consumed` (`produced`) by a related (see CR1) `business rule activity` in A^1.

CR3 `Data objects` in A^1 must be *implemented* by a `class` or `attribute` in A^2.

CR4 Every `column` in A^3 must *correspond* to an `attribute` in A^1 such that their `types` are compatible.

CR5 `Data objects` in A^1, `attributes` in A^2 and `columns` in A^3 must be in ternary "to-one" *correspondence*.

3. State of the art

The problem of inconsistency among inter-related software models has been a major concern of the software engineering community since the late eighties [SZ01]. A prominent study from these early times is the *ViewPoints* framework [FKN+92, FGH+93]: A complex system is described by a set of loosely coupled viewpoints, where each viewpoint may use its own notation. Viewpoints pioneered the usage of logic to define consistency. Each viewpoint has an internal consistency specification and the framework can check consistency both internally to a viewpoint and externally between multiple viewpoints. When inconsistencies are discovered, theframework automatically tries to resolve them using so called meta-level axioms stated in temporal logic, which specify how inconsistencies shall be addressed. A successor in this line of conception is *Xlinkit* [NEFE03, NEF03], a tool for consistency management of XML documents. Consistency rules are defined by a combination of First-Order-Logic (FOL) and XML Path expressions. When the tool discovers inconsistencies, it generates repair actions based on the structure of formulas and the XML document.

With the advent of MDE, the issue of consistency among multiple models has become even more significant [CCP19, Ste20]. This issue is featured in the following contemporary research domains, which mark the related research areas of this work.

- *Multi-View Modeling (MVM)* [BBCW19, CCP19] can be seen as the continuation of the viewpoints idea within MDE. The specification of a complex system requires a multitude of views, i.e. (partial) specifications focusing on a certain aspect of the system (data types, behaviour, components). A prominent example of this principle is UML: It comprises 14 different diagram types for modelling structural and behavioural aspects of a system. The need for view-based specification has also been identified for domain specific modelling languages [GBB12]. A major issue are *overlaps* between views, i.e. when they refer to the "*same*" concepts. When a view is changed, it must ensure that all occurrences of overlaps are changed accordingly in other views to not violate global consistency.
- The latter, known as the view-update problem, embodies the origin of the cross-disciplinary research area *BX* [CFH$^+$09, ASCG$^+$18, ABW$^+$19]. This area comprises researchers from databases, pure mathematics, functional programming, graph transformation and (model-driven) software engineering. The solutions produced in BX are called *synchronisers*, i.e. propagation functions that translate updates from one data source to another and vice versa. BX represents consistency as a *correspondence* relation [Ste08] between synchronised data sources. The update propagation is considered *correct* when the propagation functions always return a result that satisfies the correspondence relation.
- *Megamodelling* stands for a fundamental idea in MDE, where every artefact in the software development process is a model [BJV04, Bé05, FN05]. Models are *transformed* (refined, translated, migrated) to eventually yield a running system via model execution or code generation. The definition of a model transformation can be seen as a model itself and its execution produces a trace-model. Thus, model transformations can again be transformed by higher-order transformations. The fact that these artefacts depend on other models rises the question of megamodel consistency [Ste20].

We group the research areas mentioned above under the term Multi-Model Consistency Management. To avoid confusion by different terminology, we clarify the concepts of Multi-Model Consistency Management here. Fig. 3 gives an overview of both the *artefacts* and *activities* in Multi-Model Consistency Management.

Multi-models are built from (local) *models* (abstract representations of certain parts of a system). Models contain *elements* and are denoted in a graphical or textual modelling language. A collection of models denoted in the same modelling language is called a *model space* and is defined by a *metamodel*. A metamodel comprises a definition of the language's abstract concepts (compare the terms denoted in `teletype` font in Section 2) together with their relationships and structural integrity rules. A *multi-model* (global) is a reification of a correspondence relation among several models, called *components* of the multi-model. The *definition* of a *correspondence* relation is based on *consistency rules* (e.g. CR1–CR5), which can be evaluated on a multi-model resulting in either *true* or *false*. Validity of consistency rules is witnessed by *commonalities*, which establish structural relationships between elements from disparate models (the dashed elements in Fig. 2). Thus, a multi-model is given by a collection of models and commonalities among their elements. We can distinguish between consistency rules that only refer to elements within the same model and those involving multiple models and commonalities. Following the terminology from [UNKC08] the former are called *intra-model* consistency rules, also known as *constraints*. The latter are called *inter-model* consistency rules.

The early literature [SZ01, FST96] identified the following list of activities of the Multi-Model Consistency Management process: *Detection of overlaps*, *Detection of inconsistencies*, *Diagnosis of inconsistencies*, *Handling of inconsistencies*, *Tracking of inconsistencies*, *Specification and application of an inconsistency management policy*. This list still applies today but to simplify presentation, we will merge them into a three-stage process comprising (I) Alignment, (II) Verification, and (III) Reconciliation. Providing an extensive overview of the state of the art for each of these stages would go beyond the scope of this paper. Thus, we only give a brief description of each stage and provide references to existing surveys for further details.

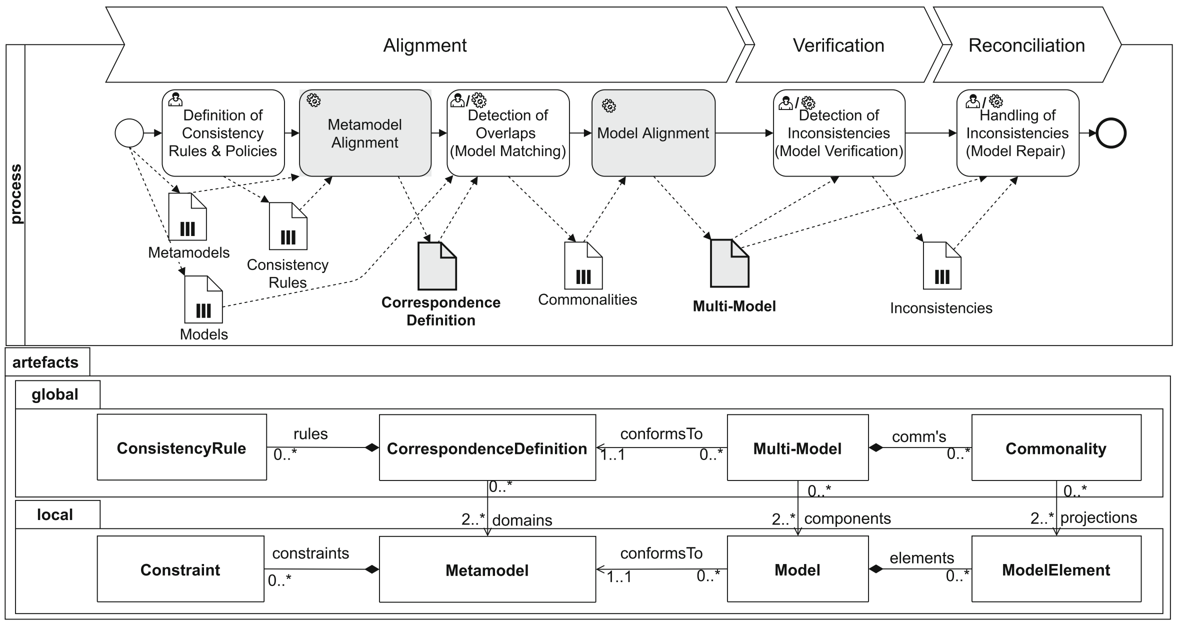


Fig. 3. Multi-Model Consistency Management: Process and Artefacts

3.1. Alignment

Alignment involves the preparatory actions of detecting and representing commonalities. In addition, consistency rules and *policies* [FKN+92] are defined. Policies are meta-rules, that specify how inconsistencies shall be addressed. Spanoudakis and Zisman [SZ01] distinguish between *preventive* (not allowing certain actions in the first place), *remedial* (immediate reaction on inconsistencies) or *tolerating* (doing nothing) policies.

This paper is about a novel formalism for representing multi-models to support consistency management. Thus commonalities play a prominent role, which is why we provide a detailed treatment of their detection and representation.

3.1.1. Commonality detection

The ISO 42010 standard [ISO11] considers the architecture description of a system to comprise multiple views (i.e. models in our terminology). It further distinguishes between *projective* and *synthetic* approaches. In the former case, every model is merely a projection of an underlying all-encompassing system model. The most popular representative of this approach is UML [Obj15] itself: Every diagram displays a certain part of one comprehensive underlying UML model. For example, the same `method`-instance may appear in a class diagram and a sequence diagram. Thus, commonalities are already implicitly known and do not need to be discovered any more. There can still be conflicts between views, e.g. when a `method` name is changed in one view and not the other. The synthetic approach considers all models to be independent entities. Eventually, they have to be *composed* [BCE+06] to yield the resulting system. There are also proposals for combining both projective and synthetic approaches. Orthogonal software modelling [ASB10] is such a representative where one first has to create a *single underlying model (SUM)* (synthetic) based on existing independent local models. The SUM is then used to derive (projective) views from it, i.e. the composition of the synthetic approach is antedated. The construction of a SUM [MWK+20] may be difficult and therefore some researchers proposed to only construct it virtually [KKL+21].

Both synthetic and hybrid approaches require discovery of commonalities, which is also known as *model matching* [KDRPP09]. Spanoudakis and Zisman [SZ01] identify four primary approaches for model matching. The simplest (and arguably most naive) approach is to establish a commonality when there are two or more elements in disparate modelssharing the same *name*.

Table 1. Commonality representation approaches

	Global	Local
Intra-model	Model merging	Dynamic extension
Inter-model	Model weaving	Heterogenous transformation

Another variant is to rely on a *shared ontology*, which requires that all model elements have to be annotated with a term from this ontology. In many cases, model matching via *human inspection* is required, i.e. users have to manually define commonalities. Commonalities come in all different kinds and identifying them is far from obvious. In [BEEH+19], the authors describe how a collaborative decision process can be used for this. Finally, *automated similarity analysis* can be an option. However, being a special case of the weighted bipartite graph matching problem, it is an NP-complete problem and may therefore run into complexity issues [RC13, WWS+17]. For further surveys on (automatic) matching, we refer to [RB01, ES13, KDRPP09].

3.1.2. Commonality representation

Upon studying the literature related to the analysis of the structure of multi-models [KM18, CCP19, dLGKH18, MJC17], we identified four primary approaches for representing commonalities. We call them *Model Merging*, *Model Weaving*, *Heterogeneous Transformations* and *Dynamic Extension*.

Model Merging [SNL+07] means to collect all elements into an all-encompassing model, where elements that are related by a commonality become identified. Projective view modelling approaches are always implicitly based on model merging and Kienzle et.al. [KMCD19] claim that model merging is the default approach to align structural models.

Model weaving is a different approach, which was originally introduced to trace the execution of model transformations [BBDF+06]. It is closely related to model traceability [ARNRSG06]. Commonalities are stored in a separate trace model, i.e. a collection of cross-reference-links. The trace model can be queried and modified independently of the local models.

The Heterogeneous transformations approach does not represent commonalities explicitly. Instead they are implicitly encoded in the definition of transformation functions, which are established between every pair of models. This approach is common in BX, where these functions are called PUT and GET [FGM+07]. Also, the *Queries Views Transformation (QVT)* [Obj16a, Ste08] standard specifies commonalities in this way.

Dynamic extension was pioneered in [EHHS00] and is nowadays often implemented with more lightweight dynamic modelling techniques such as *facets* [dLGKH18]. The idea is related to aspect-oriented programming. Local models are enhanced with commonality meta-data when needed. For instance, we may add a boolean flag to every `business rule activity` in Fig. 2 to check whether the activity has an associated `decision table`, compare CR1.

These four approaches can be classified along a two-dimensional grid, shown in Tab. 1 with the two dimensions *global/local* and *intra-model/inter-model*. Both merging and weaving store commonality information globally (as a merged model or a set of all cross-reference links). Heterogeneous transformations and dynamic extension do not need to consider all models at once since commonality information is stored locally (usually pairwise). Merging and dynamic extension represent the commonality information *within* models while weaving and heterogeneous transformations represent it *outside* of the models.

3.2. Verification

Verification involves the activities centred around finding and tracking (storing and reporting) inconsistencies. The four approaches for finding inconsistencies, according to [SZ01], are *logic-based* (using a resolution procedure), *model-checking* (enumerating all possible instances), *specialized automated analysis* or *human-centred* exploration (inconsistencies are reported manually).

The first two approaches are generic: When models and consistency rules have an encoding as predicates and formulas in a logic, one can use a resolution or model checking procedure that exist for the respective logic to verify consistency. The limitation of both approaches is complexity (state explosion, non terminating resolution). Thus, several means for specialized automated analysis have been developed. An example in the MDE domain is given by the *Object Constraint Language (OCL)* [WK99], which can be used to define and verify consistency rules defined on UML models. The work by Egyed and his collaborators [Egy07, RE12] comprises powerful and field-tested tools that implement consistency verification and restoration in the context of UML/OCL. Human-centred exploration requires the most effort, however, it is the only way to discover inconsistencies for informally given models and consistency rules. A comprehensive survey on consistency verification in the context of UML is given in [KM18]. A more recent survey is [TvdBS20], which also includes other domains than software engineering, e.g. electrical and mechanical engineering.

3.3. Reconciliation

When inconsistencies arise, they have to be analysed and addressed according to the predefined policies. In general, consistency violations trigger a semi-automatic consistency restoration procedure. The latter is also known as *update propagation*, *synchronisation* or *model repair*. It is a vast research field and we can only sketch the primary approaches and concepts here. Surveys on the field are given in [MJC17, ABW+19, SKRL21].

Approaches have been classified into constraint-, search- and propagation-based [OPN20, FKM+20, WAF+19]. However, we want to use the classification from [SKRL21] and propose *search*-based and *rule*-based as the two top level classifiers: Constraint-based can be seen as a special case of search-based model repair and the term rule-based is used to include hand-crafted imperative approaches into the picture.

Search-based approaches are declarative. The repair problem is conceived as a search problem, where the state space is given by the model space, state transitions are given by possible model modifications, and goal states are those models that satisfy all user-defined consistency rules. A naive *atomic search* implementation treats models as black boxes. Due to the sheer size of the state space, atomic search has to combined with additional techniques such as heuristics [SMBB10] or machine learning [BMdlC+20] to cope with complexity. Thus, a prevalent implementation strategy is to translate the problem into a logical representation such that efficient off-the-shelf *solvers* can be used to perform the search. Examples are given by *Echo* [MC16] or *JTL* [EMM+12]. This type of search-based repair is well-aligned with verification using resolution or model-checking as it requires a logical representation of models and consistency rules as well.

Rule-based approaches require more concrete user guidance on how to react to inconsistencies. We distinguish further between *imperative* or *grammar*-based approaches. In the former case, the developer has to write a procedure, which will be executed in the event of a consistency violation [SDZKR18]. Imperative approaches give no further guarantee about correctness. Grammar-based approaches represent a more declarative a approach to rule-based repair. The grammar defines repair rules on a higher level of abstraction, which are then operationalised to produce concrete repairs. An example is the *Model/Analyzer* tool by Egyed et. al. [RE12], which derives possible repairs from the structural rules defined by the UML metamodel. Another prominent formal representative is given by the *(algebraic) graph transformations* framework [EEPT06]. The latter represents rules by means of graph-homomorphisms and rule application is defined via a *Double Pushout (DPO)* construction [EPS73]. Consistency rules are defined by means of a graph grammar [Roz97]. This framework offers means for the analysis of *internal properties* (concurrency, confluence, termination) [EEPT06] and *correctness properties* (compliance between rules and static conditions) [HP09]. Model repair approaches, which utilize this framework, exist both for local models [KR17, OPKK18, SLO19] and multiple models [HEO+11, WAF+19, FKM+20]. The latter is represented under the umbrella of TGGs.

Orthogonally, model repair approaches must take into account cross-cutting concerns such as *user interaction*, *incrementality*, *concurrency* and *optimality*. The fact that repair results are not always unique necessitates user interaction. For example, by letting the user choose a preferred solution among several possible solutions. The *Model/Analyzer* tool [RE12] relies heavily on user interaction. Incrementality was highlighted by Giese [GW09]: Complexity of model repairs should not depend on the size of the models but on the size of the modification, i.e. avoiding to re-compute the whole correspondence relation from scratch. Support for concurrent model synchronisation (repairing inconsistencies in the aftermath of parallel and independent model modifications) has been rather limited until lately [OBE+13]. But recently there has been some interesting new results in this direction related to rule-based approaches [OPN20, FKM+20, WFA20]. Finally, if there a multiple correct solutions for a repair, there arises the questions of what should be considered the “best” solution. Both quantitative (i.e. metrics) [MC16] and qualitative [CGMS15] measures have been proposed. However, it may be noted that some changes can only be *ameliorated* instead of *rectified* completely [SZ01]. In fact, toleration of inconsistencies may be an adequate reaction [NER01] as well.

3.4. Existing tools

In accordance with the phases previously described, there is a large number of existing tools. Hence, it is not possible to cover a broad selection here. We pick a small selection of contemporary tools to illustrate existing issues in Multi-Model Consistency Management that we want to address by comprehensive systems.

3.4.1. Epsilon (matching, merging, verification)

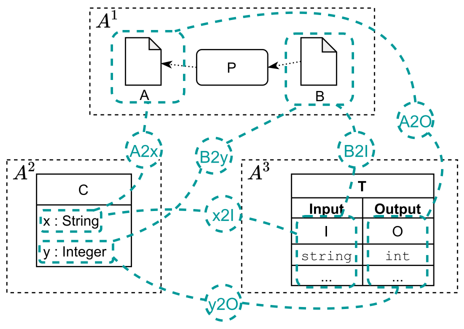
Model-driven design and development is supported by model management tools providing facilities for common tasks such as querying and modifying a model's contents, verifying the consistency of a model, merging two models or translating a model into a different representation. *Epsilon*[1] [PKR+09] is a well-established representative of such a model management tool. It is organized as a set of Domain Specific Languages (DSLs), one for each model management task, and, among others, DSLs for matching (ECL), merging (EML) and verifying models (EVL) [KPP06].

Listing 1: Model Matching and Merging with Epsilon DSL

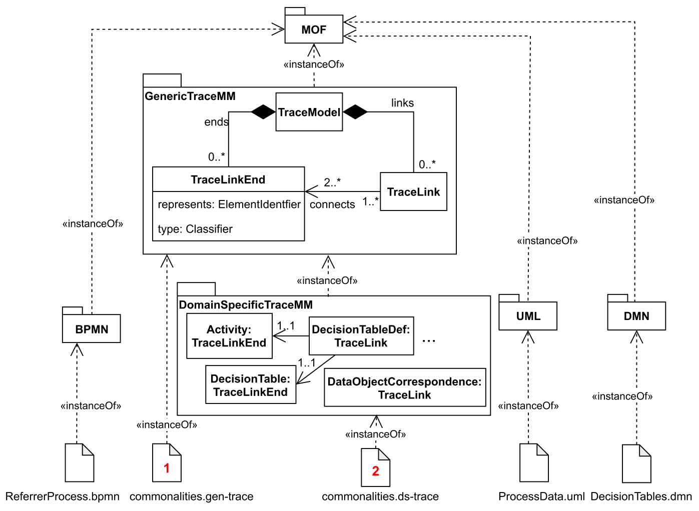
```
1  /* match rules */
2  rule MatchDecisionTableDef match l:A^1!Activity with r:A^3!DecisionTable
3    {    guard : l.type = ActivityType:BUSINESS_RULE
4         compare : l.name = r.name }
5  rule MatchDataObjectCorrespondence match l:A^1!DataObject with r:A^3!Column
6    {  guard: l.consumers.matches().inputSideColumns.includes(r) or
7             l.producers.matches().outputSideColumns.includes(r)
8       compare:  l.name.isAlike(r.name) }
9  /* merge rules */
10 rule MergeDecisionTableDef
11 merge l:A^1!Activity with r:A^3!DecisionTable into t : A^+!DecisionTableDef
12   { t.name = l.name;
13     t.isMatched = true; }
14 rule MergeDataObjectCorrespondence
15 merge l : A^1!DataObject with r : A^3!Column into t : A^+!DataObjectCorrespondence
16   { t.name = l.name; }
17 rule CopyBusinessRuleActivity
18 transform s : A^1!BusinessRuleActivity to t : A^+!DecisionTableDef
19   { guard : s.type = ActivityType:BUSINESS_RULE
20     t.name = s.name;
21     t.isMatched = false; }
22 /* consistency rules */
23 context DecisionTableDef {
24   {  constraint CR1 { check : self.isMatched
25          fix { var table : new A^3!DecisionTable
26                 table.name = self.name;  } }
27      constraint CR2 { check : self.produces = self.outputSideColumns and
28          self.consumes = self.inputSideColumns } }
```

Listing 1 shows the Epsilon code needed to implement consistency verification for CR1 and CR2. In the first step, elements from disparate models have to be matched (lines 2–8). Epsilon performs automatic pairwise model matching, which is controlled by user-defined rules. A rule defines the model element types that should be matched with each other (keywords `match` and `with`), when a commonality should be established (`compare`), and optionally a filter-criterion (`guard`). The matching engine compares all pairs of model elements with the respective types and creates commonalities when guard and match criterion are fulfilled. In Epsilon vernacular, commonalities are called *match traces*. They are processed further to create a merged model A^+, compare Fig. 1d. This step is controlled via respective merge-rules (lines 10–16) and copy-rules (lines 17–21). Merge rules are invoked for all match traces with the respective types and produce an element in the merged model. Afterwards copy rules are invoked, which copy unmatched elements into the merged model. It is important to note that our example in Section 2 is *heterogeneous*, i.e. the models are denoted in different modelling languages. In order to create a merge in this case, we have to create a single underlying metamodel (SUMM) beforehand, which encompasses concepts from BPMN, DMN and UML (i.e. another instance of model matching and merging on the metamodel level). For more details about this issue, we refer to [DXC11].

[1] https://www.eclipse.org/epsilon/



(a) Insufficient Ternary Relation Representation



(b) Representing Trace Models

Fig. 4. Practical Challenges

When the merged model is created, we can check whether it fulfils the global consistency rules (lines 23–28), which are formulated in an OCL-like language called Epsilon Verification Language (EVL). These rules can be augmented with a `fix` statement (lines 25–26). The latter defines an imperative program to restore consistency, e.g. creating the missing `decision table`.

Concerning the Epsilon solution in relation to our presentation of Multi-Model Consistency Management: Matching is performed via automatic model comparison, which is controlled by user-defined rules, commonalities are reified in a merged model, consistency verification is implemented via specialized verification means (EVL) and repair is performed in an imperative rule-based manner.

In Section 1, we mentioned that merging is a forgetful operation: The origin of elements and the information whether an element was merged with another is lost after the merge. Thus, in order to verify CR1, we had to augment the merge-rule (line 13) and the copy-rule (line 21) with this meta-information to be available in the resulting merged model, where consistency verification is performed.

While information loss can be overcome in the aforementioned way, Epsilon suffers from a major limitation: In its present version it only supports *pairwise* matching. Therefore, while CR1–CR4 are implementable, CR5 cannot be realised with this tool since it requires a ternary relation. It is not enough to only look at the pairs (A^1, A^2), (A^2, A^3) and (A^1, A^3). Consider the situation in Fig. 4a: Each model pairing is apparently consistent since there are binary one-to-one correspondences but taken altogether the *ternary* "to-one" correspondence is violated. We conclude with two requirements for a formal multi-modelling foundation.

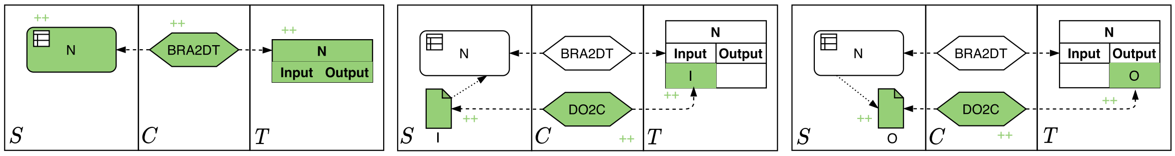
Requirement 1 Comprehensive Systems must not forget the origin of elements from the original models.

Requirement 2 Comprehensive Systems must be able to handle arbitrary n-ary correspondences.

