algorithms

Article

Union Models for Model Families: Efficient Reasoning over Space and Time

Sanaa Alwidian [1], Daniel Amyot [2,*] and Yngve Lamo [3]

[1] Department of Electrical, Computer and Software Engineering, Ontario Tech University, Oshawa, ON L1G 0C5, Canada
[2] School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, ON K1N 6N5, Canada
[3] Department of Computer Science, Electrical Engineering and Mathematical Sciences, Western Norway University of Applied Sciences, 5020 Bergen, Norway
[*] Correspondence: damyot@uottawa.ca; Tel.: +1-613-562-5800 (ext. 6947)

Abstract: A model family is a set of related models in a given language, with commonalities and variabilities that result from evolution of models over time and/or variation over intended usage (the spatial dimension). As the family size increases, it becomes cumbersome to analyze models individually. One solution is to represent a family using one global model that supports analysis. In this paper, we propose the concept of union model as a complete and concise representation of all members of a model family. We use graph theory to formalize a model family as a set of attributed typed graphs in which all models are typed over the same metamodel. The union model is formalized as the union of all graph elements in the family. These graph elements are annotated with their corresponding model versions and configurations. This formalization is independent from the modeling language used. We also demonstrate how union models can be used to perform reasoning tasks on model families, e.g., trend analysis and property checking. Empirical results suggest potential time-saving benefits when using union models for analysis and reasoning over a set of models all at once as opposed to separately analyzing single models one at a time.

Keywords: model families; model evolution; union models; variability; analysis; graph theory; type graphs

Citation: Alwidian, S.; Amyot, D.; Lamo, Y. Union Models for Model Families: Efficient Reasoning over Space and Time. *Algorithms* **2023**, *16*, 105. https://doi.org/10.3390/a16020105

Academic Editor: Roberto Montemanni

Received: 28 December 2022
Revised: 8 February 2023
Accepted: 9 February 2023
Published: 11 February 2023

1. Introduction

Model-based Software Engineering (MBSE) is a software engineering paradigm that endorses the use of models as first-class artifacts [1]. A model can be defined with a domain-specific modeling language (DSML) [2], which provides modeling primitives to capture the semantics of a specific application domain (such as web-based languages) or can be defined using a general-purpose modeling language (such as the UML). The main goal of MBSE is to enhance the development, maintenance, and evolution of complex software systems by raising the level of abstraction from source code to models. In particular, the use of DSMLs allows developers to focus on their essential tasks while the recurring engineering tasks are lifted to higher abstraction levels and automatically generated by transformations specified by domain experts [3]. The use of the MBSE paradigm is beneficial because it allows engineers to focus on the product to be developed and on generic and reusable assets that express problems and solutions. Other measured or claimed benefits of MBSE include better communication, effectiveness, and quality compared to pure code-oriented approaches [4,5].

In MBSE, models expressed using any modeling language often undergo continuous evolution during their lifecycles. For example, such evolution can occur when new requirements are added or existing requirements are changed as engineers gain better understanding of the domain to be modeled. Models can evolve over *time*, resulting in a family of related models, which we refer to as a *model family*, with differences and similarities between family members. Another common source of model families is found in (software) product lines, where different configurations of a product can exist simultaneously (i.e., in the same *space* dimension) for different customers without necessarily being caused by evolution over time. Model families can result from having several alternatives due to design-time uncertainty or uncertainty at other stages of the development process. In such cases, all models need to exist together (as a model family) to tolerate a certain level of uncertainty until decisions absolutely need to be made [6]. Finally, a model family can appear as a result of *both* evolution over time and variation over space, such in the case of a set of software/product configurations that evolve over time [7].

The phenomenon of model families can be observed in several domains, including the *regulatory domain*, where regulations evolve over time and have variations (e.g., for different regulated parties or different versions of regulations) that need to be modeled using slightly different individual models, resulting in a *family* of related regulatory models [8,9]. In regulatory domains it is usually desirable to analyze models, for instance, for compliance assessment by regulators. However, the existence of several versions/variations of models makes it cumbersome and inefficient to analyze and reason about these models one at a time. This challenge was observed in practice through our previous experience in modeling and analyzing safety regulations at Transport Canada; the number of models in a model family can be very large due to multiple versions and variations that nonetheless share many elements in common. Exploiting this redundancy during analysis has the potential to reduce the effort and time required for analysis.

This paper is based on the first author's PhD thesis [10], and proposes *union models* as first-class generic artifacts to: (1) support the representation of model families (for time and space dimensions) using one generic model; (2) achieve performance gains during analysis of model families *all at once*, compared to analysis of individual models, *one model at a time*; and (3) support types of analyses that are more easily feasible with union models compared with individual models.

The use of union models is challenging and non-trivial. At the *model level*, the challenges stem mainly from the following requirements:

Req. 1 Models in a family shall be captured (in both dimensions of variability) in a *complete* and *exact* way such that all and only individual members of a family are included in one union model.

Req. 2 The resulting union model shall be as *compact* as possible, in the sense that it should not contain redundant elements, especially when there are many elements in common between models.

Req. 3 The union model shall be *self-explanatory*, i.e., it should be supported with a mechanism that distinguishes which elements belong to which models.

There are other challenges at the *metamodel level* associated with the use of union models. In particular, a union model may *not* be a valid instance of the language's metamodel, and the latter might need to have its constraints relaxed accordingly. The metamodel-level challenges have been already discussed in our previous work [11], and are outside the scope of this paper.

This paper extends previous work in [12] with a complete formalization of union models, including annotations, and enhanced experiments with a deeper analysis, and empirically demonstrates the usefulness of union models for analyzing a family of models *all at once* compared to individual models *one model at a time*. The main contributions of this paper are:

1. A *language-independent* graph-based formalization of model families and union models;
2. A generic language-independent *algorithm* to produce a union model from a set of models (in a compact and exact manner) in a given language (satisfying *Req. 1* and *Req. 2*);

3. A *Spatio-Temporal Annotation Language* (STAL) to support the representation of variability in model families in the space and time dimensions and to facilitate reasoning about union models (satisfying *Req. 3*); and
4. Improved *efficiency* of analysis and reasoning over a set of models all at once using union models, compared to reasoning on single models one model at a time.

As our approach starts with existing models as input, how such models are constructed in general is beyond the scope of this paper.

From a methodological perspective, this paper first presents the motivation behind this work in Section 2. Then, the theoretical and algorithmic foundations of our work are presented in the next three chapters, specifically, a graph-based formalization of model families (Section 3), a complementary Spatio-Temporal Annotation Language (Section 4), and the propositional encoding of models (Section 5). To determine how efficient and beneficial the reasoning and analysis with union models are, reasoning tasks and the experimental setup are first presented in Section 6, whereas empirical results and threats to validity are reported in Section 7. A discussion of closely related works is covered in Section 8. Finally, Section 9 concludes the paper.

2. Motivation

This work was inspired by issues faced previously with regulation modeling in collaboration with Transport Canada, where there are regulations that evolve over time and that apply to different types of organizations (i.e., spaces). We explain the challenges associated with regulatory model families and motivate our proposed solution using a running example from this domain, namely, airports regulated by Transport Canada, where we use the Goal-oriented Requirement Language (GRL) as a modeling language. GRL is a part of the User Requirements Notation (URN) standard [13].

Without loss of generality, the subsequent discussion about GRL model families in regulatory domains and their challenges is applicable to other model families from modeling languages other than the GRL. Therefore, we posit that our approach is feasible for any metamodel-based modeling languages and their model families.