3.4.2. Generic and domain-specific trace models

Instead of turning the match traces into a merged model, we can turn them into a *trace model*. As pointed out in [DKPF09], commonalities represent an entity of its own right. Augmenting a set of models with a trace-model is known as *model weaving* [BBDF+06]. In its most generic form, a trace model is a (hyper-) graph. Elements are either `trace links` (edges) or `trace link ends` (nodes). The latter serve as *proxies* [GHJV95] of elements in another model. Working with such generic trace models is cumbersome and the definition of consistency rules over such trace models becomes rather involved. In practice, one distinguishes between different types of commonalities that are established only among elements having a specific type. For example, `columns` in A^3 can only be related to `data objects` in A^1 and `attributes` in A^2.

The default approach to capture this notion is the definition of a *domain-specific* trace metamodel [FKWVH19, SDZKR18], which contains domain-specific refinements of `trace links` and `trace link ends`. An overview of generic and domain specific trace-models, their metamodels and instantiation relationships is shown in Fig. 4b.



(a) Business Rule Activities correspond to Decision Tables **(b)** Consumed data object correspond to input side columns **(c)** Produced data object correspond to output side columns

Fig. 5. TGG production rules

Metamodels are depicted as packages and models as files at the bottom. Notice the double nature of the generic trace-metamodel: It can either be instantiated directly by a generic trace-model (1) or serve as the metamodel for a domain-specific trace-metamodel, which is instantiated by a domain-specific trace model (2). The domain specific trace metamodel is a suitable carrier for the definition of consistency rules. One may use Epsilon or other model management tools to implement verification and repair [SDZKR18].

Note, that the existence of a separate trace model induces a new challenge: Because `trace link ends` are proxies of elements in other models, we have created a situation that requires n-binary synchronisations, i.e. when an element in a local model changes also its proxy in the trace model must change. Compared to the solution in 3.4.1, the only major difference is the way of representing commonalities. The solution of using domain-specific trace models is very common in practice, see [FKWVH19, SDZKR18]. However, Drivalos et.al. [DKPF09] reported that there is missing interoperability between solutions using trace-models due to the fact that many implementations are created in an ad-hoc manner.

Requirement 3 Comprehensive Systems shall provide a formal foundation for domain-specific trace-models in model weaving.

3.4.3. *Triple graph grammars*

Finally, we investigate a formal approach that is based on an auxiliary commonality structure. TGGs [Sch94] are means for defining consistency rules between two structures (e.g. models) represented as *graphs* in a declarative manner. A TGG is graph grammar [Roz97], which trades "ordinary" directed graphs for *triple graphs*. The latter is formally given by a pair of graphs[2] (S, T) connected by a "correspondence graph" C that relates S and T via graph homomorphisms, resulting in a span $S \leftarrow C \rightarrow T$. A single $TGG = (tg_0, \mathcal{R})$ comprises a *start* triple graph tg_0, e.g. one that is empty in all components, and a set of *production* rules $\mathcal{R}$. In the default case, production rules are monotonic production rules, which are formally given by inclusion-morphisms $r := L \hookrightarrow R$ (L and R being triple graphs), that specify how the two structures evolve simultaneously. Intuitively, monotonic rules *add* elements to an existing context. In Fig. 5, we depict three exemplary production rules in an integrated presentation: Elements in $R \setminus L$ are added to an existing context. These are highlighted in Fig. 5 by a shaded background and a ++-annotation. The remaining elements are members of L. The TGG induces a language: The set of all triple graphs producible by applying a sequence of production rules on the start triple graph. This language is a subset of all possible triple graphs and hence defines a consistency relation on the collection of all triple graphs or equivalently a relation between S and T if the middle part C is ignored. The language generated by the three rules in Fig. 5 (with an empty start triple graph) models the semantics of CR1 and CR2.

TGGs are a declarative approach. The grammar rules are used to automatically derive programs for (incremental) model transformation [EEE+07, GW09], model matching [EEH08], consistency verification [LAS17] and update synchronisation [HEO+11, HEEO12]. This is done via a so-called "operationalisation" of the grammar rules [HEO+11]. Recently, the limited support [OBE+13] for concurrent model synchronisation has been addressed from several researchers. Orejas et.al. [OPN20] investigate a theoretical approach to cope with conflicting updates via parsing, Fritsche et.al. [FKM+20] address this issue from a practical side using precedence graphs, and Weidmann et.al. [WFA20] combine the rule-based nature of TGGs with a search-based approach. There exist

[2] Often named *source* and *target* due to historic reasons.

industrial-proven state-of-the-art tool implementations for TGGs, e.g. *eMoflon*[3] [WAF+19] and *MoTE*[4] [GHL10]. Finally, practitioners have used TGGs as a graphical language for the definition of consistency rules, which were used to generate an implementation in the Epsilon framework [FKWVH19, GdLKP10].

Classifying TGGs within Multi-Model Consistency Management, they come with automatic model matching, specialized verification and a rule-based repair mechanism, which are based on a declarative grammar specification. The commonality representation lies somewhere between model weaving and heterogeneous transformations: There is no explicit trace-model but commonalities appear implicitly as correspondence graphs in rule definitions.

As a conclusion, TGGs combine an intuitive visual language with a powerful theoretical framework and tool support. However, triple graphs are by definition limited to binary situations and thus fail to capture the semantics of CR5. A generalisation of triple graphs for multi-ary situations has been introduced in the form of *graph diagrams* [TA15, TA16]. Graph diagrams allow the definition of multi-ary relations but require that the arity of all relations is known beforehand because their underlying schema is fixed. We doubt that the set of all necessary relations in a concrete use case can be known beforehand. Thus, we want to to develop a formalism that can deal with these inter-relations more flexibly. In particular, we want to support the introduction of new relations at runtime.

Requirement 4 Comprehensive Systems must support the flexible expression of multi-ary correspondence relations and also support relations that may change their arity over time.

4. Comprehensive systems

In this section, we define comprehensive systems. We begin with reviewing a formalisation of (local) models and constraints imposed on them in Section 4.1. Afterwards, we develop the idea behind comprehensive systems intuitively along our running example in Section 4.2 before providing the formal definition of comprehensive systems using algebra and category theory in Section 4.3. In Section 4.4, we explore the theoretical properties of comprehensive systems. Moreover, we show that they generalise graph diagrams and triple graphs in Section 4.5. We conclude with a short discussion about the application of comprehensive systems in practice (Section 4.6), their current limitations and how they satisfy the requirements from Section 3.4.

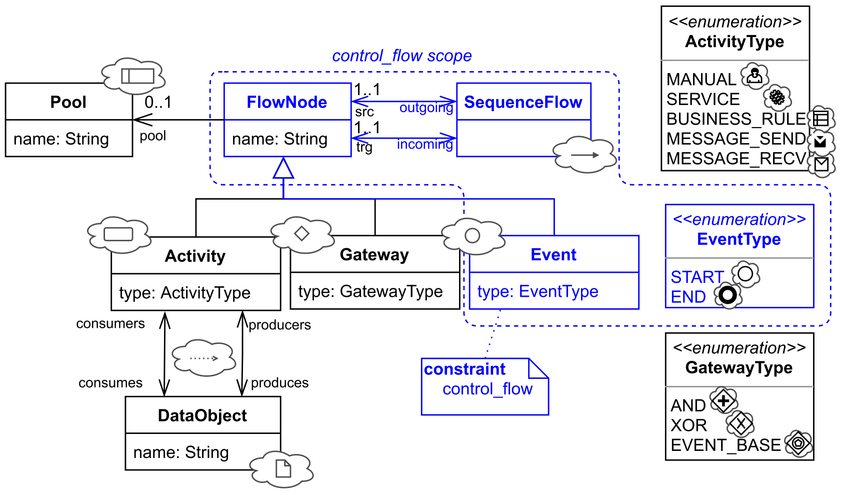
4.1. Software model formalisation

In our example use case (Section 2), we employ three modelling languages: BPMN, UML and DMN. Each language is defined by a metamodel (syntactical representation of a class of models). These metamodels are themselves defined using the meta-metamodelling language *Meta Object Facility (MOF)* [Obj16b]. MOF is essentially a subset of the UML class diagram language comprising `classes`, `attributes` and `references`. A simplified version of the BPMN metamodel, termed M^1, is depicted in Fig. 6a (clouds allude to concrete syntax). Metamodels M^2 and M^3 for UML class diagram and DMN decision tables can be defined accordingly (excerpts of them are shown in Fig. 8). A metamodel defines the concepts of the language together with structural relations between concepts (signature) and structural integrity rules (formulas) over them. Structural integrity rules, e.g. multiplicities (`1..0`, `1..1`), are used to enforce common domain-specific requirements. Often, the built-in mechanisms are not enough to encode all domain-specific requirements. Therefore, constraint languages such as EVL [KPP08] or OCL [WK99] exist, which allow the designer to attach arbitrary user-defined constraints onto metamodels. Fig. 6a features an attached constraint ϕ :=`control_flow`, which is expressed as an OCL invariant defined in Listing 2.

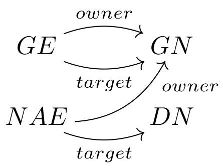
Listing 2: Constraint ϕ:=`control_flow` formulated in OCL

```
context Event inv control_flow:
  (self.type=EventType::START implies
    self.incoming->count() = 0)
  and (self.type=EventType::END implies
    self.outgoing->count() = 0)
```

[3] https://emoflon.org/
[4] https://www.hpi.uni-potsdam.de/giese/public/mdelab/



(a) BPMN metamodel M^1



$$\begin{tikzcd} GE \arrow[r, bend left, "owner"] \arrow[r, bend right, "target"'] & GN \\ NAE \arrow[ur, "owner"] \arrow[r, "target"'] & DN \end{tikzcd}$$

(b) Simplified E-graph signature

Fig. 6. Metamodel Example and Base Language Signature

This constraint requires that every `start event` must not have any incoming `sequence flow` [Obj14, p. 237], whereas `end events` must not have any outgoing `sequence flow` [Obj14, p. 245].

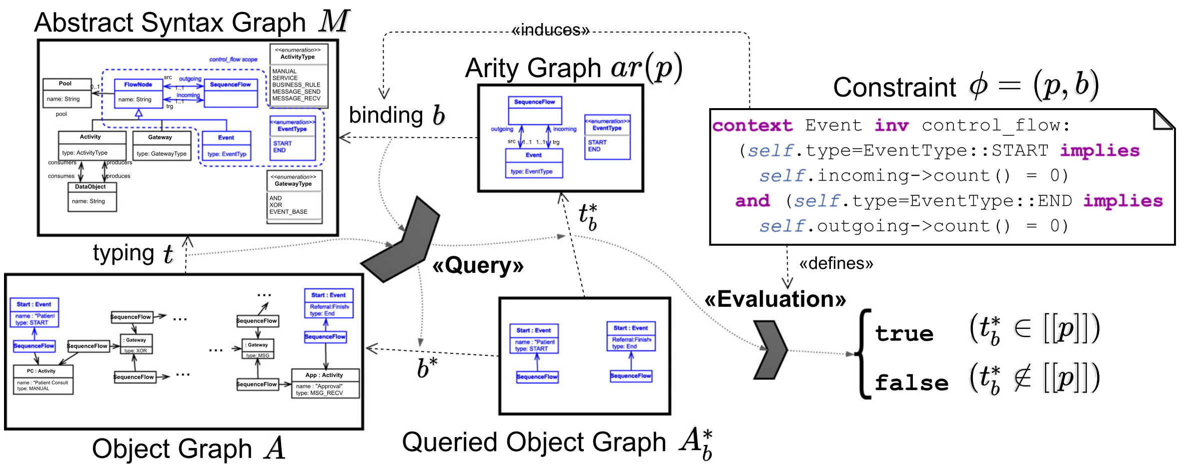
MOF and its derivations, such as *Ecore* [SBMP08], are widespread. However, we do not endorse one particular modelling language and instead seek for a technology independent (= mathematical) formulation of models, metamodels and constraints.

E-graphs [EEPT06] (see Fig. 6b) are one suitable formal interpretation of the MOF/class-diagram syntax and thus an appropriate *base modelling language* $\mathbb{B}$ (*linguistic (meta-) metamodel* in [Kü06]) to encode the abstract syntax graph defined by a metamodel. The E-graph language comprises the concepts Graph Nodes GN (complex types), Data Nodes DN (primitive types), as well as Graph Edges GE (associations) and Node Attribute Edges NAE (attributes) together with appropriate owner and target functions. For the sake of simplicity we omitted edge attribute edges, which are usually included in E-graphs.

It must be mentioned, that our formalism is not "tied" to E-graphs: The formal definitions in Section 4.3 are based on arbitrary graph-like structures (see Definition 1 in Section 4.3), where E-graphs are one concrete example. Hence in the following, we use the term *graph* to refer to any kind of graph-like structures. To require that the content of a (meta-) model must have a graph-like structure is not a major limitation since the majority of graphical and textual modelling languages admit such a representation.

Metamodels are instantiated by models, which are object graphs typed over the abstract syntax graph defined by the metamodel. Thus, ignoring their concrete syntax, the three models A^1,A^2 and A^3 in Fig. 2 each form an object graph w.r.t. M^1, M^2, and M^3. The instantiation relationship between a model A (object graph) and its metamodel (class diagram) is formally represented by a graph homomorphism $t : A \to M$. Let for example a be the element named *"Diagnosis"* in A^1 then $t(a) = \texttt{DataObject} \in M^1$. Given a fixed metamodel M, we call the collection of all typing morphism $\mathbf{Mod}(M) := \{t \mid t : A \to M\}$ the *model space* defined by M.

Graphs and graph homomorphisms alone are not enough since metamodels also comprise built-in (e.g. `1..1`, `0..1`) and attached (e.g. `control_flow`) constraints. *Generalised sketches* [DW07, Dis97] and the *Diagram Predicate Framework* [RRLW12, RRLW09] (an MDE oriented adaptation of the former) provide an elegant, yet flexible formalism to express different kinds of constraints in a uniform way and generalise the four approaches presented in Section 3.2. The idea is to express constraints as diagrams (in the category theoretical sense) that are bound to graph elements. Constraint semantics are kept abstract, i.e. delegated to an arbitrary predicate library. This allows designers to implement constraint semantics using the formalism or tool of their choice: EVL/OCL [KPP08, WK99], (nested) graph conditions [HP09], First-Order Logic (FOL) [NEFE03], or arbitrary programming languages.



```
context Event inv control_flow:
  (self.type=EventType::START implies
    self.incoming->count() = 0)
  and (self.type=EventType::END implies
    self.outgoing->count() = 0)
```

Fig. 7. Diagrammatic Constraints in a Nutshell

The idea behind diagrammatic constraints is sketched in Fig. 7. A constraint ϕ (similar to a formula in FOL) is formally given by a pair (p, b), which consists of a binding morphism[5] b and a predicate p.

Predicates p are organised in an abstract but fixed predicate signature Π. It can be thought of as a library, which may contain the UML/MOF-constraints [RRLW09], functions on some base data types as in OCL [WK99], and logical connectives [Wol21]. Each predicate has a fixed *arity* graph $ar(p)$ and a semantic interpretation $[\![p]\!] \subseteq \mathbf{Mod}(ar(p))$, i.e. a chosen subset of graphs typed over the arity graph closed under renaming. Semantics are commonly defined given a boolean function $check_p : \mathbf{Mod}(ar(p)) \rightarrow \{\texttt{true}, \texttt{false}\}$ which is equivalent to a subset: Given a model $i \in \mathbf{Mod}(ar(p))$ the check function applied on i returns `true` if and only if i belongs to the semantics ($check_p(i) = \texttt{true} \Leftrightarrow i \in [\![p]\!]$). We then say that i satisfies p, written $i \models p$. As an a example, consider the predicate `target[1..1]`, which represents the UML multiplicity *exactly one* at the target side of an edge. The arity of this predicate is a graph containing a single edge: $ar(\texttt{target[1..1]}) := Edge$ with $GN_{Edge} = \{s, t\}$, $GE_{Edge} = \{e\}$, $NAE_{Edge} = DN_{Edge} = \emptyset$, $owner^{Edge}(e) = s$, and $target^{Edge}(e) = t$, recall Fig. 6b. Thus the function $check_{\texttt{target[1..1]}}$ expects a typed graph $i : I \rightarrow Edge$ as input and verifies whether there is exactly one outgoing edge for every node typed over s in I. A FOL-theoretic formulation of `target[1..1]`'s semantics may be given as follows:

$$check_{\texttt{target[1..1]}}(i) = \texttt{true} \Leftrightarrow \quad (\forall\, n \in GN_I : i(n) = s \Rightarrow \exists\, e \in GE_I : owner^I(e) = n \tag{1}$$

$$\wedge \quad \forall\, n \in GN_I,\, e, e' \in GE_I : owner^I(e) = n = owner^I(e') \Rightarrow e = e') \tag{2}$$

The metamodel in Fig. 6a comprises several unnamed constraints, e.g. the edges `src` and `trg` both have the predicate `target[1..1]` attached to them. The named constraint ϕ :=control_flow exemplifies the binding of a more complicated predicate. Its semantics are defined by the OCL-code in Listing 2 and the arity (scope) is the subgraph of M^1 highlighted in Fig. 6a.

To check whether an M-model satisfies a constraint $\phi = (p, b)$, one first has to "pull" the respective typing homomorphism $t : A \rightarrow M$ "back" along the binding homomorphism b (i.e. querying the scope) and then verify membership of the result w.r.t the semantics of p (i.e. invoking the check-function). When the binding homomorphism is injective, this "pulling-back" or querying operation simply means forgetting all parts of A, which t maps to an element outside of the scope of ϕ. The result of querying a typed graph $t : A \rightarrow M$ along the binding homomorphism b of a diagrammatic constraint $\phi = (p, b)$ is denoted by $t_b^* : A_b^* \rightarrow ar(p)$ and we say that t satisfies ϕ if and only if t_b^* satisfies p:

$$t \models \phi \Leftrightarrow t_b^* \models p \tag{3}$$

[5] Considering binding as a homomorphism generalises binding of predicates in atomic formulas in FOL. E.g. think of a predicate *lessEq* stating whether one number is less or equal than another. It can be used as a diagrammatic constraint where the arity graph is *discrete*, i.e. a graph containing two nodes {1, 2}. Now writing $lessEq(a, b)$ is interpreted as a graph homomorphism that binds 1 to a and 2 to b.

If a set Φ of diagrammatic constraints (a set of formulas given in a specific logic) is imposed on M, then the space is reduced to the subset $\mathbf{Mod}(M_\Phi) \subseteq \mathbf{Mod}(M)$ of all *consistent* models typed over M subject to Φ (those satisfying all constraints).

The fact that instances are interpreted as typed graphs induces a default multiplicity for edges: If there is no additional multiplicity bound on the respective edge, this edge implicitly has the `0..*` at both ends. This actually differs from the default multiplicity in UML [Obj15], which is `1..1` at the target side (`0..1` at the source). Thus, for every metamodel in this paper it holds that, if there is no explicitly specified multiplicity, there is an implicit `0..*` multiplicity at both ends.

To summarise the essence of generalised sketches/diagrammatic predicates: Software models and metamodels have a graph-like structure, models are typed graph-like structures, and the definition and verification of constraints requires the existence of *chosen subsets* and a *"pulling-back"* (query) operation.

4.2. Intuition behind comprehensive systems

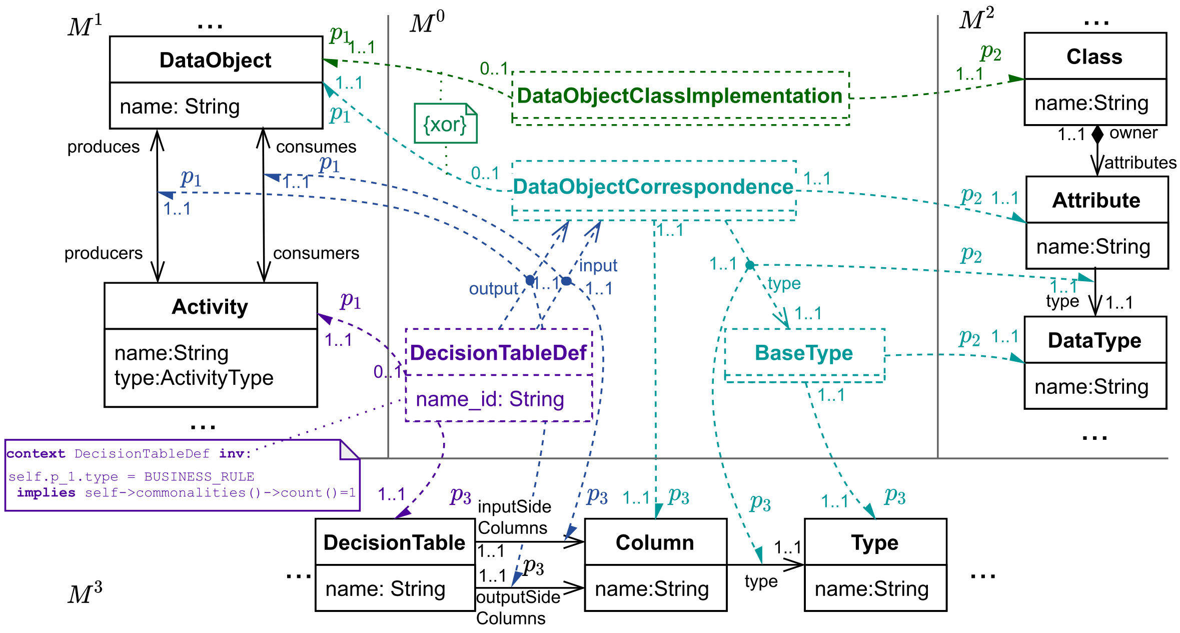
To align multiple models with each other in a multi-model, one needs a language to express commonalities between these models. As discussed in Section 3, the majority of contemporary mapping languages are bound to binary situations. A notably exception, which allows multi-ary correspondences is the *commonalities* language by Klare and Gleitze [KG19].

Listing 3: Type Commonalities

```
1  commonality DataObjectClassImplementation { with BPMN:DataObject with UML:Class }
2  commonality BaseType { with UML:DataType with DMN:Type }
3  commonality DataObjectCorrespondence { with BPMN:DataObject with UML:Attribute with DMN:Column
4     has type referencing BaseType { = UML:Attribute.dataType = DMN.Column.type }
5  }
6  commonality DecisionTableDef { with BPMN:Activity whereat BPMN:Activity.type=BUSINESS_RULE
7                                 with DMN:DecisionTable
8     has name { = BPMN:Activity.name = DMN:DecisionTable.name }
9     has input referencing DataObjectCorrespondence {
10       = BPMN:Activity:consumes = DMN:Table:inputSideColumns }
11    has output referencing DataObjectCorrespondence {
12       = BPMN:Activity:produces = DMN:Table:outputSideColumns }
13 }
```

Listing 3 demonstrates how the commonalities from our running example in Fig. 2 are expressed in this language. The keyword `commonality` initiates the definition of commonalities between instances of respective elements, which stem from disparate metamodels (referenced via the `with` keyword). Additionally, commonalities may be linked with each other (keywords `has` and `referencing`). In [KG19], commonalities are used to define expressions on them, which encode consistency rules. These expressions are translated into a so-called *Reactions* language, which provides event-based model modification facilities to perform consistency restoration. Their approach is part of the *Vitruvius* framework [KKL+21].

We do not want to go further into concrete details of this practical approach, but instead analyse the formal semantics of the code in Listing 3. Commonalities together with their attributes and references, again, form a graph. Consequently, it is reasonable to use the same graph-like language $\mathbb{B}$ for it. In such a way, the content of Listing 3 induces an E-graph, shown in Fig. 8. The elements of this graph are depicted using dashed lines and we call them *commonality witnesses*. Commonality witnesses reify a "tupling" of terms from disparate (meta-) models. They are defined via the `with` keyword in Listing 3, visualised as dashed arrows (p_1, p_2, p_3) in Fig. 8. These arrows are called *projections* and represent the fundamental innovation compared to the situation of only local models from Section 4.1. For example, lines 3–5 specify a commonality of the triple `DataObject` (M^1), `Attribute` (M^2), and `Column` (M^3) reified under the name `DataObjectCorrespondence` in M^0. However, not only the nodes (of the graphs) are related: In Listing 3, we see how the keyword `has` defines how inherent features, i.e. edges, are related as well, e.g. line 4 specifies a commonality between the `type` attributes in M^2 and in M^3. The same goes for the `consumes`/`inputSideColumns`, `produces`/`outputSideColumns` and `name` features of `Activity` (M^1) and `DecisionTable` (M^3). Common edges require that their respective source and target nodes are also related, e.g. the `type`-commonality *depends* on a commonality between `Attribute` and `Column`, which is already given by the surrounding `commonality`-statement, as well as commonality between `Type` and `DataType` (see line 2). Hence, commonality specifications must preserve edge-node-incidences.



```
context DecisionTableDef inv:
self.p_1.type = BUSINESS_RULE
 implies self->commonalities()->count()=1
```

Fig. 8. Commonality representative metamodel M^0

Projections represent an *extension* of our base language. The whole setting can be interpreted as a *linguistic extension* in the sense of [dLG10, AK02]: The linguistic metamodel, which is induced by the base language $\mathbb{B}$ (e.g. nodes and edges) is extended by the new *projection* concept.

The formal result of the specification in Listing 3 are 3 *projection mappings* $p_j : M^0 \rightharpoonup M^j$ ($j \in \{1, 2, 3\}$), depicted by dotted arrows in Fig. 8. For example, $p_1(\texttt{DecisionTableDef}) = \texttt{Activity} \in M^1$, the target metamodel now encoded in p's index.

Since the commonality tuples can be of arbitrary arity, these mappings may be *partial* (highlighted by denoting them as arrows headed with a half-tick: $\rightharpoonup$):

$$p_1(\texttt{BaseType}) = \perp,\ p_2(\texttt{BaseType}) = \texttt{DataType},\ p_3(\texttt{BaseType}) = \texttt{Type}$$

The above required edge-node-incidence means that defined-ness of $p_j(e)$ entails defined-ness of $p_j(v)$, where v is the *owner* of e in M^j, and

$$p_j(v) = owner^{M^j}(p_j(e)) \tag{4}$$

for all edges e in M^0 (and likewise for targets).

Hence, Listing 3 defines a comprehensive metamodel M in which commonalities are accurately specified with the help of (a graph of) commonality representatives. Formally, we obtain a new graph M^0 and partial projections $p_j : M^0 \rightharpoonup M^j$ for all $j \in \{1, \ldots, 3\}$.

Now, turning to models $A^1 \in \mathbf{Mod}(M^1), \ldots, A^3 \in \mathbf{Mod}(M^3)$, i.e. typed graph structures $t_1 : A^1 \to M^1, \ldots, t_3 : A^3 \to M^3$, it becomes apparent that we can establish a corresponding construction relating elements in the domains of these typing homomorphisms. They may be defined manually as in Listing 3 or using (semi-)automatic matching procedures, see Section 3.4.1, based on keys, metrics or ontological equivalence. Independently of *how* they are established, their formal representation is again a graph of commonality representatives A^0 and partial projections $p_j^A : A^0 \rightharpoonup A^j$ for all $j \in \{1, \ldots, 3\}$. This alignment of models is implicitly shown in Fig. 2. Each dashed circle (1a,1b,1c,1d,2,3) represents a commonality representative and each line ends at the value under the respective projection. Some of the lines are binary, while others are ternary. The complete content of Fig. 2 is called a *comprehensive system* where the dashed part represents the *commonalities* and the models $A^1, \ldots, A^3$ are the *components*.

Models A^i are typed over their metamodels, i.e. there are typing morphisms $t_i : A^i \to M^i$ which can be combined to one comprehensive typing of all components. This typing extends to A^0 as well because elements a_j and a_k ($j \neq k$) of model components A^j and A^k are relatable only if their types $t_j(a_j)$ and $t_k(a_k)$ are related via a representative $w \in M^0$. Thus, the specification in Listing 3 defines the possible types of commonalities. This is the formal equivalent to domain-specific trace models, compare Section 3.4.2.

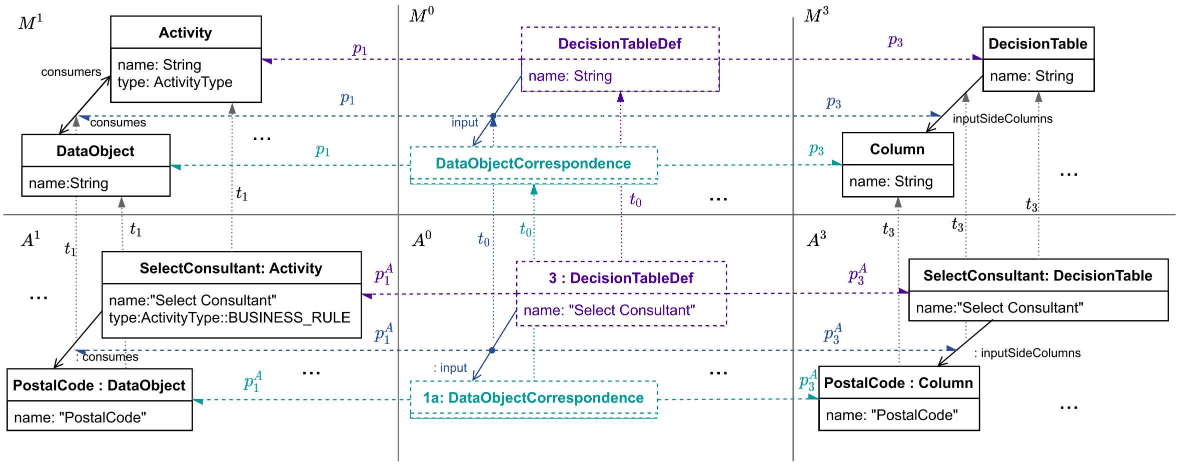


Fig. 9. Compatibility of typing

A natural typing t_0 of a commonality representative v of a_j and a_k is $t_0(v) := w$, such that

$$p_j(t_0(v)) = p_j(w) = t_j(a_j) = t_j(p_j^A(v)), \tag{5}$$

which shows that the typing extension t_0 integrates smoothly (respecting commonalities) into a typing of all parts of the comprehensive model, such that we end up with a *single typed* comprehensive system: $t : A \to M$. Conditions (4) (compatibility of projections with owner/target) and (5) (compatibility of typing and projections) are visualized in Fig. 9, which shows an excerpt of the complete typed comprehensive system.

4.3. Formal definition of Comprehensive Systems

In this section, we want to develop a precise formal definition of the structures described so far. For this, we resort to the mathematical language of *category theory*. This has several reasons. First of all, category theory allows for very concise definitions due to its abstract nature. Secondly, category theory offers a built-in mechanism called *functor*, which allows to compare two seemingly different formal structures. Finally, triple graphs and graph diagrams, which represent the most directly related formal approach, are formulated in terms of category theory as well. Thus, we can refer to them more easily using the same "language".

This and the following Section 4.4 rely on the categorical concepts *Category*, *Functor*, *Natural Transformation*, *Universal Construction* (Pushouts and Pullbacks), and *Partial Arrow Classifier*. To make this paper self-contained, Appendix A contains a short overview over each of them. For a more detailed presentation, we refer to the introductory textbooks [BW90, Pie91, Wal92].

Intuitively speaking, a category (Definition 8; Appendix A) can be seen as a generalised pre-order or alternatively as a directed graph equipped with a (path) monoid. A category $\mathbb{C}$ comprises *objects* and *morphisms* a.k.a. *arrows*. We write $|\mathbb{C}|$ to denote the class of objects in $\mathbb{C}$ and $Arr_{\mathbb{C}}$ for the class of all morphisms in $\mathbb{C}$. If the class of objects is a set, the respective category $\mathbb{C}$ is called *small*. By convention, objects are denoted in capital letters $(A, B, \ldots)$ and morphisms in small letters $(f, g, \ldots)$. A morphism f is an abstract means to compare two objects $A, B \in |\mathbb{C}|$, which are called *domain* ($\mathrm{dom}(f) = A$) and *codomain* ($\mathrm{codom}(f) = B$) of f. One may think of it as an edge where domain and codomain represent source and target. Hence we will often denote them in an integrated arrow-notation $f : A \to B$. A hom-set $\mathbb{C}(A, B)$ is a subclass of $Arr_{\mathbb{C}}$ and contains all morphisms that have A as domain and B as codomain. In addition to that, there is a unique identity morphism $id_A : A \to A$ for each object A and morphisms $f : A \to B$, $g : B \to C$ with incident domain/codomain (B) can be *composed* to yield a morphism $g \circ f : A \to C$ (spoken "g after f"). Composition is associative and neutral w.r.t. identities.

The most important example of a category is the category of sets and mappings $\mathbb{SET}$. In this category, objects are given by sets and morphisms are given by mappings between sets, i.e. total functions.

Functors (Definition 10; Appendix A.1) are means to compare different categories. They comprise two mappings: One for objects and one for morphisms. Also they must assure that identities and composition are preserved. A functor $G : \mathbb{B} \to \mathbb{SET}$ from a small category $\mathbb{B}$ into the category of sets and mappings $\mathbb{SET}$ is called a *presheaf*. Furthermore, there is a functor category $\mathbb{SET}^{\mathbb{B}}$ (Fact 2; Appendix A.1), which has such functors as objects and morphisms are given by natural transformations (Definition 11; Appendix A.1) between them (think homomorphisms). Presheaves have some interesting properties: From a theoretical point of view they behave similar to objects in $\mathbb{SET}$ [Gol06]. From a more practical point of view they are sufficiently concrete such that one can talk about *elements*: Saying $x \in G$ means that there is some object $s \in |\mathbb{B}|$ such that $x \in G(s)$. They have been called *graph structures* in [Lö93] and are closely related to algebras. A small category $\mathbb{B}$ can be interpreted as a signature with unary operation symbols only. A presheaf G "interprets" every (sort) object $s \in |\mathbb{B}|$ as a set $G(s)$ and every (unary operation) morphism[6] $op : s \to s' \in Arr_{\mathbb{B}}$ as a mapping $G(op) : G(s) \to G(s')$. This is also called *functorial* or *indexed semantics* and $\mathbb{SET}^{\mathbb{B}}$ corresponds to the class of algebras for a signature with unary operations only $\mathbb{B}$ (think instance worlds of a metamodel). This also allows to consider substructures $F \subseteq G$, given by sort-wise subset relations. Categorically, this is represented by an inclusion morphism $F \hookrightarrow G$, which is a special monomorphism (Definition 16; Appendix A.2.2).

Finally, interpreting the diagram in Fig. 6b as the category $\mathbb{B}$ and setting $G := M^1$ (from Fig. 6a), G has the following components: $G(GN) = \{\texttt{Pool}, \texttt{FlowNode}, \texttt{SequenceFlow}, \ldots\}$, $G(GE) = \{\texttt{pool}, \texttt{src}, \texttt{trg}, \ldots\}$, $G(NAE) = \{\texttt{name}, \texttt{type}, \ldots\}$, $G(DN) = \{\texttt{String}, \texttt{ActivityType}, \ldots\}$, together with the respective *owner* and *target* mappings.

Definition 1 (Base Language $\mathbb{B}$ and graph-like structures $\mathbb{G}$). Let $\mathbb{B}$ be a small category called *base (modelling) language*. The base language gives rise to a category of graph-like structures $\mathbb{G} := \mathbb{SET}^{\mathbb{B}}$ (presheaves).

We will now introduce two formal definitions to express our linguistic extension. The first definition is closer to practical implementations, while the second is closer to existing categorical frameworks. Both, formulations will turn out to be equivalent.

4.3.1. Set-based definition

Let us fix a sufficiently large natural number n, the *degree* of the multi-model, and considering a synchronisation scenario with model spaces $(\mathbf{Mod}(M^j_\Phi))_{j\in\{1,\ldots,n\}}$, e.g. BPMN, UML, DMN and so on. As a consequence, we will be regularly working with indices. By convention we will use i and j as index variables, where i runs between $0 \le i \le n$ and j runs between $1 \le j \le n$, if not specified otherwise.

The build-up of a comprehensive system is similar to a graph-like structure (Definition 1) and encompasses local models (components) together with their commonalities (witnesses + projections):

Definition 2 (Comprehensive Systems, Components, Commonalities). A *comprehensive system* C consists of

1. For every $s \in |\mathbb{B}|$ and $0 \le i \le n$, there is a set $C_i(s)$
2. For every $op : s \to s' \in Arr_{\mathbb{B}}$ and $0 \le i \le n$, there is a *total* function $C_i(op) : C_i(s) \to C_i(s')$.
3. For every $s \in |\mathbb{B}|$ and $1 \le j \le n$, there is a *partial* function $p^C_{j,s} : C_0(s) \rightharpoonup C_j(s)$

such that for all $op : s \to s' \in \mathbb{B}$ and $1 \le j \le n$ the following statement holds:

$$\text{If } p^C_{j,s}(x) \text{ is defined, then } p^C_{j,s'}(C_0(op)(x)) \text{ is defined} \tag{6}$$

$$\text{and } p^C_{j,s'}(C_0(op)(x)) = C_j(op)(p^C_{j,s}(x)). \tag{7}$$

The sets $C_j(s)$ together with the total maps $C_j(op)$ constitute the *components*, the sets $C_0(s)$ and total maps $C_0(op)$ constitute the *commonality witnesses*, and the partial functions $p^C_{j,s}$ represent the *projections*.

Note that (6) and (7) generalise the edge-node-incidences, mentioned in Section 4.2, compare(4).

[6] The abbreviation "op" for arrows of the base shall indicate that $\mathbb{B}$-arrows are certain operations constituting the structure of the base language, such as *owner* and *target* operations of edges in graphs.

Definition 3 (Homomorphisms between Comprehensive Systems). Let C, D be comprehensive systems as defined in Definition 2. A *homomorphism between comprehensive systems* is a family

$$(f_{i,s} : C_i(s) \to D_i(s))_{s\in|\mathbb{B}|,0\le i\le n}$$

of mappings compatible with (operation) arrows, i.e. $\forall\, i \in \{0, \dots, n\}, \forall\, op : s \to s' \in Arr_{\mathbb{B}}$:

$$f_{i,s'} \circ C_i(op) = D_i(op) \circ f_{i,s} \tag{8}$$

and compatible with partial (projection) mappings: For all $j \in \{1, \dots, n\}$, $s \in |\mathbb{B}|$ and $x \in C_0(s)$:

$$p^C_{j,s}(x) \text{ is defined } \Rightarrow p^D_{j,s}(f(x)) \text{ is defined and} \tag{9}$$

$$p^D_{j,s}(f(x)) = f(p^C_{j,s}(x)) \tag{10}$$

where we write f instead of $f_{j,s}$, if the indexing becomes clear from the context.

Alternatively, we can visualize Definition 3 by a family of commutative cubes in $\mathbb{SET}$, shown in (11) and indexed by all $op : s \to s' \in \mathbb{B}$ and $1 \le j \le n$. Commutativity of the top and bottom faces encode that the projections in the comprehensive systems C and D fulfil (6)+(7), while left and right faces encode compatibility of f with operation arrows (8), and back and front faces encode compatibility of f with projections (9)+(10). Compare also this formal cube with the example in Fig. 9.

$$\begin{tikzcd}
& D_j(s) \arrow[dl, "D_j(op)"'] & & D_0(s) \arrow[ll, harpoon, "p^D_{j,s}"'] \arrow[dl, "D_0(op)"'] \\
D_j(s') & & D_0(s') \arrow[ll, harpoon, "p^D_{j,s'}"'] & \\
& C_j(s) \arrow[uu, "f_{j,s}"] \arrow[dl, "C_j(op)"'] & & C_0(s) \arrow[ll, harpoon, "p^C_{0,s}"'] \arrow[uu, "f_{0,s}"'] \arrow[dl, "C_0(op)"'] \\
C'_j(s) \arrow[uu, "f_{j,s'}"] & & C_0(s') \arrow[ll, harpoon, "p^C_{j,s'}"'] \arrow[uu, "f_{0,s'}"] &
\end{tikzcd} \tag{11}$$

Definition 3 provides the material for formalising multi-models. A multi-model is a morphism $t : A \to M$ between two comprehensive systems A and M, where M is the correspondence definition, see Fig. 3. In our example M is the alignment of metamodels M^1, M^2, M^3 augmented with type commonalities defined in Listing 3 and partly visualized in Fig. 8. The comprehensive system A typed over M is shown in Fig. 2. Members of A^0 are all dashed circles and $p^A_{j,s}$ assigns to each circle a line end in model A^j, where s is the respective element type (node or edge). The mapping definition of the typing homomorphism t is implicitly given by the concrete syntax and the legend in Fig. 2. See also Fig. 9.

Equations (9) and (10) (f substituted by t) reflect the demanded property (5), i.e. compatibility of commonalities and typing. This can be seen in Fig. 2: the commonality `2` must connect a `class` with a `data object` for instance.

Proposition 1 C̲omprehensive S̲ystems together with their homomorphisms constitute a category $\mathbb{CS}$.

Proof. An identity is a family of identities, composition is composition of mappings $f_{j,s}$. This yields neutrality and associativity. Moreover, composed homomorphisms are still compatible with the inner structure (op,$p_{i,s}$). Whereas this follows in the usual way for $op : s \to s'$, transitivity of the defined-ness implication in (9) also yields compatibility with partial functions. □

4.3.2. Span-based definition

An alternative approach for encoding commonality relations in a multi-model is to use *spans*. This approach was used by the present authors in previous works [KD17, SKLR18]. Its formulation avoids $\mathbb{SET}$-based concepts and is based on the categorical concept of a *diagram*. Recall thatthe semantic interpretation of Listing 3 is a family of

n partial $\mathbb{G}$-morphisms $(m_j : M^0 \rightharpoonup M^j)_{1\leq j\leq n}$. The latter can formally be expressed by a special diagram functor $M : \mathbb{I} \to \mathbb{G}$, where the schema category $\mathbb{I}$ has the star-shape defined in (12) (identity arrows of $\mathbb{I}$ are omitted). Additionally, these diagram functors are subject to the condition that M maps the inner edges $(10, \dots, n0)$ to monomorphisms. This condition is due to a well-known categorical construction [RR88], which expresses partial morphisms as a classes of binary spans (Definition 21; Appendix A.3).

$$\begin{tikzcd}
2 & & & & & & 1 \\
& -2 \arrow[ul, "22"'] \arrow[drr, "20"] & & & & -1 \arrow[ur, "11"] \arrow[dll, "10"'] & \\
\dots & & & 0 & & & \\
& -j \arrow[urr, "j0"] \arrow[dl, "jj"'] & & & & -n \arrow[ull, "n0"'] \arrow[dr, "nn"] & \\
j & & & \dots & & & n
\end{tikzcd} \tag{12}$$

We call these functors multi-span relations because spans are the categorical counterpart of relations.

Definition 4 (Multi-Span Relation). A functor $M : \mathbb{I} \to \mathbb{G}$ where the image of $M(j0)$ for all $1 \leq j \leq n$ is a monomorphism is called a multi-span relation.

Multi-Span Relations are functors, hence we can relate them by natural transformations (families of $\mathbb{G}$-morphisms). The latter are called multi-span relation morphisms.

Definition 5 (Multi-Span Relation Morphism). Let M and N be two multi-span relations. A multi-span relation morphism $f : M \to N$ is a family of $\mathbb{G}$-morphisms, depicted by the j-indexed family of diagrams ($1 \leq j \leq n$) in (13) with the condition that squares (i) and (ii) commute.

$$\begin{tikzcd}
M(0) \arrow[r, "f_{M(0)}"] & N(0) \\
M(-j) \arrow[u, "M(j0)"] \arrow[r, "f_{M(-j)}"] \arrow[d, "M(jj)"'] \arrow[ur, phantom, "(i)"] \arrow[dr, phantom, "(ii)"] & N(-j) \arrow[u, "N(j0)"'] \arrow[d, "N(jj)"] \\
M(j) \arrow[r, "f_{M(j)}"] & N(j)
\end{tikzcd} \tag{13}$$

Proposition 2 Commonality spans together with their morphisms establish a category $\mathbb{M}$.

Proof. Follows immediately from the fact that $\mathbb{M} \subseteq \mathbb{G}^{\mathbb{I}}$ is a full subcategory of the functor category $\mathbb{G}^{\mathbb{I}}$. □

4.3.3. Equivalence of definitions

The following theorem shows the useful fact that the set-based definition in Section 4.3.1 of comprehensive systems and the span-based definition Section 4.3.2 of multi-span relations are equivalent. The span-based definition depicts commonalities *externally* while comprehensive systems *internalise* them. Thus, we may use $\mathbb{M}$ as a drop-in-replacement for $\mathbb{CS}$ and vice versa. The external notion $\mathbb{M}$ turns out to be more easy to handle in the theoretical considerations in Section 4.4 while the internal notion $\mathbb{CS}$ is more closely aligned with the definition of local models (functors into $\mathbb{SET}$) and therefore easier to implement in concrete tools.

Theorem 1 (Equivalence of Categories). $\mathbb{CS} \cong \mathbb{M}$.

Proof. See Appendix B.1. A part of the proof relies on the fact that (small) categories are *cartesian closed*, i.e. there is an equivalence between functor categories $\mathbb{SET}^{\mathbb{B}\times\mathbb{I}} \cong (\mathbb{SET}^{\mathbb{B}})^{\mathbb{I}}$. □
In the following we are only speaking of comprehensive systems, bearing the above equivalence in mind.

4.4. Formal properties

In the following, we investigate the formal properties of comprehensive systems, which demonstrates their theoretical utility as a foundation for multi-modelling. They fulfil all formal requirements for applying existing frameworks for model verification and model transformation.

4.4.1. Consistency verification

Arguably the most important feature in Multi-Model Consistency Management is a means for consistency verification. The diagrammatic constraint framework [RRLW12, RRLW09, DW07, Dis97] demonstrated in Section 4.1 generalises many established verification tools and approaches. To be applicable on a certain class of formal structures, the latter must form a category, which possesses all pullbacks (Definition 15; Appendix A.2.2).

Theorem 2 $\mathbb{CS}$ possesses all pullbacks.

Proof. See Appendix B.2. The proof is carried out component-wise and involves some diagram chasing using the universal property of pullbacks. □

Theorem 2 guarantees that we (theoretically) can apply mature consistency verification methods. We will now demonstrate how to use multiplicities and OCL invariants for implementing CR1–CR5 from Section 2. Here, we will also utilize Theorem 1. The latter allows to "internalize" projections and commonalities, i.e. "flattening" the linguistic extension by interpreting projections and commonalities as edges and nodes. Thus, they can equally be carriers for diagrammatic constraints. Reconsider Fig. 8, this time paying special attention to multiplicities and OCL constraints on the dashed part: The elements of M^0 become regular nodes with edges and attributes. The projections p_j become edges that come with an implicit `0..1` multiplicity at the target side (= partial function). To navigate these elements, the comprehensive systems framework may enhance the OCL library with some helper methods, shown in Listing 4, which allow to navigate projections in a forward (`projection`) and backward (`commonalities`)[7] direction.

Listing 4: Linguistic Extension in OCL

```
-- Retrieves all commonalities for the given element
context OclAny commonalities : Set
-- Retrieves all commonalities of given type for the given element
context OclAny commonalities(commonalityType : Classifier) : Set(T) =
  self->commonalities()->select(c|oclIsTypeOf(commonalityType))
-- Returns the representation in the given component if invoked on a commonality witness
context OclAny projection(component: String) : OclAny
```

The following list explains the *consistency rule implementations (CRIs)* of the rules from Section 2.

CRI1 Is implemented with an OCL-invariant attached to `Activity`, which requires existence of the respective commonality if the activity is a `BUSINESS_RULE`:

```
context Activity inv:
self.type = ActivityType::BUSINESS_RULE implies self->commonalities()->count() = 1
```

CRI2 Is implemented via an `1..1`-multiplicity at the end of projection p_1 on `output` and `input` together with an `1..1`-multiplicity at the source of p_3 on the same elements. The implicit source-edge-incidence guarantees that owner/target relationships are also respected. Furthermore, it is important to note that multiplicities on projections of edges are *conditional* because they depend on other commonalities, i.e. they are only enforced if the respective owner-commonality exists.