At Transport Canada there are many regulations that need to be modeled by regulators, for example, to enable compliance and performance assessments. These regulations evolve over time, and apply to multiple types of organizations (such as aerodromes and airlines) of different sizes [8,9]. For example, as Figure 1 shows, certain rules/regulations apply differently depending on the targeted aerodrome type (configurations modeled abstractly here as TYPE1, TYPE2, and TYPE3), while other regulations are only applicable to specific aerodrome types. In Figure 1, regulations related to Fence rule 3 are only applicable to aerodrome TYPE1, Access Control System (ACS) rule 2 is applicable to all aerodrome types, and ACS rule 1 is applicable to TYPE1 and TYPE2 aerodromes but not to TYPE3 ones.

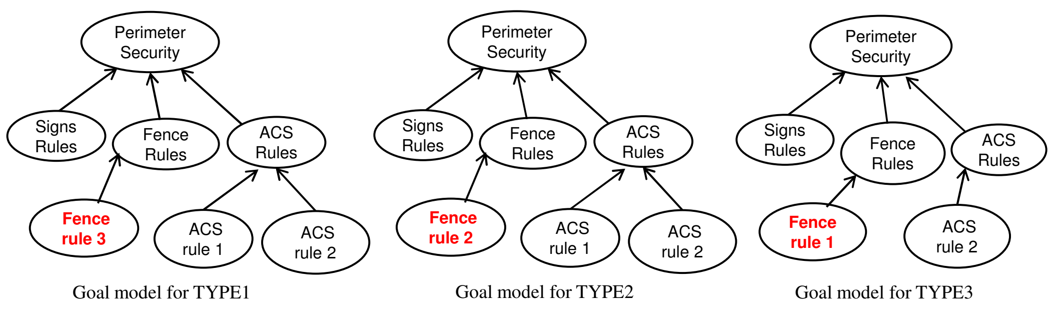


Figure 1. Goal model family of regulations.

The different aerodrome types (i.e., TYPE1, TYPE2, and TYPE3) can be considered as different *configurations* (or spaces) that are available at the same time for different aerodromes. This means that the three goal model variations depicted in Figure 1 represent a goal model family along the space dimension, where each space has different regulations.

Goal models in this family can evolve over time, e.g., when the regulatory context evolves, resulting in several versions of the same model. Evolution of goal models can involve the addition/deletion of goals and/or links or modifications to attributes of goals and/or links, as illustrated in each row of Figure 2. In this figure, the time-related changes across each of the initial configuration models (from Figure 1) are highlighted in Yellow.

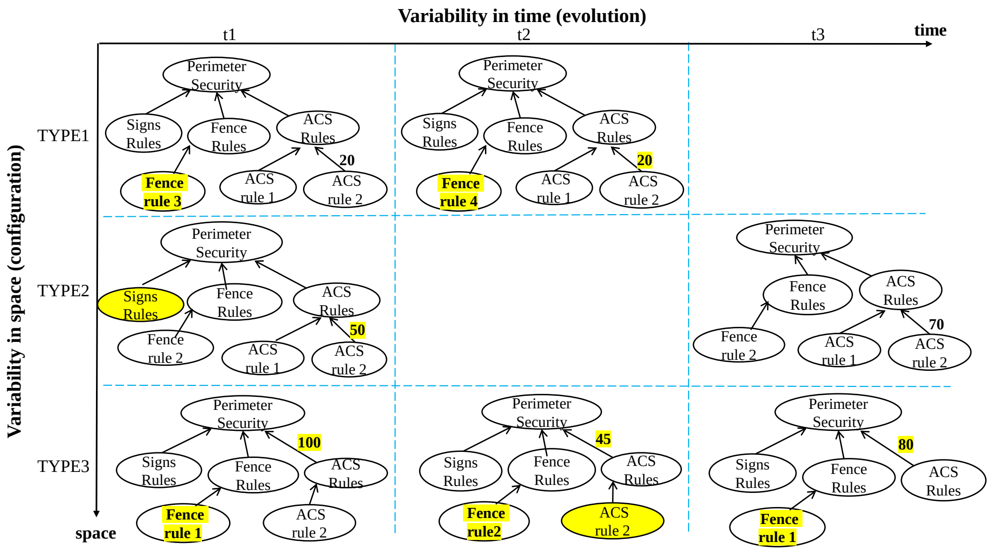


Figure 2. Goal model family of regulations with variability in space and time.