CRI3 Is implemented by an OCL-invariant that checks existence of exactly one type of commonality exists:

```
context DataObject inv:
self->commonalities(DataObjectClassImplementation)->count() = 1
xor
self->commonalities(DataObjectCorrespondence)->count() = 1
```

CRI4 Is implemented by `1..1`-multiplicities at the source of the projections p_2 and p_3 at `type` (similar to CRI2).

[7] For readers with knowledge of the Epsilon Transformation Language: `commonalities()` is comparable to `equivalents()`.

CRI5 Is implemented by `1..1`-multiplicities at the source and target of the projections at `DataObjectCorrespondence`. Note that the source of p_1 only has `0..1` because `DataObjects` could alternatively be related via `DataObjectClassImplementation`, see CRI3.

This list exemplifies that already multiplicities are "enough" to model many common consistency rules by intelligently imposing them on projections. To make this mechanism more "user-friendly" one may think of a catalogue of frequent commonality constraints. One example is the `ForAll` [DKPF09] constraint: "For every element of type X there exists a related element of type Y." This translates into an `1..1`-multiplicity at the source of the projection going into X and an `1..1`-multiplicity at the target of the projection going into Y. Another common case is the `PropertyConsistency` constraint: "For two R-related elements x and y the values of the properties x`.p` and y`.q` must be equal". The comprehensive system representation of this constraint will encompass a commonality R having a property, which projects to `p` and `q`. Then, the implicit node-edge-incidence performs the necessary check. An empirical investigation of such common constraints is an interesting future research direction.

However, not all consistency rules can be implemented this way, as seen above. In these cases, one can resort to the expressive power of a constraint language such as OCL to define arbitrary user-defined constraints. Given that one can resort to arbitrary OCL-invariants makes this framework very expressive [MC99], but it lacks a reasoning system. The latter is useful for automatic analysis of inconsistencies [SLO18] and/or automatic consistency restoration [SLO19], which is another interesting direction for future investigations.

4.4.2. Advantages over model merge

Alternatively, we could have tried to formulate CR1–CR5 utilizing model merge [SNL+07]. The latter is often considered to be the standard approach for verifying consistency of multiple related models [KM18, KMCD19]. Formally, model merging can be defined by calculating a *colimit object* [DXC11, KD17, Gog73]: Every object in $\mathbb{M}$ represents a diagram in $\mathbb{G}$ and the colimit object of this diagram is the merged model, a graph $A^+ \in \mathbb{G}$. Intuitively, this result can be described as the union of all components wherein elements related by commonalities are identified. For example, in the merge of models A^1, A^2, A^3 in Fig. 2 the `data object` *"Diagnosis"* $\in A^1$, the `attribute` *"shortDesc"* $\in A^2$ and the `column` *"Diagnosis"* $\in A^3$ will be merged into the same element, say `Diag/descr` of type `DataObjectCorrespondence`.

There are, however, global consistency rules that cannot be realised as a constraint on a merged model. This holds especially for rules, which depend on the knowledge of the membership in local models, because the latter information is lost in the merge.

This can be demonstrated with consistency rule CR1, which relies on the containment of elements (in this case containment in A^1 and A^3). After merging A^1 with A^3 there is only a single node representing *"Select Consultant"* and there is no way of telling if this node had a representation in A^1 and A^3. We only know that it was present somewhere. In contrast, we do not loose this differentiation in comprehensive systems and can successfully check the validity of CR1.

Simultaneously, comprehensive systems can express everything that is expressible with constraints on a merged model, by including respective computations in the verification procedure. Let us reconsider the example from Fig. 1. In the introduction, we mentioned that the trace model (Fig. 1e) is able to uncover the inconsistency just as the merge model (Fig. 1d) does. An OCL-implementation is shown in Listing 5. The central ingredient part is the definition of the derived property `globalSuper`, which aggregates the super-class information for every class over all models: A_1, A_2, A_3. This is done by iterating over all commonalities. This principle of aggregating a property over all related elements can be applied universally. A generic algorithm is described in [KD17, SKLR18]. Finally, the absence of cycles is checked in the invariant `noCycles`, which is based on this derived property.

Listing 5: Simulating Merge Colculations

```
context (A_i)_{i∈{1,2,3}}::Class
def: globalSuper : Set = self->commonalities()->collect(c|c.super)->including(self.super)
inv noCycles: self->closure(globalSuper)->includes(self) = false
```

Thus, comprehensive systems are strictly more expressive than the (naive) model merge approach.

4.4.3. *Transformations*

"Model transformations are the heart and soul of MDE" [SK03]. A mature, widespread and declarative (rule-based) approach to model transformations is given by the graph transformation framework, see Section 3.3. The framework is heavily based on the categorical universal construction of a pushout (Definition 17; Appendix A.2.3). To apply graph transformation to a certain class of structures of interest, one first has to show that they form a so-called *weak adhesive HLR category [EP06] w.r.t.* $\mathcal{M}$, where $\mathcal{M}$ is a special sub-class of *admissible* monomorphisms in the respective category.

Corollary 1 $\mathbb{CS}$ is a weak adhesive HLR category w.r.t. $\mathcal{M}$.

Proving this Corollary requires to verify the existence of pushouts (where some morphisms of the pushout diagram belong to the special class $\mathcal{M}$) in our category $\mathbb{CS}$ (or $\mathbb{M}$ equivalently) and to check whether pushouts have the so-called *(weak) van Kampen* property (Definition 18; Appendix A.2.3) [LS04, EP06]. The latter enforces a well-behaved interplay between pushouts and pullbacks. Yet, Tobias Heindel, in his PhD thesis [Hei10a], showed that it equivalently suffices to show the existence of (i) pushouts along $\mathcal{M}$-morphisms, (ii) $\mathcal{M}$*-partial-arrow classifiers* (Definition 24; Appendix A.3), and that (iii) pushouts are preserved by pullbacks. This is the strategy we are going to use to prove Corollary 1. First, we have to define the class of admissible monos for our category of comprehensive systems. It turns out that we cannot choose *all* monomorphisms: For example, let $(m : A \to B, f : A \to C)$ be a span of $\mathbb{CS}$-morphisms. If there is an incomplete commonality specification in A containing a commonality representative which relates not as many elements as its images in B and C, the pushout construction may produce a commonality specification D, in which the projection is no longer well-defined. This effect has been studied in [KFST19, Ex.6.] as well. Thus, we cannot expect the existence of pushouts in general.

However, we claim that for $\mathcal{M}$ being the class of *reflective* monomorphisms, $\mathbb{CS}$ becomes a respective weakly adhesive HLR category, in particular pushouts along $\mathcal{M}$-morphisms exist.

Definition 6 (Reflective $\mathbb{CS}$-Monomorphisms). Let C, D be two comprehensive systems, a *reflective* $\mathbb{CS}$*-monomorphism* $m : C \rightarrowtail D$ is a family

$$(m_{s,i} : C_i(s) \to D_i(s))_{s\in|\mathbb{B}|, 0\le i\le n}$$

as defined in Definition 3 where every $m_{s,i}$ is injective and, additionally, the implication in (9) is turned into an equivalence. Thus, "defined-ness" of a projection is not only *preserved* but also *reflected*.

Since $\mathbb{CS} \cong \mathbb{M}$, there is an equivalent formulation of this condition in $\mathbb{M}$. An $\mathbb{M}$-monomorphism where additionally the squares (i) in (13), are *pullbacks*, is called a reflective monomorphism.

Think of monomorphisms m as models of *insertion*: When elements that are not in the image of m are thought of as being added by D to the existing context C, then reflective morphisms are not allowed to "make" projections for witnesses that already exist C "defined" in the target D.

Example 1 (Non-reflective $\mathbb{CS}$-Monomorphisms). Let $\mathbb{G} := \mathbb{SET}$, and $n = 2$. Further let L and R be two comprehensive systems with $L_1 = R_1 = \{A\}$, $L_2 = R_2 = \{B\}$, and $L_0 = R_0 = \{C\}$. The projections are defined as follows: $p_1^R(C) = A$, $p_2^L(C) = p_2^R(C) = B$, and $p_1^L(C)$ is undefined. Now let $m : A \to B$ be a comprehensive system morphism that is component-wise the identity. This morphism is monic but not reflective since defined-ness of p_1^R is not reflected.

Example 1 illustrates a non-reflective monomorphism. From a practical point of view, this property prevents the dynamic changes of the arity of a commonality. Next, we have to show that $\mathcal{M}$ is admissible [RR88].

Proposition 3 The class of all reflective monomorphisms is an admissible class $\mathcal{M}$ of monos, i.e.

- it contains all isomorphisms,
- it is closed under composition,
- it is stable under pullback.

Proof. See Appendix B.3. The proof is carried out by diagram chasing and using the universal property of pullbacks. □

Now, one can show the existence of pushouts along $\mathcal{M}$-morphisms, i.e. for spans where one of the legs is a reflective $\mathbb{CS}$-monomorphism.

Theorem 3 $\mathbb{CS}$ has pushouts along $\mathcal{M}$ morphisms.

Proof. See Appendix B.4. The proof is largely carried out by component-wise considerations. The last part however, requires a set-wise consideration to assure that projections are well-defined. □

The next part of the proof, following Heindel's approach, concerns partial arrow-classifiers (Definition 24; Appendix A.3). Intuitively, a partial-arrow classifier adds a substructure to a given object that represents "error" (failed computations or unmappable elements). It is similar to the `java.util.Optional` data type in Java or the `Maybe`-monad in Haskell. In $\mathbb{SET}$, the partial arrow-classifier adds a $\perp$-element to a given set. In the context of van Kampen squares, this construction becomes relevant because it turns out to represent a right-adjoint (Fact 10; Appendix A.3) to the so-called *graphing functor* (Definition 22; Appendix A.3) [Hei10a]. This guarantees that pushouts are *hereditary* (Definition 23; Appendix A.3), a property closely related to the weak van Kampen property [Hei10b], which was originally introduced in [Ken91].

Theorem 4 $\mathbb{CS}$ has *$\mathcal{M}$-partial arrow classifiers.*

Proof. See Appendix 4. The proof is carried out component-wise and uses diagram chasing. □

The final ingredient is stability of pushouts under pullbacks, which corresponds to the "$\Leftarrow$"-direction in the definition of van Kampen squares (Definition 18; Appendix A.2.3).

Theorem 5 Pushouts along $\mathcal{M}$-morphisms in $\mathbb{CS}$ are stable under pullbacks, i.e. when (n, g) is the pushout of (f, m) in (14) and all vertical faces (front, back, left, right) are pullbacks then also (n', g') is the pushout of (f', m').

$$\begin{tikzcd}
& A' \arrow[rr, "m'"] \arrow[dl, "f'"'] \arrow[dd, "a"] & & B' \arrow[dl, "g'"] \arrow[dd, "b"] \\
C' \arrow[rr, "n'"] \arrow[dd, "c"'] & & D' \arrow[dd, "d"] & \\
& A \arrow[rr, "m"] \arrow[dl, "f"'] & & B \arrow[dl, "g"] \\
C \arrow[rr, "n"'] & & D &
\end{tikzcd} \tag{14}$$

Proof. It is straightforward to prove this property from the fact that pushouts in $\mathbb{G}$ are stable under pullbacks [LS04]. Thus, we can apply the fact that pushouts and pullbacks are constructed component-wise, compare proofs of Theorem 2 and Theorem 3, and that stability of pushouts under pullbacks holds for each component in $\mathbb{G}$. □

4.5. Comparison with triple graphs and graph diagrams

In Section 3.4.3, we briefly introduced triple graphs, which are similar to comprehensive systems; both of them are based on graph-like structures and their formulation is given in categorical terms. The original formulation by Schürr [Sch94] was based on directed multi-graphs. It was later reformulated by Ehrig et. al. [EEE$^+$07] in terms of a functor category $\mathbb{G}^{\mathbb{X}}$ and abstracted into the framework of weak adhesive HLR categories [EP06], i.e. $\mathbb{G}$ being an arbitrary weak adhesive HLR category. The schema category $\mathbb{X}_{TGG}$ has the shape of a span, depicted in (15):

$$1 \xleftarrow{01} 0 \xrightarrow{02} 2 \tag{15}$$

Thus, the solution space is limited to binary scenarios. Trollmann and Albayrak [TA15, TA16] generalised the TGG framework to cope with multiple models within a *graph diagram* (GD) framework.

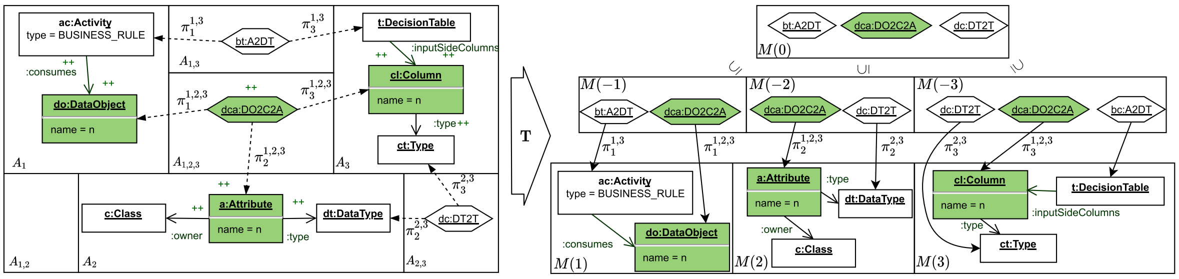


Fig. 10. Graph Diagram production rule

The idea is to allow for different types of schema categories $\mathbb{X}$, which must satisfy the condition that the set of objects can be divided into two disjoint sets of *models* N and *relations* R, i.e. $|\mathbb{X}| = R \sqcup N$. All non-identity morphisms are required to have a domain in R (relations) and codomain in N (models). Further, there is at most one arrow in $Arr_{\mathbb{X}}(r, m)$ for fixed $r \in R$ and $m \in N$. In such a way, graph diagrams, i.e. functors $D : \mathbb{X} \to \mathbb{G}$, can specify relations of different arities. Graph diagrams (GD) subsume TGGs, with $R = \{0\}$ and $N = \{1, 2\}$.

They are, however, *static*: If $r \in R$ has k outgoing morphisms with targets $m_1, ..., m_k \in N$, $D(r)$ is a k-ary correspondence relation with representatives which relate to exactly one element in each of the k models $D(m_j)$. Consequently, the schema category has to change each time a new relation is added!

In the remainder of this section, we show that our framework is more general than graph diagrams $\mathbb{G}^{\mathbb{X}}$ for the case that $\mathbb{G}$ is a presheaf ($\mathbb{G} = \mathbb{SET}^{\mathbb{B}}$) in that there is an embedding functor $\mathbf{T} : \mathbb{G}^{\mathbb{X}} \to \mathbb{CS}$. The latter further preserves pushouts, which model derivations in Graph Diagram Grammars (GDG). Hence we are able to replay all TGG/GDG-computations in our framework, yet being able to cope with new relations *without* changing the schema category, compare Requirement 4 in Section 3.4.3.

In the following, we write $\coprod_{i \in I} D_i$ to denote the coproduct (Definition 3; Appendix A.2.1) of a collection $(D_i)_{i \in I}$ of $\mathbb{G}$-objects. Note that a collection $(f_i : D_i \to D)_{i \in I}$ of morphisms yields the morphism $\coprod_{i \in I} f_i : \coprod_{i \in I} D_i \to D$ by the universal property of coproducts, i.e. the morphism, which acts as f_i on each D_i. Further, we introduce a shorthand notation: $Arr_{\mathbb{C}}(\_, B) := \{f \in Arr_{\mathbb{C}} \mid codom(f) = B\}$.

By Theorem 1, it suffices to define a functor from $\mathbb{G}^{\mathbb{X}}$ to $\mathbb{M}$. The composition of this functor with the equivalence yields the desired result. This functor will also be called $\mathbf{T}$.

Definition 7 (Translation Functor T). Let a schema category $\mathbb{X}$ for graph diagrams be given with $|\mathbb{X}| = R \sqcup N$ and let n be the cardinality of N. Without loss of generality, we assume $N = \{1, \ldots, n\}$. Let D be a graph diagram, then we define a multi-span relation $M := \mathbf{T}(D)$ intuitively as follows (recall the schema in (12)): The model components of $M(j)$ ($j \in N$) are the same as those of D, the commonality specification $M(0)$ is the disjoint union of all relations in D, the middle objects $M(-j)$ are the union of those relations, the model $D(j)$ participates in:

$$\begin{array}{ll} M(j) := D(j) & \text{(Models are untouched)} \\ M(0) := \coprod_{r \in R} D(r) & \text{(Coproduct of all relations)} \\ M(-j) := \coprod_{f \in Arr_{\mathbb{X}}(\_,j)} D(\mathrm{dom}(f)) & \text{(Participating Relations of } D(j)\text{)} \end{array}$$

for all $j \in \{1, \ldots, n\}$. Furthermore,

$$\begin{array}{ll} M(jj) = \coprod_{f \in Arr_{\mathbb{X}}(\_,j)} D(f) & \text{(Projections)} \\ M(j0) : \coprod_{f \in Arr_{\mathbb{X}}(\_,j)} D(\mathrm{dom}(f)) \hookrightarrow \coprod_{r \in R} D(r) & \text{(Domains)} \end{array}$$

Morphisms $M(jj)$ are the unions of the domains of those morphisms that have target $D(j)$ and inclusions arise from the fact that coproducts in the above definition of $M(-j)$ (taken over some relations) are always subgraphs of the complete coproduct $M(0)$ (which is taken over all relations).

The definition of $\mathbf{T}$ on arrows is straightforward and we give it only informally: If $n : D \Rightarrow D'$ is an arrow (natural transformation) between graph diagrams, then (1) $T(n)_i$ is a morphism which acts in the same way as n_i on $D(i)$, if $i > 0$, (2) it amalgamates the actions of n on relations, if $i = 0$, which (3) naturally restricts to the respective actions, if $i < 0$. It is then easy to see, that $T(n)$ is a natural transformation.

We illustrate the construction in Definition 7 at the example of a graph diagram production rule depicted in the left side half of Fig. 10. The figure shows a production rule $r : B \hookrightarrow A$ in an integrated way, where B (before) and A (after) are graph diagrams ($A, B \in \mathbb{G}^{\mathbb{X}}$). A contains all elements shown in Fig. 10 and B contains only those elements, which are not shaded and missing the ++-annotation, compare Fig. 5 in Section 3.4.3. The set of models in $\mathbb{X}$ is a three element set: $N = \{1, 2, 3\}$ representing the three model spaces for BPMN, UML and DMN. The relation set in $\mathbb{X}$ contains four elements: $R = \{(1, 2), (2, 3), (1, 3), (1, 2, 3)$, representing all binary relations between the three model spaces and the ternary relation between all of them. Elements of R are tuples and morphisms in $\mathbb{X}$ are projections $\pi_N^R : R \to N$. This schema is visualised by compartments in Fig. 10, where each compartment depicts a graph (object in $\mathbb{G}$), i.e. the image of $A(x)$ ($B(x)$) for an $x \in |\mathbb{X}|$. We introduce the notation: $G_x := G(x)$ and if x is a tuple we may omit parentheses. The application of the translation functor $\mathbf{T}$ on A will produce a comprehensive system M with degree $n = 3$, which is depicted in the right side half of Fig. 10. The graphs $M(j)$ are identical with A_j $(1 \leq j \leq 3)$. The commonalities graph $M(0)$ is the coproduct (disjoint union) of $A_{1,2}$, $A_{2,3}$, $A_{2,3}$ and $A_{1,2,3}$, i.e. the nodes $\{bt, dca, dc\}$. The domain of definition $M(-1)$ for the projection on component 1 is the coproduct of $A_{1,3}$, $A_{1,2}$ and $A_{1,2,3}$, i.e. the nodes $\{bt, dca\}$. The other domains of definition are constructed accordingly. The mappings $M(j0)$ are simply the inclusion between coproducts, and $M(jj)$ are given by universal coproduct property ($j \in \{1, 2, 3\}$): Taking the union of all arrows from all relations into component j, see Fig. 10.

Theorem 6 Functor $\mathbf{T} : \mathbb{G}^{\mathbb{X}} \to \mathbb{CS}$ is an embedding and preserves pushouts.

Proof. See Appendix 6. □

We obtain as a consequence:

Corollary 2 Every sequence of rule applications in $\mathbb{G}^{\mathbb{X}}$ has a unique representation of corresponding rule applications in $\mathbb{CS}$ and hence can be replayed in the general framework of comprehensive systems.

4.6. Comprehensive systems for consistency management

Finally, we discuss the role of comprehensive systems in the conceptual Multi-Model Consistency Management process introduced in Section 3 and visualised in Fig. 3. Note that the artefacts *Correspondence Definition* and *Multi-Model* as well as the activities *Metamodel Alignment* and *Model Alignment* have a shaded background to highlight the activities that concerned with creation of comprehensive system.

A correspondence definition is built from given metamodels and consistency rules and is formally represented by a comprehensive system M, defined using a suitable DSL such as the one in Listing 3. A multi-model is built from local models and commonalities and is formally represented as a comprehensive system morphism $t : A \to M$, see Section 4.2. The added value of using these artefacts instead of simply working with a collection of models and commonalities (trace model), is that they provide a global view (like model merging), where one can reuse existing means for verification and repair.

Comprehensive systems have a structure similar to those of local models and theoretically they allow to apply existing methods for consistency verification, see Section 4.4.1. In particular, we can use established technologies, such as MOF-based modelling languages to encode comprehensive systems and OCL/EVL to encode consistency rules. A prototype implementation[8] based on EMF has been started.

Moreover, comprehensive systems can be used in different approaches for model repair. On the one hand, using the translation pioneered by Courcelle [Cou97], every graph, typed graph, E-graph and thus also comprehensive system can be translated into first order logic (monadic second order logic). Let C be a comprehensive system and recall Definitions 2 and 3: Every membership $x \in C_i(s)$ becomes a unary predicate `inC_i_S`(x); operation mappings $C_i(op)(x) = y$ become binary predicates `op_C_i`(x, y). The same principle applies for projections $p^C_{j,s}$ and homomorphism components $f_{i,s}$. Additionally, we have to add axioms that force $C_i(op)$ and $f_{i,s}$ to be total functions (left total and right unique), $p^C_{j,s}$ to be partial functions (right unique), as well as the conditions in (6)+(7), (10), and (9)+(8). When consistency rules are encode-able in FOL, we can utilise optimized off-the-shelf SAT/SMT-solvers, e.g. the popular model finder *Alloy* [Jac16], or resolution procedures, such as e.g. *Prolog* [CR96] to perform consistency verification and search-based model repair (finding a model satisfying the formulas). However, it must be noted that this naive translation most likely will run into complexity issues.

[8] https://github.com/webminz/comprehensivesystems-emf-prototype

On the other hand, with Corollary 1 we have opened the door for graph transformation, i.e. rule-based repair. Thus, we can built on existing results w.r.t. verification [HP09] and repair [KR17, SLO19, OPKK18]. Consistency rules may be encoded as a set of consistency-preserving grammar rules. Upon the detection of a consistency violating model modification, the detected edit rule application may be "completed" to an application of a consistency-preserving rule [KKT13, TOLR17, OPKK18] based on the idea of match-consistent splitting [EEE+07]. Furthermore, these rules can be analysed w.r.t. nested graph conditions supported by specialized reasoning tools [LO14, SLO18], which have been shown to outperform generic solutions using off-the-shelf solvers [Pen08]. Reasoning facilities enable various possibilities for investigating model repair in our formal framework, see also [SLO19, HS18], and will therefore play an important role for future work.

As a conclusion, comprehensive systems are not "opinionated" in terms of what means for consistency verification and model repair should be applied on them and we can re-use existing tools and methods.

4.7. Summary and limitations

Comprehensive Systems can be summarized by the slogan "from many models to one model": The issue of dealing with multiple models is addressed by a construction that yields a *single artefact*, on which existing means for consistency verification and model repair can be reused, see Section 4.4. This includes technologies such as MOF/EMF (model representation) and OCL/EVL (model verification) as well as theory and methods such algebraic graph transformation. In the past, the construction of global artefacts was often equated with model merging [SNL+07, BCE+06, RC13, DXC11]. Merging, however, poses some difficulties, especially if the verification of a global constraint depends on knowledge about membership of model elements. In terms of the four requirements stated in Section 3.4, comprehensive systems represent an alternative approach to merging providing a formal construction for expressing multi-models and consistency rules on them, which does not forget the original membership of model elements (Requirement 1), see Section 4.4.2. Comprehensive Systems support general multi-ary ($n \geq 2$) scenarios by definition (Requirement 2) and they formally capture the practical workflow concerning trace models (Requirement 3). The workflow of constructing a domain-specific trace model is mapped to the well-known (meta-) model-instance-pattern, see Section 4.2. Finally, comprehensive systems generalise graph diagrams and triple graphs and allow a flexible introduction and removal of correspondences of different arities (Req. 4), see Section 4.5. Thus, comprehensive systems represent a formal foundation for Multi-Model Consistency Management that combines the practicality of a single artefact from model merging with the flexibility and expressiveness from model weaving, see Section 3.1.2. The construction stresses the utility of *partial* mappings in commonality specifications, which have been promoted in [SKLR18] and were also picked up in [KFST19].

Regarding current limitations of our approach, we first have to state the conceptual restriction that we require the existence of a graph-like universal meta-language. From our experience, this is often the case. However, it might hamper applicability and may require to implement necessary translators or adapters to integrate heterogeneous modelling tools. But, the fact that MOF and EMF/Ecore [SBMP08] are widespread graph-like languages allows diverse applications. The main limitation of our approach is its current lack of practical evidence. A prototype implementation has been started but empirical data w.r.t performance and scalability is missing. Furthermore, comprehensive systems do not provide their own model repair concept and rely on existing solutions. We want to address these challenges in the future.

5. Related and future work

Comprehensive Systems are located in the field of Multi-Model Consistency Management, which was briefly overviewed in Section 3. We highlight the most tightly related studies here:

Triple graphs [Sch94] and its multi-ary variant, graph diagrams [TA15, TA16], are a mature formal framework for multi-model consistency management comprising industry-proven methods for consistency verification and restoration [HEO+11, WFA20, FKM+20]. In Section 4.5, we showed that comprehensive systems are a strict generalisation of triple graphs and graph diagrams.

Model weaving, i.e. using trace models (= commonalities), is often applied in practice. Samimi-Dehkordi et.al. [SDZKR18] use trace models in their implementation of a model synchronisation framework based on Epsilon. Their approach does not encompass a formalisation and does not provide any guarantee for the correctness of the model repair. Feldmann et.al. [FKWVH19] use a similar approach in a multi-domain scenario. They use TGGs as a specification formalism to generate Epsilon code. Thus, the respective consistency rules can only express binary consistency rules. Vitruvius [KKL+21] is a framework for view-based modelling based on a virtual SUM and allows view synchronisation via user-defined expressions. The virtual SUM is created by defining mappings (= type commonalities) between different metamodels. While their expression language is analysed from a theoretical point of view, their proposal of a multi-ary mapping language [KG19], which was also featured in Section 4.2, is missing such a formalisation.

Multi-ary delta lenses (MX-lens) [DKL19] are a formal framework for describing multi-ary model synchronisation. It is defined on a more abstract categorical level than comprehensive systems and comprehensive systems can be considered as a more concrete instantiation of the former. However, MX-lens also comprise propagation-based means for model repair as a built-in feature. Comprehensive System are not directly tied to a specific model repair approach. It is left open whether model-repair should be implemented using a rule-based or a search-based approach.

Stevens [Ste20] proposes another approach to Multi-Model Consistency Management. Her approach takes a workflow-oriented point of view and considers a network of correspondence relations, which are implemented by abstract *builders* that implement consistency verification and restoration. In [Ste20], the correct and optimal scheduling of these builders is analysed. This approach can be considered as a meta-approach that may be combined with other approaches, including comprehensive systems.

The biggest limitation of our approach is *practical evidence*, which is currently lacking. Therefore, our immediate next goal is to provide this missing evidence. We will also have to address the challenge of model repair. Here, the goal must be to re-use as much of existing approaches as possible. With the validity of Corollary 1, we are able to use the algebraic graph transformation framework [EEPT06] and related approaches [HP09, SLO19, KR17]. We aim to built our repair approach on existing rule-based frameworks [KKT13, TOLR17, OPKK18], where consistency is inductively defined via consistency-preserving rules and repair is performed by completing applications of arbitrary edit-rules to consistency preserving rules. For this we have to further investigate the comprehensive system equivalent of match-consistent rule splitting [EEE+07] in triple graph grammars. Theoretical research on admissibility of other graph transformation approaches has already begun: In [KS20], we studied the possibility of using a subclass of comprehensive systems for single pushout rewriting.

Synchronising multiple behavioural models with comprehensive systems is another open issue. In this paper we focused mainly on (more or less static) structural models. Including behavioural semantics into the picture requires to investigate commonalities between the *dynamics* of behavioural models as well.

Finally, analysing the nature of the most common multi-ary consistency rules poses as an interesting research direction, see Section 4.4.1. An example of such a consistency rule is given by the `ForAll`-constraint [DKPF09], which requires the simultaneous existence of a tuple of elements in disparate models. An empirical investigation resulting in a catalogue of such rules is another possible future direction.

A. Categorical background

A structural overview over the contents and the dependencies between the individual sections of the Appendix is given in Figure 11. This first appendix, section A briefly summarizes the categorical background that is required for this paper. A more in-depth introduction can be found in textbooks such as [BW90, Pie91, Wal92]. The second appendix section B contains detailed proofs of the Theorems and Propositions in this paper.

A *category* is a collection of similarly-structured mathematical objects equipped with means to "compare" these objects:

Definition 8 (Category). A category $\mathbb{C}$ consists of the following:

- A class of *objects* $|\mathbb{C}|$.
- For every pair of objects $A, B \in |\mathbb{C}|$, a class of *morphisms* called a *hom-set* $\mathbb{C}(A, B)$. For every member $f \in \mathbb{C}(A, B)$, $A = \text{dom}(f)$ is called the *domain* of f and $B = \text{codom}(f)$ is called the codomain of f. The class of all morphisms (union of all hom-sets) is denoted by $Arr_{\mathbb{C}}$.
- For every object $A \in |\mathbb{C}|$ there exists a unique *identity* morphism $id_A \in \mathbb{C}(A, A)$.
- For every triple of objects $A, B, C \in |\mathbb{C}|$ and morphisms $f \in \mathbb{C}(A, B)$ and $g \in \mathbb{C}(B, C)$, there exists a composite $g \circ f \in \mathbb{C}(A, C)$.

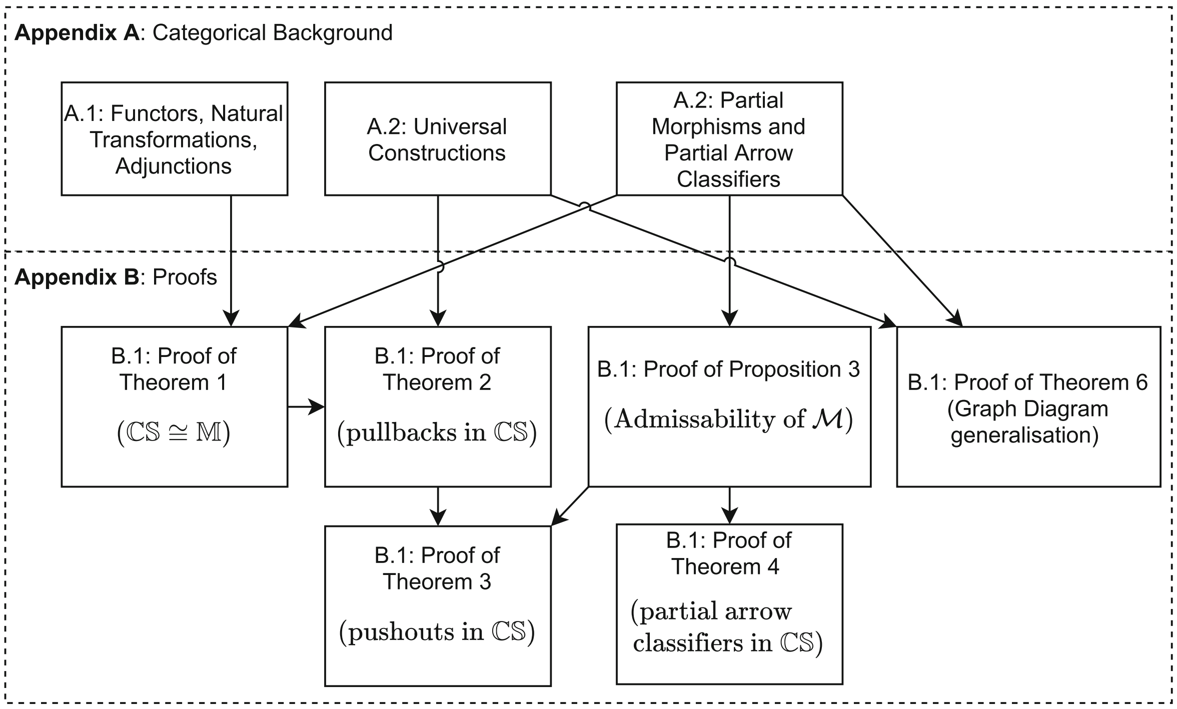

Appendix A: Categorical Background
A.1: Functors, Natural Transformations, Adjunctions
A.2: Universal Constructions
A.2: Partial Morphisms and Partial Arrow Classifiers
Appendix B: Proofs
B.1: Proof of Theorem 1 ($\mathbb{CS} \cong \mathbb{M}$)
B.1: Proof of Theorem 2 (pullbacks in $\mathbb{CS}$)
B.1: Proof of Proposition 3 (Admissability of $\mathcal{M}$)
B.1: Proof of Theorem 6 (Graph Diagram generalisation)
B.1: Proof of Theorem 3 (pushouts in $\mathbb{CS}$)
B.1: Proof of Theorem 4 (partial arrow classifiers in $\mathbb{CS}$)

Fig. 11. Appendix Structure and Dependencies

such that

- Composition $\circ$ is *neutral* w.r.t. identities, i.e. for all $f \in \mathbb{C}(A, B)$:
$$id_B \circ f = f = f \circ f \circ id_A \tag{16}$$
- Composition $\circ$ is *associative*, i.e. for all $f \in \mathbb{C}(A, B)$, $g \in \mathbb{C}(B, C)$, and $h \in \mathbb{C}(C, D)$:
$$(h \circ g) \circ f = h \circ (g \circ f) \tag{17}$$

Due to the abstract nature of categories, it is often not possible to check if two objects represent the same thing because we cannot look *into* the internal structure objects. However, we can compare them via morphisms and if two objects are related by invertible morphisms, they are called *isomorphic*, i.e. identical modulo internal renaming.

Definition 9 (Isomorphism). Let $\mathbb{C}$ be a category and $A, B \in |\mathbb{C}|$ two objects in this category. A and B are *isomorphic*, written $A \approx B$, if there exist two morphisms $i : A \to B \in Arr_{\mathbb{C}}$ and $i^{-1} : B \to A \in Arr_{\mathbb{C}}$ such that $id_A = i^{-1} \circ i$ and $id_B = i \circ i^{-1}$. Further, i and i^{-1} are then called isomorphisms.

Thus, in category theory many construction are only unique "up to isomorphism".

Arguably, the most important category is the category of sets and mappings.

Fact 1 (Category $\mathbb{SET}$). There is a category $\mathbb{SET}$, whose class of objects is the class of all sets. The class of morphisms is given by the class of all total mappings between sets. Identities are given by identical mappings and composition is given by function composition.

A.1. Functors, natural transformations, adjunctions

A *functor* represents a means to "compare" two categories.

Definition 10 (Functor). Let $\mathbb{C}$ and $\mathbb{D}$ be two categories. A functor $F : \mathbb{C} \to \mathbb{D}$ comprises,

- an object mapping, i.e. for every object $A \in |\mathbb{C}|$ in the source category, F assigns an object $F(A) \in \mathbb{D}$ in the target category,

- and a morphism mapping, i.e. for every morphism $f : A \to B$, F assigns a morphism $F(f) : F(A) \to F(B)$ in the target category,

such that

- identities are mapped to identities, i.e. for all $A \in |\mathbb{C}|$: $F(id_A) = id_{F(A)}$.
- and composition is preserved, i.e. for all $f \in \mathbb{C}(A, B)$ and $g \in \mathbb{C}(B, C)$: $F(g \circ f) = F(g) \circ F(f)$[9].

F is called an *embedding*, if it is injective on objects of $\mathbb{C}$ and injective on $\mathbb{C}(A, B)$ for all $A, B \in |\mathbb{C}|$.

A *natural transformation* is a means to "compare" functors.

Definition 11 (Natural Transformation). Let $F : \mathbb{C} \to \mathbb{D}$ and $G : \mathbb{C} \to \mathbb{D}$ be two functors between the same categories. A natural transformation $\alpha : F \Rightarrow G$ is given by a $|\mathbb{C}|$-indexed family of $\mathbb{D}$-morphisms $((\alpha_A : F(A) \to G(A)_{A\in|\mathbb{C}|}$, such that for every $f \in \mathbb{C}(A, B)$ the following diagram commutes:

$$\begin{tikzcd} F(A) \arrow[r, "F(f)"] \arrow[d, "\alpha_A"'] & F(B) \arrow[d, "\alpha_B"] \\ G(A) \arrow[r, "G(f)"] & G(B) \end{tikzcd} \tag{18}$$

The diagrams are also known as *naturality squares*.

Functors and natural transformations organise themselves into a category:

Fact 2 (Functor Category.) For every pair of categories $\mathbb{C}$ and $\mathbb{D}$, There exists a *functor category* $\mathbb{D}^{\mathbb{C}}$, whose objects are the functors between $\mathbb{C}$ and $\mathbb{D}$ morphisms are given by the natural transformations between these functors.

Functors and natural transformations allow us to check whether two classes of mathematical structures are essentially "the same":

Definition 12 (Equivalence of Categories). Let $\mathbb{C}$ and $\mathbb{D}$ be two categories. They are said to be *equivalent*, written $\mathbb{C} \cong \mathbb{D}$, if there exists a pair of functors $R : \mathbb{C} \to \mathbb{D}$ and $L : \mathbb{D} \to \mathbb{C}$ together with two natural transformations $\approx_{\mathbb{C}}: L \circ R \Rightarrow 1_{\mathbb{C}}$ and $\approx_{\mathbb{D}}: R \circ L \Rightarrow 1_{\mathbb{D}}$, where $1_{\mathbb{C}}$ and $1_{\mathbb{D}}$ denote identity functors (= identity in all components) and all members of $\approx_{\mathbb{C}}$ ($\approx_{\mathbb{D}}$) are isomorphisms in $\mathbb{C}$ ($\mathbb{D}$).

If these families of isomorphisms are actually identities, then $\mathbb{C}$ and $\mathbb{D}$ are said to be isomorphic.

Moreover, in category theory there is a weaker notion than equivalence, called *adjunction*[10]. Intuitively speaking it means that two classes of structures are equivalent modulo some free construction that can be universally applied. An example for such a construction is the free monoid A^* (Kleene star) over a set A.

Definition 13 (Adjunctions, (Co)-Free constructions). Let $\mathbb{C}$ and $\mathbb{D}$ be two categories and $R : \mathbb{C} \to \mathbb{D}$, $L : \mathbb{D} \to \mathbb{C}$ be two functors between them. L and R are said to be adjoint, written $L \dashv R$ if there exists two natural transformations $\eta : 1_{\mathbb{D}} \Rightarrow R \circ L$ (called unit) and $\varepsilon : L \circ R \Rightarrow 1_{\mathbb{C}}$ (called co-unit).

Equivalently, an adjunction can be defined as co-free construction w.r.t to a functor $L : \mathbb{D} \to \mathbb{C}$. A co-free constructions assigns to every $\mathbb{C}$-object B a $\mathbb{D}$-object $R(B)$ and $\mathbb{C}$-morphism $\varepsilon_B : L(R(B)) \to B$ such that for every $\mathbb{D}$-object A and $\mathbb{C}$-morphism $f : L(A) \to B$ there exists a unique morphism $\overline{f} : A \to R(B)$ such that $f = \varepsilon_B \circ L(\overline{f})$. This is summarized in the diagram in (19).

[9] Note, that the composition $\circ$ in the left hand side of the identity is the composition of $\mathbb{C}$, while the composition in the right hand side is the composition of $\mathbb{D}$.

[10] It is sometimes noted that this notion is the actual reason category theory was invented

$$\begin{tikzcd}
\mathbb{C} & & & & \mathbb{D} \\
B & L(A) \arrow[l, "\forall f"'] \arrow[dl, dashed, "L(\bar{f})"] & & & & A \arrow[dl, dashed, "\exists!\bar{f}"'] \\
L(R(B)) \arrow[u, "\varepsilon_B"] & & & & R(B) &
\end{tikzcd} \tag{19}$$

A.2. Universal constructions

Universal constructions have proven to be important for many software theoretical methods. Intuitively universal constructions can be described as a generalisation of *meets* and *joins* in a pre-order. Some well known examples for universal constructions in $\mathbb{SET}$ are cartesian products or disjoint unions (coproduct). It is important to note that $\mathbb{SET}$ possesses all these universal constructions and thus every category $\mathbb{SET}^{\mathbb{B}}$ does as well [Gol06]. The construction of universal constructions in those categories is carried out "pointwise". We say that a universal object is constructed "pointwise" in $\mathbb{SET}^{\mathbb{B}}$, if it is constructed separately for each $\mathbb{B}$-object, e.g. in the case of E-Graphs separately for the set of graph nodes, the set of attribute nodes, the set of graph edges, and the set of node attribute edges. The universal properties of the universal constructions then guarantee, that the resulting object is a well-defined object in $\mathbb{SET}^{\mathbb{B}}$. Examples are given in the proofs of Lemma 1, Lemma 2 and Lemma 3. For more details on this idea, we refer to [Gol06].

A.2.1. Coproducts

Coproducts a.k.a. *sums* provide means to collect a set of objects and work with them uniformly, similar to type abstraction in programming.

Definition 14 (Binary Coproduct). Let $\mathbb{C}$ be a category and $A, B \in \mathbb{C}$ be objects. A binary *coproduct* of A and B is given by an object $A + B$ and two coproduct *injection* morphisms $\iota_A : A \to A + B$ and $\iota_B : B \to A + B$ such that for all pairs of $\mathbb{C}$-morphisms $f : A \to C$ and $g : B \to C$ with $C \in \mathbb{C}$ there exists a unique morphism $[f;\ g] : A + B \to C$ such that $[f;\ g] \circ \iota_A = f$ and $[f;\ g] \circ \iota_B = g$, visualized in the diagram in (20):

$$\begin{tikzcd}
 & C & \\
 & A + B \arrow[u, dotted, "{[f,g]!}"] & \\
A \arrow[ur, "\iota_A"] \arrow[uur, dashed, bend left, "f"] & & B \arrow[ul, "\iota_B"'] \arrow[uul, dashed, bend right, "g"']
\end{tikzcd} \tag{20}$$

The mediating morphism $[f;\ g]$ acts like f and g via case distinction. If a category $\mathbb{C}$ has coproducts of arbitrary arity then there is a special nullary coproduct, the initial object 0, that has unique morphisms $0_A : 0 \to A$ into every object $A \in |\mathbb{C}|$ and it is neutral w.r.t. binary coproducts, i.e. $A + 0 \cong A$. A multi-ary coproduct $\coprod$ is then given by multiple applications of the binary coproduct operator, because the latter are associative $((A_1 + A_2) + A_3 \cong A_1 + (A_2 + A_3))$ and commutative $(A_1 + A_2 \cong A_2 + A_1)$ up to isomorphism. The (multi-ary) coproduct over an I-indexed family of $\mathbb{C}$-objects $(A_i)_{i\in I}$ is denoted $(\coprod_{i\in I} A_i, (\iota_i : A_i \to \coprod_{i\in I} A_i)_{i\in I})$ and the mediating morphism for a family of morphisms $(f_i : A_i \to C)_{i\in I}$ by $\coprod f_i : \coprod_{i\in I} A_i \to C$.

Fact 3 (Coproducts in $\mathbb{SET}$.) $\mathbb{SET}$ has all coproducts. A binary coproduct in $\mathbb{SET}$ is given by disjoint union $A \uplus B := \{(i, x) \mid (x \in A \wedge i = 1) \vee (x \in B \wedge i = 2)\}$ for A and B being sets. The initial object 0 in $\mathbb{SET}$ is the empty set $\emptyset$.

Lemma 1 (Coproducts in $\mathbb{SET}^{\mathbb{B}}$.) Every functor category $\mathbb{SET}^{\mathbb{B}}$ has coproducts due to the fact that $\mathbb{SET}$ has all coproducts and we can construct them pointwise.

Proof. Let F and G be two functor objects in $\mathbb{SET}^{\mathbb{B}}$ and consider the family of diagrams in (21), whichindexed by $f : A \to B \in Arr_{\mathbb{B}}$

$$\begin{tikzcd}
F(A) \arrow[dd, "\mathbf{F}(f)"'] \arrow[dr, "\iota_{F(A)}"] & & G(A) \arrow[dl, "\iota_{G(A)}"'] \arrow[dd, "G(f)"] \\
& F(A)+G(A) \arrow[dd, dotted, "F(f)+G(f)!"] & \\
F(B) \arrow[dr, "\iota_{F(B)}"] & & G(B) \arrow[dl, "\iota_{G(B)}"'] \\
& F(B)+G(B) &
\end{tikzcd} \tag{21}$$

The coproduct of F and G for objects A, B is given by constructing the respective coproducts $F(A) + G(A)$ and $F(B) + G(B)$ in $\mathbb{SET}$, the morphism mappings $(F + G)(f)$ (dotted line) arises uniquely from the universal property of coproducts. $\square$

A.2.2. Pullbacks

A pullback can be seen as the categorical version of an *inner join*: two structures A and B are combined where they coincide on a common structure C.

Definition 15 (Pullback). Let $\mathbb{C}$ be category and a co-span of $\mathbb{C}$-morphisms $A \overset{a}{\to} C \overset{b}{\leftarrow} B$ be given. The pullback of a and b is given by the span $A \overset{\pi_A}{\leftarrow} A \times_{(a,b)} B \overset{\pi_B}{\to} B$ such that $a \circ \pi_A = b \circ \pi_B$ and for all pairs of $\mathbb{C}$-morphisms $f : D \to A$ and $g : D \to B$ such that $b \circ g = a \circ f$ and there exists a unique morphism $\langle f, g\rangle : D \to A \times_{(a,b)} B$ such that $\pi_A \circ \langle f, g\rangle = f$ and $\pi_B \circ \langle f, g\rangle = g$, visualized in (22):

$$\begin{tikzcd}
D \arrow[dr, dotted, "\langle f,g\rangle !"] \arrow[drr, dashed, bend left, "g"] \arrow[ddr, dashed, bend right, "f"'] & & \\
& A \times_{(a,b)} B \arrow[r, "\pi_B"] \arrow[d, "\pi_A"'] \arrow[dr, phantom, "p.b."] & B \arrow[d, "b"] \\
& A \arrow[r, "a"'] & C
\end{tikzcd} \tag{22}$$

Fact 4 (Pullbacks in $\mathbb{SET}$.) $\mathbb{SET}$ has all pullbacks: Given two mappings $f : A \to C$ and $g : B \to C$ with same codomain the pullback $A \times_{(f,g)} B$ is given by the fibred product $A \times_{(f,g)} B := \{(a, b) \mid a \in A, b \in B, f(a) = g(b)\}$.

Lemma 2 (Pullbacks in $\mathbb{SET}^{\mathbb{B}}$.) Every functor category $\mathbb{SET}^{\mathbb{B}}$ has pullbacks due to the fact that $\mathbb{SET}$ has all pullbacks and we can construct them pointwise.