Analyzing the different versions/variations of models in Figure 2 individually using one goal model per type of aerodrome and at each time instance is impractical for the following reasons:

1. If a modeler plans to conduct goal satisfaction analysis (using some GRL forward propagation algorithm [13]) on each individual model by running a strategy that initially assigns ACS rule 2 to study the impact of its satisfaction on the satisfaction of other goals, she would end up running the same evaluation algorithm six times for this example, even though there are several common elements among the seven models. Intuitively, if there are M individual models in a model family and each model has E elements, then the complexity of running a satisfaction propagation algorithm on all models would be on the order of $\mathcal{O}(M \times E)$. Such complexity becomes more significant if there are hundreds of models with hundreds of elements in each model, which is not an atypical situation.
2. Each individual model (e.g., goal model for TYPE1 at time t1) does not represent the whole set of regulations per se. Hence, a regulator who wishes to reason about all regulations (e.g., to study the evolution trend of regulations over time) would have to check all models one model at a time and reason about each one separately. This process becomes inefficient and time consuming as the number of models increases.
3. Models in a model family are subject to frequent evolution over time that are asynchronous by nature. Such asynchronous evolution sometimes requires that older versions of models (which represent legacy models) need to be maintained, as they may remain in use even after being superseded by newer versions. This is an issue over the time dimension, and can be witnessed in the space dimension as well (i.e., along configurations). In this scenario, legacy models and new models need to co-exist together in one model in order to be analyzed together.

Similar challenges exist in the software engineering field, particularly in the Software Product Line (SPL) domain, where products vary over the space dimension and may evolve over time as well. By nature, an SPL encodes a set of related product variants; thus, dealing with the evolution of multiple products over time means that developers should consider an additional dimension of variability by reasoning about sets of products. This combination of variation and evolution is not well supported by existing SPL engineering techniques, and only a few recent approaches in the literature have tried to address this issue. For example, Seidl et al. [14] and Lity et al. [15] considered variation of SPLs in space and time and proposed a so-called 175% modeling formalism to allow for the development and documentation of evolving product lines. In their paper, they urged the community to extend annotative variability modeling to tackle evolution and variation by the same means. Famelis et al. [16] proposed an approach for combining SPLs with partial models to represent design-time uncertainty. The resulting artifact, called Software Product Lines with Design Choices (SPLDCs), integrates variability and design-time uncertainty in a common formalism to help developers differentiate between the kinds of decisions that are relevant during the design and configuration stages of the SPL lifecycle.

The above-mentioned three challenges (especially the first one) and the gaps identified in the literature motivate the need to find a way to represent model families other than using separate individual models. This paper proposes *union models* (M_U) as a generic modeling artifact to capture all model elements in all members of a model family MF, aggregated in a compact and exact way such that analysis of a group of models is faster than for individual models and where members of a family can be extracted and analyzed. Generally speaking, a union model M_U of any potential model family would be the union of all elements e in all individual models M_i of the family MF; that is,

$$M_U = (\bigcup_{i=1}^{|MF|} \bigcup_{j=1}^{|M_i|} e_{i,j}) \tag{1}$$

In addition, the elements of the resulting union model need to be annotated in a way that enables the identification of variants. For illustration purposes, an M_U that captures the model family shown in Figure 1 is represented in Figure 3, with all elements (i.e., goal and links) of the union model included and annotated with space information (TYPES in this example) to distinguish which element belongs to which model variant.