Proof. Let F, G and H be objects in $\mathbb{SET}^{\mathbb{B}}$ and $\nu : F \Rightarrow H$ and $\mu : G \Rightarrow H$ morphisms in $\mathbb{SET}^{\mathbb{B}}$. Consider the following cube for some $f : A \to B \in Arr_{\mathbb{B}}$:

$$\begin{tikzcd}
 & F(A) \times_{(\mu,\nu)} G(A) \arrow[rr, "\nu'_A"] \arrow[dl, "\mu'_A"'] \arrow[dd, dotted, "F\times_{(\mu,\nu)} G(f)!"] & & G(A) \arrow[dl, "\mu_A"] \arrow[dd, "G(f)"] \\
F(A) \arrow[rr, "\nu_A"] \arrow[dd, "F(f)"'] & & H(A) \arrow[dd, "H(f)"] & \\
 & F(B) \times_{(\mu,\nu)} G(B) \arrow[rr, "\nu'_B"] \arrow[dl, "\mu'_B"'] & & G(B) \arrow[dl, "\mu_B"] \\
F(B) \arrow[rr, "\nu_B"'] & & H(B) &
\end{tikzcd}$$

The pullback of μ and ν for objects A and B is given by constructing the respective pullbacks $F(A) \times_{(\mu,\nu)} G(A)$ and $F(B) \times_{(\mu,\nu)} G(B)$ in $\mathbb{SET}$ along (μ_A, ν_A) and (μ_B, ν_B) respectively, the morphism mapping $(F \times_{(\mu,\nu)} G)(f)$ (dotted line) arise uniquely from the universal property of the pullbacks in the bottom face of the cube. □

Definition 16 (Monomorphism). A morphism $m : A \to B$ is called a *monomorphism* iff the pullback of m and m coincides with id_A, see (23).

$$\begin{tikzcd}
A \arrow[r, "id_A"] \arrow[d, "id_A"'] \arrow[dr, phantom, "p.b."] & A \arrow[d, "m"] \\
A \arrow[r, "m"'] & B
\end{tikzcd} \tag{23}$$

In this case m has the left cancellation property, which is a consequence of the pullback property in (23):

$m \circ f = m \circ g \Rightarrow f = g$

We sometimes highlight the special property of m by denoting it with a special arrow $m : A \rightarrowtail B$.

Fact 5 (Monomorphism in $\mathbb{SET}$). In $\mathbb{SET}$ the class of monomorphisms is exactly the class of *injective mappings*.

Fact 6 (Pullbacks preserve Monos). If a is a monomorphism in the diagram of Def. 15, then π_B is a monomorphism, as well.

A.2.3. Pushouts

A pushout can intuitively be described as *gluing* of two structures at a defined interface.

Definition 17 (Pushout). Let $\mathbb{C}$ be a category and a span of $\mathbb{C}$-morphisms $A \overset{a}{\leftarrow} C \overset{b}{\rightarrow} B$ be given. The pushout of a and b is given by the co-span $A \overset{\iota_A}{\rightarrow} A +_{(a,b)} B \overset{\iota_B}{\leftarrow} B$ such that $\iota_A \circ a = \iota_B \circ b$ and for all pairs $f : A \to D$ and $g : B \to D$ there exists a unique morphism $[f;\ g] : A +_{(a,b)} B \to D$ such that $[f;\ g] \circ \iota_A = f$ and $[f;\ g] \circ \iota_B = g$, visualized in (24):

$$\begin{tikzcd}
C \arrow[r, "b"] \arrow[d, "a"'] \arrow[dr, phantom, "p.o."] & B \arrow[d, "\iota_B"] \arrow[ddr, dashed, bend left, "g"] & \\
A \arrow[r, "\iota_A"'] \arrow[drr, dashed, bend right, "f"'] & A +_{(a,b)} B \arrow[dr, dotted, "{[f,g]!}"] & \\
 & & D
\end{tikzcd} \tag{24}$$

Fact 7 (Pushouts in $\mathbb{SET}$). $\mathbb{SET}$ has all pushouts: Given two mappings $f : C \to A$ and $g : C \to B$ with same domain, consider a relation $\sim$ on $A \uplus B$, defined as follows (ι_A and ι_B are the embeddings into the disjoint union

$A \uplus B)$

$$a \sim b \text{ iff } \exists\, c \in C : \iota_A(f(c)) = a \wedge \iota_B(g(c)) = b$$

and $\equiv$ the least equivalence relation containing $\sim$, then the pushout of f and g is given by $A +_{(f,g)} B := (A \uplus B)/_{\equiv}$.

Lemma 3 (Pushouts in $\mathbb{SET}^{\mathbb{B}}$.) Every functor category $\mathbb{SET}^{\mathbb{B}}$ has pushouts due to the fact that $\mathbb{SET}$ has all pushouts and we can construct them pointwise.

Proof. Dual to the proof of Lemma 2. □

Pushouts play an integral role in the algebraic graph transformation framework [EEPT06], i.e. rule-based rewriting is represented by pushout-diagrams in a suitable category. These categories are referred to as *adhesive* categories and their definition is based on the so-called Van-Kampen property [LS04]:

Definition 18 (Van Kampen square). A pushout square (f, m, n, g) as shown in the bottom of (25)

$$\begin{tikzcd}
& A' \arrow[rr, "m'"] \arrow[dl, "f'"'] \arrow[dd, "a" near start] & & B' \arrow[dl, "g'"] \arrow[dd, "b"] \\
C' \arrow[rr, "n'" near start, crossing over] \arrow[dd, "c"'] & & D' & \\
& A \arrow[rr, "m" near start] \arrow[dl, "f"'] & & B \arrow[dl, "g"] \\
C \arrow[rr, "n"'] & & D \arrow[from=uu, "d" near start, crossing over] &
\end{tikzcd} \tag{25}$$

is called a *Van Kampen* square iff

$$\text{back faces are pullbacks} \Rightarrow (\text{front faces are pullbacks} \Leftrightarrow \text{top face is pushout}) \tag{26}$$

Definition 19 (Adhesive Category). A category $\mathbb{C}$ is called *adhesive* iff

- $\mathbb{C}$ has all pullbacks,
- $\mathbb{C}$ has pushouts along monomorphisms (i.e. for spans where at least one leg is a monomorphism) which also have the Van Kampen property (Definition 18).

A more general and more widespread notion than adhesive categories is given by so-called weak adhesive HLR categories, which also include practically relevant structures such as attributed graphs. This definition weakens both the notion of Van Kampen squares and requires the existence of weak Van Kampen pushouts only along for an admissible sub class of all monomorphism $\mathcal{M}$. The latter is explicated further in Section A.3.

Definition 20 (Weak Adhesive HLR Category). Let $\mathcal{M}$ be an admissible class of monomorphisms. A category $\mathbb{C}$ is called a *weak adhesive HLR (High Level Replacement)* category iff

- $\mathbb{C}$ has pushouts and pullbacks along $\mathcal{M}$-morphisms and
- $\mathbb{C}$ pushouts along $\mathcal{M}$-morphisms have the weak Van Kampen property, i.e. (26) is only required to hold for commutative situations where $m \in \mathcal{M}$ or $b, c, d \in \mathcal{M}$, cf. the diagram in (25).

A.2.4. Universal constructions and adjunctions

Fact 8 If $L \dashv R$ are two adjoint functors, then L preserves coproducts and pushouts and R preserves pullbacks.

A.3. Partial morphisms and partial arrow classifiers

The category $\mathbb{SET}$ denoted the category of sets and *total* functions. There is also a strict super-category $\mathbb{SET} \subset \mathbb{PSET}$ of sets and *partial* functions (every total function is a special partial function). Here, we present a a generic and category-independent approach to construct a category $Par(\mathbb{C})$ of partial morphisms over a given category

$\mathbb{C}$, whose morphisms are called total. This well-known approach only requires $\mathbb{C}$ to have all pullbacks and was introduced in [RR88].

The category $Par(\mathbb{C})$ is a subcategory of the span category $Span(\mathbb{C})$ over $\mathbb{C}$, where the inner legs of these spans are required to be monomorphisms:

Definition 21 (Partial Map category $Par_{\mathcal{M}}(\mathbb{C})$). The constituents of $Par(\mathbb{C})$ are defined as follows:

- The class of objects coincides with class of objects in the base category, i.e. $|Par(\mathbb{C})| = |\mathbb{C}|$
- Morphisms $[m, f\rangle : A \rightharpoonup B \in Arr_{Par(\mathbb{C})}$ are equivalence classes of spans in $\mathbb{C}$:

$$\begin{tikzcd} A & X \arrow[l, tail, "m"'] \arrow[r, "f"] & B \\ & X' \arrow[ul, tail, "m'"] \arrow[u, "\cong"'] \arrow[ur, "f'"'] & \end{tikzcd} \tag{27}$$

  Consider (27): a span (m, f) where m is a monomorphism can be seen as a representative of a partial map from A to B. Isomorphisms between apexes of these spans $\cong$: $X' \to X$ that are compatible with both legs (i.e. $m' = m\circ \cong$ and $f' = f\circ \cong$) generate an equivalence relation and we write $[m, f\rangle$ to denote one equivalence class where (m, f) is a chosen representative. In $\mathbb{SET}$-based categories, it is natural to choose m to be the inclusion $dom(f) \subseteq A$ as the chosen representative.
- Identities in $Par(\mathbb{C})$ are those equivalence classes of spans, whose representatives are identities in $\mathbb{C}$
- Composition of two partial morphisms $[m, f\rangle : A \rightharpoonup B$ and $[n, g\rangle : B \rightharpoonup C$ is defined via pullback:

$$\begin{tikzcd} & & B & & \\ A & X \arrow[l, tail, "m"'] \arrow[ur, "f"] & p.b. & Y \arrow[ul, tail, "n"'] \arrow[r, "g"] & C \\ & & X \times_{(n,f)} Y \arrow[ul, tail, "n'"] \arrow[ur, "f'"'] & & \end{tikzcd} \tag{28}$$

  i.e. the composition $[n, g\rangle \circ [m, f\rangle$ is given by $[m \circ n', g \circ f'\rangle$ and it can be shown that the choice of representative for this span is unique up to isomorphism.

Neutrality w.r.t. identities and associativity of composition results from the fact that pullbacks preserve isomorphisms and the universal property of pullbacks.

The construction of partial map categories can further be restricted by replacing the class of all monomorphisms with an admissible subclass $\mathcal{M}$ of all monomorphisms, called *dominion* in [RR88]. To be considered admissible, this class $\mathcal{M}$ must allow the constructions shown in (27) and (28), i.e. it is closed under isomorphisms, its is closed under composition, and stable under pullback, see Proposition 3. In this case, we call the respective category $Par_{\mathcal{M}}(\mathbb{C})$ an $\mathcal{M}$-partial map category.

The category $\mathbb{C}$ is embedded into $Par(\mathbb{C})$[11] by the so called *graphing functor*, which is the identity on objects and maps every morphism $f : A \to B$ to the span $[id_A, f\rangle$, where the identity id_A on A is trivially a monomorphism.

Definition 22 (Graphing Functor $\Gamma_{\mathcal{M}}$).

$$\Gamma_{\mathcal{M}} := \begin{cases} \mathbb{C} \to Par_{\mathcal{M}}(\mathbb{C}) \\ A \mapsto A \\ f : A \to B \mapsto [id_A, f\rangle : A \rightharpoonup B \end{cases} \tag{29}$$

Pushouts in $\mathbb{C}$ that remain pushouts in $Par_{\mathcal{M}}(\mathbb{C})$ after embedding them via $\Gamma_{\mathcal{M}}$ are called *hereditary* [Ken91]. They are closely related to Van Kampen Squares [Hei10b].

[11] We omit the index $\mathcal{M}$ if $\mathcal{M}$ is equal to the class of all $\mathbb{C}$-monomorphisms.

Definition 23 (($\mathcal{M}$)-Hereditary Pushout). Let (m', f') be the pushout of (f, m) in $\mathbb{C}$ displayed in (30). It is called ($\mathcal{M}$)-*hereditary* iff after embedding in $Par_{\mathcal{M}}(\mathbb{C})$ via $\Gamma_{\mathcal{M}}$, the square $\Gamma_{\mathcal{M}}(f') \circ \Gamma_{\mathcal{M}}(m) = \Gamma_{\mathcal{M}}(m') \circ \Gamma_{\mathcal{M}}(f)$ is a pushout in $Par_{\mathcal{M}}(\mathbb{C})$ as well.

Or, equivalently [Hei10b]: The pushout (m', f') of (f, m) is called hereditary iff for any commutative situation shown in (30) where the left and back faces are pullback and $a, b, c \in \mathcal{M}$ it holds that the bottom face is a pushout if and only if (1) the two front faces are pullbacks and (2) $d \in \mathcal{M}$.

$$\begin{tikzcd}
& A \arrow[rr, "m"] \arrow[dl, "f"'] & & B \arrow[dl, "f'"] \\
C \arrow[rr, "m'"'] & & D & \\
& A' \arrow[uu, tail, "a"] \arrow[rr, "n"] \arrow[dl, "g"'] & & B' \arrow[uu, tail, "b"'] \arrow[dl, "g'"] \\
C' \arrow[uu, tail, "c"] \arrow[rr, "n'"'] & & D' \arrow[uu, tail, "d"'] &
\end{tikzcd} \tag{30}$$

Hereditaryness is immediately given, when $\Gamma_{\mathcal{M}}$ has a right adjoint and therefore preserves colimits. The foundation for this right adjoint are ($\mathcal{M}$)-partial arrow classifiers, which "totalize" partial morphisms:

Definition 24 (($\mathcal{M}$)-Partial Arrow Classifier). Let B be an object in $\mathbb{C}$. A ($\mathcal{M}$-)partial arrow classifier for B is given by a monomorphism ($\mathcal{M}$-morphisms) $\eta_B : B \rightarrowtail \mathcal{L}B$ such that for ($\mathcal{M}$)-partial morphism span $[m, f\rangle : A \rightharpoonup B$ there exists a unique morphism $\overline{[m, f\rangle} : A \to \mathcal{L}B$ (the *totalization*) such that the resulting square (31) is a pullback:

$$\begin{tikzcd}
X \arrow[r, tail, "m"] \arrow[d, "f"'] \arrow[dr, phantom, "p.b."] & A \arrow[d, dashed, "\overline{[m,f\rangle}!"] \\
B \arrow[r, "\eta_B"'] & \mathcal{L}B
\end{tikzcd} \tag{31}$$

Fact 9 $\mathbb{SET}$ and $\mathbb{SET}^{\mathbb{C}}$ have partial arrow classifiers. In $\mathbb{SET}$, $\mathcal{L}$ adds a new $\perp$-element to every set and turns a partial function into a total function by mapping all non-mapped elements to $\perp$. In $\mathbb{SET}^{\mathbb{B}}$, the construction becomes more involved, see [Gol06, pp.202-210].

Fact 10 In a category with $\mathcal{M}$-partial arrow classifiers, $\mathcal{L}$ extends to a functor that is right adjoint to $\Gamma_{\mathcal{M}}$ and defined as follows:

$$\mathcal{L} := \begin{cases} Par_{\mathcal{M}}(\mathbb{C}) \to \mathbb{C} \\ A \mapsto \mathcal{L}A \\ [m, f\rangle : A \rightharpoonup B \mapsto \overline{[\eta_A \circ m, f\rangle} : \mathcal{L}A \to \mathcal{L}B \end{cases} \tag{32}$$

where the morphism-mapping $\overline{[\eta_A \circ m, f\rangle}$ is explained in further detail by the diagram in (33):

$$\begin{tikzcd}
X \arrow[r, tail, "m"] \arrow[d, equal, "id_X"'] & A \arrow[r, equal, "id_A"] \arrow[d, equal, "id_A"'] & A \arrow[d, tail, "\eta_A"] \\
X \arrow[r, tail, "m"] \arrow[d, "f"'] & A \arrow[r, tail, "\eta_A"] \arrow[d, "\overline{[m,f\rangle}"'] & \mathcal{L}A \arrow[d, "\overline{[\eta_A \circ m,f\rangle}"] \\
B \arrow[r, tail, "\eta_B"'] & \mathcal{L}B \arrow[r, equal, "id_{\mathcal{L}B}"'] & \mathcal{L}B
\end{tikzcd} \tag{33}$$

The underlying co-free construction of the adjunction $\Gamma_{\mathcal{M}} \dashv \mathcal{L}$ is shown in (34).

$$\begin{tikzcd}
Par_{\mathcal{M}}(\mathbb{C}) & & & \mathbb{C} & \\
B & \Gamma_{\mathcal{M}}A \arrow[l, harpoon', "\forall[m,f\rangle"'] \arrow[dl, dashed, "\Gamma_{\mathcal{M}}(\overline{[m,f\rangle})"] & & & A \arrow[dl, dashed, "\exists!\overline{[m,f\rangle}"'] \\
\Gamma_{\mathcal{M}}\mathcal{L}B \arrow[u, harpoon, "\varepsilon_B"] & & & \mathcal{L}B &
\end{tikzcd} \tag{34}$$

with $\varepsilon_B := [\eta_B, id_B\rangle$ for all $B \in |Par_{\mathcal{M}}(\mathbb{C})|$ and $A \in |\mathbb{C}|$.

Finally, there is an important lesser known fact about partial arrow classifiers:

Lemma 4 When the morphism f in (31) is a monomorphism, so is $\overline{[m, f\rangle}$.

Proof. Consider again the diagram in (33) and recall the adjunction $\Gamma_{\mathcal{M}} \dashv \mathcal{L}$. Thus $\mathcal{L}$ has left adjoint in $\Gamma_{\mathcal{M}}$ and therefore preserves limits, including monomorphisms (which are just special pullbacks). Hence, $\overline{[\eta_A \circ m, f\rangle}$ is a monomorphism since it $\mathcal{L}f$. Monomorphisms compose and so is $\overline{[\eta_A \circ m, f\rangle} \circ \eta_A$ a monomorphism. The latter is equal to $\overline{[m, f\rangle}$ because of the universal property of the partial arrow classifiers: The upper squares are trivially pullbacks and the lower rectangle is a pullback by the universal property. The whole outline $[id_A \circ m, f \circ id_X\rangle$ is a partial map and equal to $[m, f\rangle$. Since there is a unique morphism that make the outer square a pullback, we have

$$\overline{[m, f\rangle} = \overline{[\eta_A \circ m, f\rangle} \circ \eta_A$$

which is a monomorphism. $\square$

B. Proofs

B.1. Proof of Theorem 1

The schema for this proof is sketched in (35). Proposition 2 showed that $\mathbb{M}$ is a full subcategory of the diagram category $\mathbb{G}^{\mathbb{I}} = (\mathbb{SET}^{\mathbb{B}})^{\mathbb{I}}$. Using *cartesian closedness* of the category of small categories [AHS90, 27.3 (e)], we get that $\mathbb{SET}^{\mathbb{B}\times\mathbb{I}} \cong (\mathbb{SET}^{\mathbb{B}})^{\mathbb{I}}$. Intuitively, this means that a functor with two arguments (of type $\mathbb{B}$ and $\mathbb{I}$, resp.) can be *curried*, i.e. it can be interpreted as a family of functors, each of which has one argument of type $\mathbb{B}$, and the family varying over a parameter of type $\mathbb{I}$. Finally, we define an auxiliary category $\mathbb{N}$ (B.1.1) as a subcategory of $\mathbb{SET}^{\mathbb{B}\times\mathbb{I}}$ of those functors $N : \mathbb{B} \times \mathbb{I} \to \mathbb{SET}$ that map "the same" morphism (modulo the construction in [AHS90]) to monomorphisms as in $\mathbb{M}$ (B.1.2). Finally, we show that $\mathbb{N}$ is isomorphic to $\mathbb{CS}$.

$$\begin{tikzcd}
 & \mathbb{SET}^{\mathbb{B}\times\mathbb{I}} \arrow[rr, leftrightarrow, "\cong \text{[AHS90, 27.3 (e)]}"] & & (\mathbb{SET}^{\mathbb{B}})^{\mathbb{I}} \\
\mathbb{CS} \arrow[r, leftrightarrow, "\cong \text{(Sect. B.1.3)}"'] & \mathbb{N} \arrow[u, hook, "\text{(Sect. B.1.1)}"] \arrow[rr, leftrightarrow, "\cong \text{(Sect. B.1.2)}"'] & & \mathbb{M} \arrow[u, hook, "\text{(Prop. 2)}"']
\end{tikzcd} \tag{35}$$

B.1.1. Definition of $\mathbb{N}$

Because $\mathbb{I}$ contains $2n + 1$ objects, the cartesian product $\mathbb{B} \times \mathbb{I}$ of $\mathbb{B}$ and $\mathbb{I}$ in the category of small categories has $2n + 1$ copies $\mathbb{B}_{-n}, \ldots, \mathbb{B}_0, \ldots, \mathbb{B}_n$ of $\mathbb{B}$ together with arrow spans

$$(s, 0) \xleftarrow{(id_s, j0)} (s, -j) \xrightarrow{(id_s, jj)} (s, j)$$

for each $j \in \{1, \ldots, n\}$ and for each $s \in |\mathbb{B}|$.

As an example, we have drawn the category $\mathbb{B} \times \mathbb{I}$ for a star-shape $\mathbb{I}$ with degree $n = 2$ and $\mathbb{B} := E \overset{s}{\to} V \overset{t}{\leftarrow} E$ (the signature for directed multi-graphs, where identities are omitted) in (36).

$$\begin{tikzcd}
(V,1) \arrow[loop above, "{(id_N,id_1)}"] & (N,-1) \arrow[loop above, "{(id_N,id_{-1})}"] \arrow[l, "{(id_N,11)}"'] \arrow[r, "{(id_N,10)}"] & (V,0) \arrow[loop above, "{(id_N,id_0)}"] & (N,-2) \arrow[loop above, "{(id_N,id_{-2})}"] \arrow[l, "{(id_N,20)}"'] \arrow[r, "{(id_N,22)}"] & (V,2) \arrow[loop above, "{(id_N,id_2)}"] \\
(E,1) \arrow[loop below, "{(id_E,id_1)}"] \arrow[u, shift left, "{(s,id_1)}"'] \arrow[u, shift right, "{(t,id_1)}"'] & (E,-1) \arrow[loop below, "{(id_E,id_{-1})}"] \arrow[l, "{(id_E,11)}"] \arrow[r, "{(id_E,10)}"'] \arrow[u, shift right, "{(s,id_{-1})}"] \arrow[u, shift left, "{(t,id_{-1})}"'] \arrow[ul, shift right, "{(s,11)}"] \arrow[ul, shift left, "{(t,11)}"'] \arrow[ur, shift right, "{(t,10)}"'] \arrow[ur, shift left, "{(s,10)}"] & (E,0) \arrow[loop below, "{(id_E,id_0)}"] \arrow[u, shift right, "{(s,id_0)}"] \arrow[u, shift left, "{(t,id_0)}"'] & (E,-2) \arrow[loop below, "{(id_E,id_{-2})}"] \arrow[l, "{(id_E,20)}"] \arrow[r, "{(id_E,22)}"'] \arrow[u, shift right, "{(s,id_{-2})}"] \arrow[u, shift left, "{(t,id_{-2})}"'] \arrow[ul, shift right, "{(s,20)}"] \arrow[ul, shift left, "{(t,20)}"'] \arrow[ur, shift right, "{(t,22)}"'] \arrow[ur, shift left, "{(s,22)}"] & (E,2) \arrow[loop below, "{(id_E,id_2)}"] \arrow[u, shift right, "{(s,id_2)}"] \arrow[u, shift left, "{(t,id_2)}"']
\end{tikzcd} \tag{36}$$

Hence, objects in $\mathbb{SET}^{\mathbb{B}\times\mathbb{I}}$ are functors $N : \mathbb{B} \times \mathbb{I} \to \mathbb{SET}$, which simultaneously act as $2n + 1$ functors from $\mathbb{B}$ to $\mathbb{SET}$, augmented with spans

$$N((s,0)) \xleftarrow{N((id_s,j0))} N((s,-j)) \xrightarrow{N((id_s,jj))} N((s,j))$$

of total functions for each $s \in |\mathbb{B}|$ and all $j \in \{1, \dots, n\}$.

We define $\mathbb{N}$ to be the subcategory of $\mathbb{SET}^{\mathbb{B}\times\mathbb{I}}$, which maps all $N((id_s, j0))$ to to monomorphisms (injective functions) in $\mathbb{SET}$ for all $s \in |\mathbb{B}|$.

B.1.2. *Equivalence of* $\mathbb{N}$ *and* $\mathbb{M}$

The equivalence between $\mathbb{SET}^{\mathbb{B}\times\mathbb{I}}$ and $(\mathbb{SET}^{\mathbb{B}})^{\mathbb{I}}$ is based on currying and un-currying the respective functor definitions. The category $\mathbb{M}$ has imposed the restriction, that all images $M(j0) : M(-j) \to M(0)$ of $j0$ under an $\mathbb{M}$-object M are monomorphisms in $\mathbb{G}$. The latter is represented as a family $(M(j0)(op) : M(-j)(s) \to M(-j)(s'))_{op:s\to s'\in Arr_{\mathbb{B}}}$, which has one-to-one correspondence with the family $N((op, j0))$ $(op : s \to s' \in Arr_{\mathbb{B}}$, $j \in \{1, \dots, n\})$.

B.1.3. *Equivalence of* $\mathbb{CS}$ *and* $\mathbb{N}$

Let $N \in |\mathbb{N}|$ and $C \in |\mathbb{CS}|$. We will show that every comprehensive system C has an equivalent representation as an $\mathbb{N}$-object. First, we define a one-to-one correspondence within C's components. We identify

- $N((s, i))$ and $C_i(s)$ for all $s \in |\mathbb{B}|$, $0 \leq i \leq n$ (1. in Definition 2).
- and $N((op, id_i)) : N((s, i)) \to N((s', i))$ and $C_i(op) : C_i(s) \to C_i(s')$ for all (2. in Definition 2).

Furthermore, in Appendix A.3 it was demonstrated, how a partial morphism in some category can be expressed by an equivalence class of spans with the inner leg being a monomorphism. Therefore,

- $N((s, k))$ for all $s \in |\mathbb{B}|$, $-n \leq k < 0$ (the apex of the span),
- $N((id_s, j0)) : N(s, -j) \rightarrowtail N(s, 0)$ for all $s \in |\mathbb{B}|$, $0 < j \leq n$ (the domain embedding),
- and $N((id_s, jj)) : N(s, -j) \to N(s, j)$ for all $s \in |\mathbb{B}|$, $0 < j \leq n$ (the concrete assignment)

form a concrete representative of the projection $p^C_{j,s} : C_0(s) \rightharpoonup C_j(s)$ in C.

The remaining constituents of N:

- $N((op, j0))$ for all $0 < j \leq n$ and non-identity morphisms $op : s \to s' \in Arr_{\mathbb{B}}$,
- $N((op, jj))$ for all $0 < j \leq n$ and non-identity morphisms $op : s \to s' \in Arr_{\mathbb{B}}$

are subject to the following equations, which are a consequence of the definition of composition in the product category $\mathbb{B} \times \mathbb{I}$ (compare with the diagonals in (36)) and of N being a functor (which must preserve these compositions):

$$\begin{aligned} N((id_{s'}, jj)) \circ N((op, id_{-j})) &= N((id_{s'}, jj) \circ (op, id_{-j})) \\ &= N((op, jj)) \end{aligned}$$

$$
\begin{aligned}
&= N((op, id_j) \circ (id_s, jj)) \\
&= N((op, id_j)) \circ N((id_s, jj)) \\
N((id_{s'}, j0)) \circ N((op, id_{-j})) &= N((id_{s'}, j0) \circ (op, id_{-j})) \\
&= N((op, j0)) \\
&= N((op, id_j) \circ (id_s, j0)) \\
&= N((op, id_j)) \circ N((id_s, j0))
\end{aligned}
$$

These conditions correspond to the generalised edge-node incidence (6)+(7). Thus, $N((op, j0))$ and $N((op, jj))$ can be seen as reifications (witnesses) of this condition.

Hence $\mathbb{CS} \cong \mathbb{M}$, as claimed.

B.2. Proof of Theorem 2

Let $(m : B \to D, f : C \to D)$ be a co-span in $\mathbb{CS}$. Utilizing Theorem 1, we chose to perform the proof in $\mathbb{M}$, i.e showing the existence of $(g : A \to B, n : A \to C)$ as a pullback for (m, f) in $\mathbb{M}$.

Recall that $\mathbb{G} = \mathbb{SET}^{\mathbb{B}}$ has all pullbacks, which are constructed component-wise for each $s \in |\mathbb{B}|$ (Lemma 2). The component-wise construction lifts to $\mathbb{G}^{\mathbb{I}}$ (for each $i \in \mathbb{I}$) resulting in the j-indexed family of cubes in shown in (37). Note that for any object $M \in |\mathbb{M}|$, the span $(M(0) \hookleftarrow M(-j) \to, M(jj))$ in (13) can be seen as a partial $\mathbb{G}$-morphism $m_j : M^0 \rightharpoonup M^j$. Thus, and because the choice of the monomorphism domain object $M(-j)$ in Def. 4 is free up to isomorphism, by convention we can assume that $M(j0)$ is an inclusion: $M(j0) =: \subseteq_j^M : dom(m_j) \hookrightarrow M^0$, which explains the notation in (37).

$$\begin{tikzcd}
 & & A^0 \arrow[rr, "n_0"] \arrow[dl, "g_0"'] & & C^0 \arrow[dl, "f_0"] \\
0 & B^0 \arrow[rr, "m_0"] & & D^0 & \\
 & & dom(a_j) \arrow[uu, dashed, "A(j0)"] \arrow[dd, dashed, "a_j"] \arrow[rr, "dn_j"] \arrow[dl, "dg_j"'] & & dom(c_j) \arrow[uu, hook, "C(j0)"'] \arrow[dd, "c_j"] \arrow[dl, "df_j"] \\
-j \arrow[uu, "j0"] \arrow[dd, "jj"'] & dom(b_j) \arrow[uu, hook, "B(j0)"] \arrow[dd, "b_j"] \arrow[rr, "dm_j"] & & dom(d_j) \arrow[uu, hook, "D(j0)"] \arrow[dd, "d_j"] & \\
 & & A^j \arrow[rr, "n_j"] \arrow[dl, "g_j"'] & & C^j \arrow[dl, "f_j"] \\
j & B^j \arrow[rr, "m_j"'] & & D^j &
\end{tikzcd} \tag{37}$$

The spans (g_0, n_0), (dg_j, dn_j), and (g_j, n_j) are constructed component-wise as pullbacks in $\mathbb{M}$. The universal pullback property of the top face w.r.t to the horizontal inner face in the middle provides the morphism $A(j0)$ that makes the upper rear faces commute. And the universal pullback property of the bottom face w.r.t to the horizontal inner face in the middle provides the morphism a_j that makes the lower rear faces commute. It remains to show that $A \in |\mathbb{M}|$. For this we have to show that $A(j0)$ is a monomorphism

Assume two morphisms $x : X \to dom(a_j)$ and $y : X \to dom(a_j)$ such that $A(j0) \circ x = A(j0) \circ y$. Postcomposing this arrow simultaneosly with n_0 and n_0 yields

$$
\begin{aligned}
n_0 \circ A(j0) \circ x &= n_0 \circ A(j0) \circ y \\
g_0 \circ A(j0) \circ x &= g_0 \circ A(j0) \circ y
\end{aligned}
$$

using the commutativity of the left and back face, followed by monomorphism property of $B(j0)$ and $C(j0)$ we get

$$\begin{aligned} dn_j \circ x &= dn_j \circ y \\ dg_j \circ x &= dg_j \circ y \end{aligned}$$

Recall that the horizontal inner face is a pullback, i.e. dn_j and dg_j are jointly monic and therefore $x = y$ as required.

B.3. Proof of Proposition 3

Isomorphisms trivially yield naturality squares that are pullbacks. The composition of two reflective monomorphisms is a reflective monomorphism as well because pullbacks compose. To see that reflective monomorphisms are stable under pullback, consider again our pullback construction in $\mathbb{M}$ from the proof of Theorem 2, depicted in (37). This time, the upper front face is a pullback and all components of m are monic, i.e. m is a reflective mono. We have to show that all components of n are monic and that the upper back face is a pullback, i.e. n is a reflective mono. The former, however, is easy, since pullbacks in $\mathbb{M}$ preserve monomorphisms (recall that A was constructed via taking pullbacks component-wise in Theorem 2). For the pullback property consider the diagram in (38)

$$\begin{tikzcd}
dom(a_j) \arrow[r, hook, "A(j0)"] \arrow[d, tail, "dn_j"'] \arrow[rr, bend left, "B(j0)\circ dg_j"] \arrow[dr, phantom, "(a)"] & A^0 \arrow[r, "g_0"] \arrow[d, tail, "n_0"] \arrow[dr, phantom, "(b)"] & B^0 \arrow[d, tail, "m_0"] \\
dom(c_j) \arrow[r, hook, "C(j0)"] \arrow[rr, bend right, "D(j0)\circ df_j"'] & C^0 \arrow[r, "f_0"] & D^0
\end{tikzcd} \tag{38}$$

We use the fact that the upper front-face in (37) is a pullback because m is reflective monomorphism by assumption. We compose it with the horizontal inner face in (37), which is a pullback by construction resulting in a pullback that forms the outer rectangle in (38). The right square (b) in (38) is the top face in (37) and therefore also a pullback by construction. Now we know that the upper and lower triangles in (38) commute because they represent the upper left and upper right faces in (37). Therefore, we can use the pullback-decomposition lemma to conclude that (a) is a pullback, as desired.

B.4. Proof of Theorem 3

Let $m : A \rightarrowtail B$ and $f : A \to C$ be a span of comprehensive systems with $m \in \mathcal{M}$ a reflective monomorphism. Again, we construct the pushout $(g : B \to D, n : C \rightarrowtail D)$ component-wise in $\mathbb{M}$ like we did in the proof of Theorem 2. The construction is dual to the one in the proof of Theorem 2 and again we have to pay special attention to the upper cubes (images of $j0$)

$$\begin{tikzcd}
& A^0 \arrow[rrr, tail, "m_0"] \arrow[dl, "f_0"'] & & & B^0 \arrow[dl, "g_0"] \\
C^0 \arrow[rrr, tail, "n_0"'] & & & D^0 & \\
& dom(a_j) \arrow[uu, hook, "A(j0)"] \arrow[rrr, tail, "dm_j"] \arrow[dl, "df_j"'] & & & dom(b_j) \arrow[uu, hook', "B(j0)"'] \arrow[dl, "dg_j"] \\
dom(c_j) \arrow[uu, hook, "C(j0)"] \arrow[rrr, tail, "dn_j"'] & & & dom(d_j) \arrow[uu, dashed, "D(j0)"'] &
\end{tikzcd} \tag{39}$$

Consider the commutative cube in (39). The rear faces are given via the span (m, f) where the back face is a pullback because $m \in \mathcal{M}$. The top and bottom faces are constructed as pushouts and the by universal property of the bottom face pushout, we get the morphism $D(j0)$ that makes the front and right face commute. Since $\mathbb{G}$ is adhesive [LS04], pushouts preserve monomorphisms such that n_0 and dn_j are monomorphisms.

Next, we show that the front face is a pullback, for this consider the diagram in (40).

$$\begin{tikzcd}
dom(a_j) \arrow[r, "df_j"] \arrow[d, tail, "dm_j"'] \arrow[rr, bend left, "f_0 \circ A(j0)"] \arrow[dr, phantom, "(a)"] & dom(c_j) \arrow[r, hook, "C(j0)"] \arrow[d, tail, "dn_j"'] \arrow[dr, phantom, "(b)"] & C^0 \arrow[d, tail, "n_0"] \\
dom(b_j) \arrow[r, "dg_j"'] \arrow[rr, bend right, "g_0 \circ B(j0)"'] & dom(d_j) \arrow[r, "D(j0)"'] & D^0,
\end{tikzcd} \tag{40}$$

The square (a) is a pushout by construction (bottom face in 39) and the outer square is a pullback composed out of the back (pullback due the reflective property of m) and top face (pushouts along monomorphisms in adhesive categories are pullbacks as well) in (39). Using the special pullback-pushout property [LS06, Lemma 6] the square (b) becomes a pullback.

It remains to show that $D(j0)$ is a monomorphism. For this consider the following $\mathbb{SET}$-theoretic argument, which is stable under sort-wise construction (lifts to $\mathbb{SET}^{\mathbb{B}}$): Assume $D(j0)$ is not monic, then there are two elements $x, y \in dom(d_j)$ that $D(j0)$ maps to the same element $z \in D^0$. Now, we know that $dom(d_j)$ is the apex of a pushout, therefore dn_j and dg_j are jointly surjective and thus x, y must have pre-images x', y' under (dn_j, dg_j) in $dom(c_j)$ or $dom(b_j)$.

Note that the cases $x', y' \in dom(c_j)$ or $x' \in dom(c_j) \wedge y' \in dom(b_j)$ disqualify immediately since (a) is a pullback. Therefore consider the case $x, y \in dom(b_j)$ further. Then x', y' must have distinct images under inclusion $B(j0)$ in B^0 that must be mapped to z via g_0. Now D^0 is also constructed as the apex of a pushout and for $x', y' \in B^0$ being mapped to the same element in D^0, there must be pre-images of x', y' in A^0 that are mapped to the same element in $z \in C^0$. But $dom(c_j)$ is the pullback object of n_0 and $D(j0)$ and therefore $z \in C^0$ must have two pre-images ins $dom(c_j)$ which violates the monomorphism property of $C(j0)$ $\lightning$.

Hence, we must conclude that $D(j0)$ is a monomorphism.

B.5. Proof of Theorem 4

Let B be a comprehensive system. For the existence of $\mathcal{M}$-partial arrow classifiers, we have to show the existence of an $\mathcal{M}$-morphism $\eta_B : B \rightarrowtail \mathcal{L}B$ such that for every span $(f : X \to B, m : X \rightarrowtail A)$ there exists a unique morphism $\overline{[m, f\rangle} : A \to \mathcal{L}B$ such that the resulting square is a pullback. Again we perform the construction in

$\mathbb{M}$ and focus on the image of $j0$'s. The case for jj's works analogously.

$$\begin{tikzcd}
& X^0 \arrow[rr, tail, "m_0"] \arrow[dl, "f_0"'] & & A^0 \arrow[dl, dashed, "\overline{[m_0, f_0\rangle}"] \\
B^0 \arrow[rr, tail, "\eta_{B^0}"'] & & \mathcal{L}B^0 & \\
& dom(x_j) \arrow[rr, tail, "dm_j"] \arrow[dl, "df_j"'] \arrow[uu, hook, "X(j0)"] & & dom(a_j) \arrow[dl, dashed, "\overline{[dm_j, df_j\rangle}"] \arrow[uu, hook, "A(j0)"'] \\
dom(b_j) \arrow[rr, tail, "\eta_{B(j0)}"'] \arrow[uu, hook, "B(j0)"] & & \mathcal{L}dom(b_j) \arrow[uu, dotted, "\overline{[d\eta_{B_j}, B(j0)\rangle}"'] &
\end{tikzcd} \tag{41}$$

Consider the cube in (41). We use the fact that $\mathbb{G}$ has mono-partial arrow classifiers and construct the partial arrow classifier of B in $\mathbb{M}$ component-wise. This gives the existence of unique (dashed arrows) $\overline{[m_0, f_0\rangle}$ and $\overline{[dm_j, df_j\rangle}$ such that the top and bottom face become pullbacks. Using the partial arrow classifier property of $\eta_{B0} : B^0 \rightarrowtail \mathcal{L}B^0$ w.r.t. $[\eta_{B(j0)}, B(j0)\rangle$ yields $\overline{[\eta_{B(j0)}, B(j0)\rangle}$ such that the front face becomes a pullback, i.e. η_B is a reflective monomorphism if it is a monomorphism. Moreover, we have to show that the right face commutes. The former is due to the fact that if in a partial arrow span $[m, f\rangle$, f is a monomorphism, then also $\overline{[m, f\rangle}$ is a monomorphism, see Lemma 4. To show that the right face commutes, i.e. $\overline{[m_0, f_0\rangle} \circ A(j0) = \overline{[d\eta_{B_j}, B(j0)\rangle} \circ \overline{[dm_j, df_j\rangle}$ consider the diagram (42).

$$\begin{tikzcd}[column sep=huge, row sep=huge]
dom(x_h) \arrow[r, tail, "dm_j"] \arrow[d, "f_0 \circ X(j0)"'] & dom(a_j) \arrow[d, bend right=60, "\overline{[d\eta_{B_j}, B(j0)\rangle} \circ \overline{[dm_j, df_j\rangle}"'] \arrow[d, dashed, "\overline{[dm_j, f_0 \circ X(j0)\rangle}"] \arrow[d, bend left=60, "\overline{[m_0, f_0\rangle} \circ A(j0)"] \\
B^0 \arrow[r, tail, "\eta_{B^0}"'] & \mathcal{L}B^0
\end{tikzcd} \tag{42}$$

Using the partial map $[dm_j, f_0 \circ X(j0)\rangle$ w.r.t. η_{B^0}, there is a unique (dashed) arrow making the square a pullback. Thus

$$\overline{[m_0, f_0\rangle} \circ A(j0) = \overline{[dm_j, f_0 \circ X(j0)\rangle} = \overline{[d\eta_{B_j}, B(j0)\rangle} \circ \overline{[dm_j, df_j\rangle}$$

B.6. Proof of Theorem 6

An immediate consequence of the definitions of M in terms of coproducts is that $\mathbf{T}$ is injective on objects and on morphism sets, hence an embedding, such that it remains to show preservation of pushouts.

As seen in Theorem 3, pointwise pushout construction of a span in $\mathbb{M}$ may fail to belong to $\mathbb{M}$. This obstacle can be overcome because we use coproducts in the construction of $\mathbf{T}$. Let $\nu : M \to N := \mathbf{T}(n : D \to H)$, then the naturality squares $\nu_0 \circ M(j0) = N(j0) \circ d\nu_j$ (images of $j0$'s) are pullbacks due to the following:

The definition of $M(j0)$ can also be written

$$M(j0) : \coprod_{r \in R} D_r^j \hookrightarrow \coprod_{r \in R} D(r)$$

with $D_r^j = D(r)$, if there is $f \in Arr_{\mathbb{X}}(\_, j)$ and $r = \text{dom}(f)$, and $D_r^j = 0$ (the initial object, see Appendix A.2.1, i.e. the empty graph) otherwise, because $X + 0 \cong X$ in $\mathbb{G}$. Similarly, this inclusion can be extended for M^1.

In both cases the summand-wise squares

$$\begin{tikzcd} D(r) \arrow[r, "n_r"] & H(r) \\ D_r^j \arrow[u, "id \text{ or } 0_{D^0(r)}"] \arrow[r, "n_r \text{ or } id_0"] & H_r^j \arrow[u, "id \text{ or } 0_{D^1(r)}"'] \end{tikzcd} \tag{43}$$

are pullbacks, such that it suffices to show that two pullback squares in $\mathbb{G}$ always add up to a pullback square of their coproducts, see (44).

$$\begin{tikzcd} A_1 \arrow[r, "h_1"] & B_1 & A_2 \arrow[r, "h_2"] & B_2 & & A_1+A_2 \arrow[r, "h_1+h_2"] & B_1+B_2 \\ C_1 \arrow[u, "k_1'"] \arrow[r, "h_1'"'] \arrow[ru, phantom, "(p.b.)"] & D_1 \arrow[u, "k_1"'] & C_2 \arrow[u, "k_2'"] \arrow[r, "h_2'"'] \arrow[ru, phantom, "(p.b.)"] & D_2 \arrow[u, "k_2"'] & \Rightarrow & C_1+C_2 \arrow[u, "k_1'+k_2'"] \arrow[r, "h_1'+h_2'"'] \arrow[ru, phantom, "(p.b.)"] & D_1+D_2 \arrow[u, "k_1+k_2"'] \end{tikzcd} \tag{44}$$

This can be demonstrated as follows: $\mathbb{G}$ is known to be *extensive*, i.e. the functor $+ : \mathbb{G} \downarrow B_1 \times \mathbb{G} \downarrow B_2 \to \mathbb{G} \downarrow (B_1 + B_2)$ between comma categories is an equivalence of categories, its inverse is taking pullbacks along coproduct injections [CLW93]. This adds pullbacks adjacent on the right of the two left pullbacks in (44) and, by pullback composition [BW90], we obtain two pullbacks with the arrow $k_1 + k_2$ as right vertical arrow. Since $\mathbb{G}$ is a topos [Gol06], it can be shown that these two then add to the right pullback in (44), see \$ 5.3. in [Gol06].

Now, consider the cube from (39) in the proof of theorem 3. This time left and back faces are pullbacks. Using the fact that pushouts in $\mathbb{G}$ are mono-hereditary, cf. definition 23 in Appendix A.3, we conclude that front and right faces are pullbacks and that $D(j0)$ is a monomorphism, i.e. the result is actually a comprehensive system. Hence, we have to show that all components are pushouts, i.e. the right squares in (45) are pushouts in $\mathbb{G}$ for all $i \in Arr_{\mathbb{I}}$.

$$\begin{tikzcd} D \arrow[r, Rightarrow, "m"] \arrow[d, Rightarrow, "f"'] & G^1 \arrow[d, Rightarrow, "f'"] & M \arrow[r, Rightarrow, "\mu"] \arrow[d, Rightarrow, "\phi"'] & N \arrow[d, Rightarrow, "\phi'"] & M(i) \arrow[r, "\mu_i"] \arrow[d, "\phi_i"'] & N(i) \arrow[d, "\phi_i'"] \\ H \arrow[r, Rightarrow, "m'"] & J & K \arrow[r, Rightarrow, "\mu'"] & L & K(i) \arrow[r, "\mu_i'"] & L(i) \end{tikzcd} \tag{45}$$

This is, however, clear from the definition of $\mathbf{T}$ for $i > 0$ (because models are untouched and the left square is a pushout by assumption). For $i \leq 0$, all four objects in the right square are coproducts over a certain indexing set I ($I = Arr_{\mathbb{X}}$ for $i = 0$ and $I = Arr_{\mathbb{X}}(\_, j)$ for $i = -j < 0$), where the coproduct amalgamates relation graphs of the graph diagrams (index $r \in R$).

Finally, since $\coprod$ is a functor from $\mathbb{G}^I$ to $\mathbb{G}$, which is left-adjoint to the diagonal functor Δ_I (cf. [BW90, Ex.13.2.4]), it preserves colimits, hence all squares are pushouts, because in the left square there are pointwise pushouts separately for each relation index $r \in R$.

Funding Open access funding provided by Western Norway University Of Applied Sciences.

References

[ABW$^+$19] Anjorin A, Buchmann T, Westfechtel B, Diskin Z, Ko H-S, Eramo R, Hinkel G, Samimi-Dehkordi L, Zündorf A (2019) Benchmarking bidirectional transformations: theory, implementation, application, and assessment. In: Software and systems modeling

[AHS90] Adámek J, Herrlich H, Strecker GE (1990) Abstract and concrete categories: the joy of cats. Pure and applied mathematics. Wiley

[AK02] Atkinson C, Kühne T (2002) Rearchitecting the UML infrastructure. ACM Trans Model Comput Simul 12(4):290–321

[ARNRSG06] Aizenbud-Reshef N, Nolan BT, Rubin J, Shaham-Gafni Y (2006) Model traceability. IBM Syst J 45(3):515–526

[ASB10] Atkinson C, Stoll D, Bostan P (2010) Orthographic software modeling: a practical approach to view-based development. In: Maciaszek LA, González-Pérez C, Jablonski S (eds) ENASE 2009, communications in computer and information science. Springer, Berlin, pp 206–219

[ASCG$^+$18] Abou-Saleh F, Cheney J, Gibbons J, McKinna J, Stevens P (2018) Introduction to bidirectional transformations. In: Gibbons J, Stevens P (eds) Bidirectional transformations: international summer school, 2016, LNCS. Springer, pp 1–28

[BBCW19] Bruneliere H, Burger E, Cabot J, Wimmer M (2019) A feature-based survey of model view approaches. Softw Syst Model 18(3):1931–1952

[BBDF$^+$06] Bézivin J, Bouzitouna S, Del Fabro MD, Gervais M-P, Jouault F, Kolovos D, Kurtev I, Paige RF (2006) A canonical scheme for model composition. In: Rensink A, Warmer J (eds) Model driven architecture—foundations and applications, lecture notes in computer science. Springer, Berlin, pp 346–360

[BCE$^+$06] Brunet G, Chechik M, Easterbrook S, Nejati S, Niu N, Sabetzadeh M (2006) A manifesto for model merging. In: GaMMa '06. ACM, New York, NY, USA, pp 5–12

[BEEH$^+$19] Bennani S, Ebersold S, El Hamlaoui M, Coulette B, Nassar M (2019) A collaborative decision approach for alignment of heterogeneous models. In: 2019 IEEE 28th international conference on enabling technologies: Infrastructure for collaborative enterprises (WETICE), pp 112–117. ISSN: 2641-8169

[Ber03] Bernstein PA (2003) Applying model management to classical meta data problems. In: CIDR

[BJV04] Bézivin J, Jouault F, Valduriez P (2004) On the need for megamodels. In: Proceedings of the OOPSLA/GPCE: best practices for model-driven software development workshop, 19th Annual ACM conference on object-oriented programming, systems, languages, and applications (2004), Vancouver, Canada

[BKMW09] Boronat A, Knapp A, Meseguer J, Wirsing M (2009) What is a multi-modeling language? In: WADT 2008. Springer, Berlin pp 71–87

[BMdlC$^+$20] Barriga A, Mandow L, de la Cruz José LP, Rutle A, Heldal R, Iovino L (2020) A comparative study of reinforcement learning techniques to repair models. In: Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: companion proceedings, MODELS'20. Association for Computing Machinery, New York, NY, USA, pp 1–9

[BW90] Barr M, Wells C (1990) Category theory for computing science. Prentice Hall

[Bé05] Bézivin J (2005) On the unification power of models. Softw Syst Model 4(2):171–188

[CCP19] Cicchetti A, Ciccozzi F, Pierantonio A (2019) Multi-view approaches for software and system modelling: a systematic literature review. Softw Syst Model 18(6):3207–3233

[CFH$^+$09] Czarnecki K, Foster N, Hu Z, Lämmel R, Schürr A, Terwilliger JF (2009) Bidirectional transformations: a cross-discipline perspective. In: ICMT 2009, pp 193–204

[CGMS15] Cheney J, Gibbons J, McKinna J, Stevens P (2015) Towards a principle of least surprise for bidirectional transformations. In: Proceedings of the 4th international workshop on bidirectional transformations co-located with software technologies: applications and fFoundations (STAF 2015), vol 1396, pp 66–80

[CKSZ19] Cleve A, Kindler E, Stevens P, Zaytsev V(2019) Multidirectional transformations and synchronisations (Dagstuhl seminar 18491). Dagstuhl Rep 8(12):1–48

[CLW93] Carboni A, Lack S, Walters RFC (1993) Introduction to extensive and distributive categories. J Pure Appl Algebra 84(2):145–158

[Cou97] Courcelle B (1997) The expression of graph properties and graph transformations in monadic second-order logic. In: Rozenberg G (ed) Handbook of graph grammars and computing by graph transformation. World Scientific, River Edge, pp 313–400

[CR96] Colmerauer A, Roussel P (1996) The birth of Prolog. In: History of programming languages—II. Association for Computing Machinery, New York, pp 331–367

[Dis97] Diskin Z (1997) Towards algebraic graph-based model theory for computer science. Bull Symb Logic 3:144–145

[DKL19] Diskin Z, König H, Lawford M (2019) Multiple model synchronization with multiary delta lenses with amendment andK-Putput. Form Aspects Comput 31(5):611–640

[DKPF09] Drivalos N, Kolovos DS, Paige RF, Fernandes KJ (2009) Engineering a DSL for software traceability. In: Gašević D, Lämmel R, Van Wyk E (eds) Software language engineering, lecture notes in computer science. Springer, Berlin, pp 151–167

[dLG10] de Lara J, Guerra E (2010) Deep meta-modelling with MetaDepth. In: Vitek J (ed), Objects, models, components, patterns, lecture notes in computer science. Springer, Berlin, pp 1–20

[dLGKH18] de Lara J, Guerra E, Kienzle J, Hattab Y (2018) Facet-oriented modelling: open objects for model-driven engineering. In: SLE 2018. Association for Computing Machinery, Boston, MA, USA, pp 147–159

[DW07] Diskin Z, Wolter U (2007) A diagrammatic logic for object-oriented visual modeling. In: ACCAT '07, pp 19–41

[DXC11] Diskin Z, Xiong Y, Czarnecki K (2011) Specifying Overlaps of heterogeneous models for global consistency checking. In: MDI@MODELS 2010, pp 165–179

[EEE$^+$07] Ehrig H, Ehrig K, Ermel C, Hermann F, Taentzer G (2007) Information preserving bidirectional model transformations. In: Dwyer MB, Lopes A (eds) Fundamental approaches to software engineering, lecture notes in computer science. Springer, Berlin, pp 72–86

[EEH08] Ehrig H, Ehrig K, Hermann F (2008) From model transformation to model integration based on the algebraic approach to triple graph grammars. Electron Commun EASST 10:65

[EEPT06] Ehrig H, Ehrig K, Prange U, Taentzer G (2006) Fundamentals of algebraic graph transformation. Springer

[Egy07] Egyed A (2007) Fixing inconsistencies in UML design models. In: Proceedings—international conference on software engineering, pp 292–301

[EHHS00] Engels G, Hausmann JH, Heckel R, Sauer S (2000) Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Andy E, Stuart K, Bran S (eds) UML 2000—the unified modeling language, lecture notes in computer science. Springer, Berlin, pp 323–337

[EMM$^+$12] Eramo R, Malavolta I, Muccini H, Pelliccione P, Pierantonio A (2012) A model-driven approach to automate the propagation of changes among architecture description languages. Softw Syst Model 11(1):29–53

[EP06] Ehrig H, Prange U (2006) Weak adhesive high-level replacement categories and systems: a unifying framework for graph and petri net transformations. In: Futatsugi K, Jouannaud J-P, Meseguer J (eds) Algebra, meaning, and computation: essays dedicated to Joseph A. Goguen on the Occasion of his 65th birthday, lecture notes in computer science. Springer, Berlin, pp 235–251

[EPS73] Ehrig H, Pfender M, Schneider HJ (O1973) Graph-grammars: an algebraic approach. In: 14th Annual symposium on switching and automata theory (swat 1973), pp 167–180

[ES13] Euzenat J, Shvaiko P (2013) Ontology matching, 2 edn. Springer, Berlin

[FGH$^+$93] Finkelstein A, Gabbay D, Hunter A, Kramer J, Nuseibeh B (1993) Inconsistency handling in multi-perspective specifications. In: Sommerville I, Paul M (eds) Software engineering—ESEC'93, lecture notes in computer science. Springer, Berlin, pp 84–99

[FGM$^+$07] Foster JN, Greenwald MB, Moore JT, Pierce BC, Schmitt A (2007) Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans Program Lang Syst 29(3):6

[FKM$^+$20] Fritsche L, Kosiol J, Möller A, Schürr A, Taentzer G (2020) A precedence-driven approach for concurrent model synchronization scenarios using triple graph grammars. In: Proceedings of the 13th ACM SIGPLAN international conference on software language engineering. Association for Computing Machinery, New York, NY, USA, pp 39–55

[FKN$^+$92] Finkelstein A, Kramer J, Nuseibeh B, Finkelstein L, Goedicke M (1992) Viewpoints: a framework for integrating multiple perspectives in system development. Int J Softw Eng Knowl Eng 2(1):31–57

[FKWVH19] Feldmann S, Kernschmidt K, Wimmer M, Vogel-Heuser B (2019) Managing inter-model inconsistencies in model-based systems engineering: application in automated production systems engineering. J Syst Softw 153:105–134

[FN05] Favre J-M, NGuyen T (2005) Towards a megamodel to model software evolution through transformations. Electron Notes Theor Comput Sci 127(3):59–74

[FST96] Finkelstein A, Spanoudakis G, Till D (1996) Managing interference. In: Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (viewpoints'96) on SIGSOFT'96 Workshops, ISAW'96. ACM, New York, NY, USA, pp 172–174

[GBB12] Goldschmidt T, Becker S, Burger E (2012) Towards a tool-oriented taxonomy of view-based modelling. In: Sinz E, Schürr A (eds) Modellierung 2012. Gesellschaft für Informatik e.V., pp 59–74. Accepted 14 Nov 2018. T09:41:29Z ISSN: 1617-5468

[GdLKP10] Guerra E, de Lara J, Kolovos DS, Paige RF (2010) Inter-modelling: from theory to practice. In: Petriu DC, Rouquette N, Haugen Ø (eds) MODELS'10, lecture notes in computer science. Springer, Berlin, pp 376–391

[GHJV95] Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston

[GHL10] Giese H, Hildebrandt S, Lambers L (2010) Toward bridging the gap between formal semantics and implementation of triple graph grammars. In: Validation 2010 workshop on model-driven engineering, verification, pp 19–24

[Gog73] Goguen JA (1973) Categorical foundations for general systems theory. In: Pichler F, Trappl R (eds) Advances in cybernetics and systems research, pp 121–130. Transcripta Books

[Gol06] Goldblatt R (2006) Topoi: the categorial analysis of logic. Dover, revised edition

[GW09] Giese H, Wagner R (2009) From model transformation to incremental bidirectional model synchronization. Softw Syst Model 8(1):21–43

[HEEO12] Hermann F, Ehrig H, Ermel C, Orejas F (2012) Concurrent model synchronization with conflict resolution based on triple graph grammars. In: de Lara J, Zisman A (eds) FASE 2012, Lecture notes in computer science. Springer, Berlin, pp 178–193

[Hei10a] Heindel T (2010) A category theoretical approach to the concurrent semantics of rewriting: adhesive categories and related concepts. PhD thesis, University of Duisburg-Essen

[Hei10b] Heindel T (2010) Hereditary pushouts reconsidered. In: Ehrig H, Rensink A, Rozenberg G, Schürr A (eds) Graph transformations, lecture notes in computer science. Springer, Berlin, pp 250–265

[HEO$^+$11] Hermann F, Ehrig H, Orejas F, Czarnecki K, Diskin Z, Xiong Y (2011) Correctness of model synchronization based on triple graph grammar. In: Whittle J, Clark T, Kühne T (eds) MODELS 2011. Springer, Berlin pp 668–682

[HP09] Habel A, Pennemann K-H (2009) Correctness of high-level transformation systems relative to nested conditions†. Math Struct Comput Sci 19(2):245–296

[HS18] Habel A, Sandmann C (2018) Graph repair by graph programs. In: Mazzara M, Ober I, Salaün G (eds) Software technologies: applications and foundations, lecture notes in computer science. Springer, Cham, pp 431–446

[ISO11] ISO/IEC JTC 1/SC 7 Software and systems engineering. Iso/iec/ieee 42010:2011 - systems and software engineering—architecture description. https://www.iso.org/standard/50508.html. Accessed Dec 2011

[Jac16] Jackson D (2016) Software abstractions: logic, language, and analysis. MIT Press

[KD17] König H, Diskin Z (2017) Efficient consistency checking of interrelated models. In: ECMFA 2017, pp 161–178

[KDRPP09] Kolovos DS, Di Ruscio D, Pierantonio A, Paige RF (2009) Different models for model matching: an analysis of approaches to support model differencing. In: Proceedings of the 2009 ICSE workshop on comparison and versioning of software models, CVSM'09. IEEE Computer Society, Washington, DC, USA, pp 1–6

[Ken91] Kennaway R (1991) Graph rewriting in some categories of partial morphisms. In: Ehrig H, Kreowski H-J, Rozenberg G (eds) Graph grammars and their application to computer science, lecture notes in computer science. Springer, Berlin, pp 490–504

[KFST19] Kosiol J, Fritsche L, Schürr A, Taentzer G (2019) Adhesive subcategories of functor categories with instantiation to partial triple graphs. In: Guerra E, Orejas F (eds) Graph transformation, lecture notes in computer science. Springer, pp 38–54

[KG19] Klare H, Gleitze J (2019) Commonalities for preserving consistency of multiple models. In: MODELS 2019 companion, pp 371–378

[KKL+21] Klare H, Kramer ME, Langhammer M, Werle D, Burger E, Reussner R (2021) Enabling consistency in view-based system development—the Vitruvius approach. J Syst Softw 171:110815

[KKT13] Kehrer T, Kelter U, Taentzer G (2013) Consistency-preserving edit scripts in model versioning. In: 2013 28th IEEE/ACM international conference on automated software engineering (ASE), pp 191–201

[KM18] Knapp A, Mossakowski T (2018) Multi-view consistency in UML: a survey. In: Graph transformation, specifications, and nets, LNCS 10800. Springer, Cham, pp 37–60

[KMCD19] Kienzle J, Mussbacher G, Combemale B, Deantoni J (2019) A unifying framework for homogeneous model composition. Softw Syst Model 18(5):3005–3023

[KPP06] Kolovos DS, Paige RF, Polack FAC (2006) Merging models with the epsilon merging language (EML). In: Nierstrasz O, Whittle J, Harel D, Reggio G (eds) Model driven engineering languages and systems, lecture notes in computer science. Springer, Berlin, pp 215–229

[KPP08] Kolovos D, Paige R, Polack F (2008) Detecting and repairing inconsistencies across heterogeneous models. In: Proceedings of the 2008 international conference on software testing, verification, and validation, ICST'08. IEEE Computer Society, Washington, DC, USA, pp 356–364

[KR17] Kosiol J, Radke H (2017) Rule-based repair of emf models: formalization and correctness proof. In: GCM 2017

[KS20] König H, Stünkel P (2020) Single pushout rewriting in comprehensive systems. In: Gadducci F, Kehrer T (eds) Graph transformation, lecture notes in computer science. Springer, Cham, pp 91–108

[Kü06] Kühne T (2006) Matters of (meta-)modeling. Softw Syst Model 5(4):369–385

[LAS17] Leblebici E, Anjorin A, Schürr A (2017) Inter-model consistency checking using triple graph grammars and linear optimization techniques. In: Proceedings of the 20th international conference on fundamental approaches to software engineering—Volume 10202. Springer, New York, NY, USA, pp 191–207

[LO14] Lambers L, Orejas F (2014) Tableau-based reasoning for graph properties. In: Giese H, König B (eds) Graph transformation, lecture notes in computer science. Springer, Cham, pp 17–32

[LS04] Lack S, Sobociński P (2004) Adhesive categories. In: Walukiewicz I (ed) Foundations of software science and computation structures, lecture notes in computer science. Springer, Berlin, pp 273–288

[LS06] Lack S, Sobociński P (2006) Toposes are adhesive. In: Corradini A, Ehrig H, Montanari U, Ribeiro L, Rozenberg G (eds) Graph transformations, lecture notes in computer science. Springer, Berlin, pp 184–198

[Lö93] Löwe M (1993) Algebraic approach to single-pushout graph transformation. Theor Comput Sci 109(1):181–224

[MC99] Mandel L, Cengarle MV (1999) On the expressive power of OCL. In: Wing JM, Woodcock J, Davies J (eds) FM'99—Formal methods, lecture notes in computer science. Springer, Berlin, pp 854–874

[MC16] Macedo N, Cunha A (2016) Least-change bidirectional model transformation with QVT-R and ATL. Softw Syst Model 15(3):783–810

[MJC17] Macedo N, Jorge T, Cunha A (2017) A feature-based classification of model repair approaches. IEEE Trans Softw Eng 43(7):615–640

[MWK+20] Meier J, Werner C, Klare H, Tunjic C, Aßmann U, Atkinson C, Burger E, Reussner R, Winter A (2020) Classifying approaches for constructing single underlying models. In: Hammoudi S, Pires LF, Selić B (eds) Model-driven engineering and software development, communications in computer and information science. Springer, Cham, pp 350–375

[NEF03] Nentwich C, Emmerich W, Finkelsteiin A (2003) Consistency management with repair actions. In: ICSE'03, pp 455–464

[NEFE03] Nentwich C, Emmerich W, Finkelsteiin A, Ellmer E (2003) Flexible consistency checking. ACM Trans Softw Eng Methodol 12(1):28–63

[NER01] Nuseibeh B, Easterbrook S, Russo A (2001) Making inconsistency respectable in software development. J Syst Softw 58(2):171–180

[OBE+13] Orejas F, Boronat A, Ehrig H, Hermann F, Schölzel H (2013) On propagation-based concurrent model synchronization. Electron Commun EASST 57:66

[Obj14] Object Management Group (2014) Business process model and notation (BPMN) v.2.0.2

[Obj15] Object Management Group (2015) Unified modeling language (UML) v.2.4.1

[Obj16a] Object Management Group (2016) Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) v.1.3. http://www.omg.org/spec/QVT/1.3

[Obj16b] Object Management Group (2016) Meta object facility (MOF) core specification v. 2.4.1

[Obj19] Object Management Group (2019) Decision model and notation (DMN) v.1.2

[OPKK18] Ohrndorf M, Pietsch C, Kelter U, Kehrer T (2018) ReVision: a tool for history-based model repair recommendations. In: Proceedings of the 40th international conference on software engineering: companion proceeedings, ICSE'18. Association for Computing Machinery, New York, NY, USA, pp 105–108

[OPN20] Orejas F, Pino E, Navarro M (2020) Incremental concurrent model synchronization using triple graph grammars. In: Wehrheim H, Cabot J (eds) Fundamental approaches to software engineering, lecture notes in computer science. Springer, Cham, pp 273–293

[Pen08] Pennemann K-H (2008) An algorithm for approximating the satisfiability problem of high-level conditions. Electron Notes Theor Comput Sci 213(1):75–94

[Pie91] Pierce BC (1991) Basic category theory for computer scientists. MIT Press, Cambridge

[PKR$^+$09] Paige RF, Kolovos DS, Rose LM, Drivalos N, Polack FAC (2009) The design of a conceptual framework and technical infrastructure for model management language engineering. In: Proceedings of the 2009 14th IEEE international conference on engineering of complex computer systems, ICECCS'09. IEEE Computer Society, Washington, DC, USA, pp 162–171

[RB01] Rahm E, Bernstein PA (2001) A survey of approaches to automatic schema matching. VLDB J 10(4):334–350

[RC13] Rubin J, Chechik M (2013) N-way model merging. In: ESEC/FSE 2013. ACM, New York, NY, USA, pp 301–311

[RE12] Reder A, Egyed A (2012) Computing repair trees for resolving inconsistencies in design models. In: 2012 Proceedings of the 27th IEEE/ACM international conference on automated software engineering, pp 220–229

[Roz97] Rozenberg G (1997) Handbook of graph grammars and computing by graph transformation, vol 1. World Scientific

[RR88] Robinson E, Rosolini G (1988) Categories of partial maps. Inf Comput 79(2):95–130

[RRLW09] Rutle A, Rossini A, Lamo Y, Wolter U (2009) A diagrammatic formalisation of MOF-based modelling languages. In: TOOLS EUROPE 2009. Springer, Berlin, pp 37–56

[RRLW12] Rutle A, Rossini A, Lamo Y, Wolter U (2012) A formal approach to the specification and transformation of constraints in MDE. JLAMP 81(4):422–457

[SBMP08] Steinberg D, Budinsky F, Merks E, Paternostro M (D2008) EMF: eclipse modeling framework. Pearson Education

[Sch94] Schürr A (1994) Specification of graph translators with triple graph grammars. In: WG'94, pp 151–163

[SDZKR18] Samimi-Dehkordi L, Zamani B, Kolahdouz-Rahimi S (2018) EVL+Strace: a novel bidirectional model transformation approach. Inf Softw Technol 100:47–72

[Seg92] Segen JC (1992) The dictionary of modern medicine. CRC Press

[SK03] Sendall S, Kozaczynski W (2003) Model transformation: the heart and soul of model-driven software development. IEEE Softw 20(5):42–45

[SKLR18] Stünkel P, König H, Lamo Y, Rutle A (2018) Multimodel correspondence through inter-model constraints. In: Conference companion of the 2nd international conference on art, science, and engineering of programming, Programming'18 Companion. Association for Computing Machinery, New York, NY, USA, pp 9–17

[SKLR20] Stünkel P, König H, Lamo Y, Rutle A (2020) Towards multiple model synchronization with comprehensive systems. In: FASE 2020, volume 12076 of lecture notes in computer science. Springer, Cham

[SKRL21] Stünkel P, König H, Rutle A, Lamo Y (2021) Multi-model evolution through model repair. J Obj Technol 20(1):1:1–25

[SLO18] Schneider S, Lambers L, Orejas F (2018) Automated reasoning for attributed graph properties. Int J Softw Tools Technol Transf 20(6):705–737

[SLO19] Schneider S, Lambers L, Orejas F (2019) A logic-based incremental approach to graph repair. In: Hähnle R, van der AW (eds) Fundamental approaches to software engineering, lecture notes in computer science. Springer, pp 151–167

[SMBB10] Silva Marcos Aurélio AD, Mougenot A, Blanc X, Bendraou R (2010) Towards automated inconsistency handling in design models. In: Advanced information systems engineering, lecture notes in computer science. Springer, Berlin, pp 348–362

[SNL$^+$07] Sabetzadeh M, Nejati S, Liaskos S, Easterbrook S, Chechik M (2007) Consistency checking of conceptual models via model merging. In: RE 2007, pp 221–230

[Ste08] Stevens P (2008) Bidirectional model transformations in QVT: semantic issues and open questions. Softw Syst Model 9(1):7

[Ste17] Stevens P (2017) Bidirectional transformations in the large. In: MODELS 2017, pp 1–11

[Ste20] Stevens P (2020) Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels. Softw Syst Model 6:66

[SZ01] Spanoudakis G, Zisman A (2001) Inconsistency management in software engineering: survey and open research issues. In: Handbook of software engineering and knowledge engineering, pp 329–380

[TA15] Trollmann F, Albayrak S (2015) Extending model to model transformation results from triple graph grammars to multiple models. In: ICMT'15, pp 214–229

[TA16] Trollmann F, Albayrak S (2016) Extending model synchronization results from triple graph grammars to multiple models. In: Van Gorp P, Engels G (eds) Theory and practice of model transformations, lecture notes in computer science. Springer, pp 91–106

[TOLR17] Taentzer G Ohrndorf M, Lamo Y, Rutle A (2017) Change-preserving model repair. In: Huisman M, Rubin J (eds) Fundamental approaches to software engineering, lecture notes in computer science. Springer, Berlin, pp 283–299

[TvdBS20] Torres W, van den Brand MGJ, Serebrenik A (2020) A systematic literature review of cross-domain model consistency checking by model management tools. Softw Syst Model 6:56

[UNKC08] Usman M, Nadeem A, Kim T, Cho E (2008) A survey of consistency checking techniques for UML models. In: Proceedings of the 2008 advanced software engineering and its applications, ASEA'08. IEEE Computer Society, USA, pp 57–62

[WAF$^+$19] Weidmann N, Anjorin A, Fritsche L Varró G, Schürr A, Leblebici E (2019) Incremental bidirectional model transformation with eMoflon: : IBeX. In: Cheney J, Ko H-S (eds) Proceedings of the 8th international workshop on bidirectional transformations co-located with the Philadelphia logic week, Bx@PLW 2019, Philadelphia, PA, USA, June 4, 2019, volume 2355 of CEUR workshop proceedings, pp 45–55. CEUR-WS.org

[Wal92] Walters RFC (1992) Categories and computer science. Cambridge University Press, New York

[WFA20] Weidmann N, Fritsche L, Anjorin A (2020) A search-based and fault-tolerant approach to concurrent model synchronisation. In: Proceedings of the 13th ACM SIGPLAN international conference on software language engineering. Association for Computing Machinery, New York, NY, USA, pp 56–71

[WHR14] Whittle J, Hutchinson J, Rouncefield M (2014) The state of practice in model-driven engineering. IEEE Softw 31(3):79–85

[WK99] Warmer J, Kleppe A (1999) The object constraint language: precise modeling with UML. Addison-Wesley Longman, Boston

[WK19] Weber JH, Kuziemsky C (2019) Pragmatic interoperability for ehealth systems: the fallback workflow patterns. In: Proceedings of the 1st international workshop on software engineering for healthcare, SEH'19, pp 29–36, Piscataway, NJ, USA. IEEE Press. Montreal, QC, Canada

[Wol21] Wolter U (2021) Logics of first-order constraints—a category independent approach. arXiv:2101.01944

[WWS+17] Wille D, Wehling K, Seidl C, Pluchator M, Schaefer I (2017) Variability mining of technical architectures. In: Proceedings of the 21st international systems and software product line conference—volume A, SPLC'17, pp 39–48, New York, NY, USA. ACM, Sevilla, Spain

Received 23 November 2020
Accepted in revised form 5 June 2021 by Jordi Cabot, Heike Wehrheim and Eerke Boiten
Published online 30 July 2021