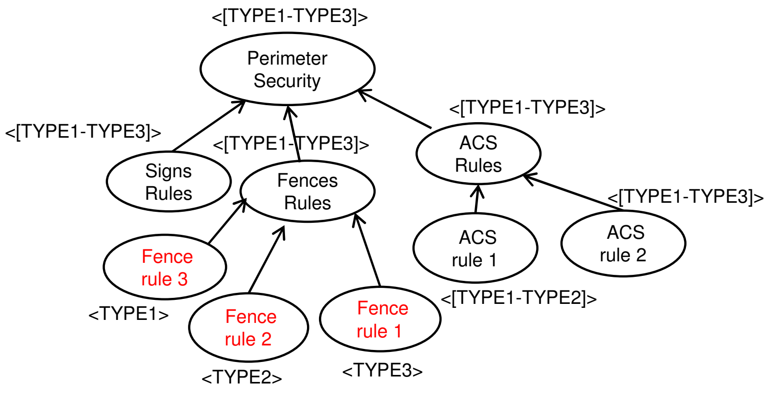


Figure 3. A union model for the model family in Figure 1.

Note that each member of the model family is embedded in the union model, i.e., for each $M_i \in MF$ there exists an embedding $i : M_i \to M_U$ defined by $i(e) = e$. Moreover, we can retract any model M_i in the family from the union model by the partial mapping $r_i : M_U \to M_i$, where $r_i(e) = e$ if e is annotated with M_i and is undefined otherwise. Hence, the union model is complete in the sense that it includes all information available in any member of the modeling family, and is complete in the sense that we can reconstruct each member of modeling family from it.

3. Formalization of Model Families

This section illustrates the use of graphs and type(d) graphs to formalize basic (meta)modeling concepts. Figure 4 illustrates the basic relationships between models and metamodels and their graphical representation. The definitions of graphs and type/typed graphs are then used as a basis for further formal definitions of model families and their union models. To model more detailed aspects of model families, we extend the definitions of basic type(d) graphs with attributes (i.e., attributed type graphs, or *E-graphs* [17]). The definitions that follow are based on [18–20].

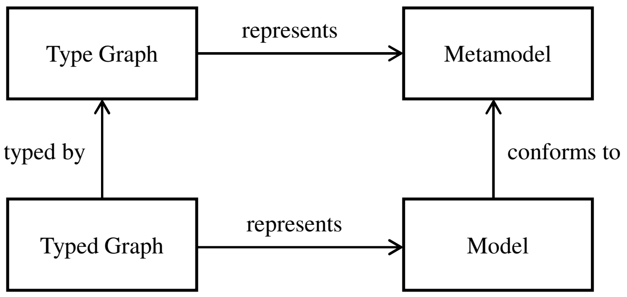


Figure 4. Relationship between models and their graphical representation.

Definition 1. ***Graph***: *A graph is a tuple $G = (N_G, E_G, src_G, tgt_G)$, where N_G is a set of graph nodes (or vertices), E_G is a set of graph edges, and functions src_G, tgt_G: $E_G \to N_G$ associate to each edge a source and a target node, respectively, such that $e\colon x \to y$ denotes an edge e with $src_G(e) = x$ and $tgt_G(e) = y$.*

Definition 2. ***Graph morphism***: *Let G and H be two graphs. A graph morphism $f\colon G \to H$ consists of a pair of functions (f_N, f_E) with $f_N\colon N_G \to N_H$ and $f_E\colon E_G \to E_H$ that preserves sources and targets of edges when composed ($\circ$), i.e., $f_N \circ src_G = src_H \circ f_E$ and $f_N \circ tgt_G = tgt_H \circ f_E$. In other words, for each edge $e_G \in E_G$ there is a corresponding edge $e_H = f_E(e_G) \in E_H$ such that $src_G(e_G)$ is mapped to $src_H(e_H)$ and $tgt_G(e_G)$ is mapped to $tgt_H(e_H)$. This is illustrated in Figure 5.*

$$\begin{tikzcd}
N_G \arrow[r, "f_N"] & N_H \\
E_G \arrow[u, "src_G"] \arrow[r, "f_E"'] \arrow[ur, phantom, "="] & E_H \arrow[u, "src_H"']
\end{tikzcd}
\qquad
\begin{tikzcd}
N_G \arrow[r, "f_N"] & N_H \\
E_G \arrow[u, "tgt_G"] \arrow[r, "f_E"'] \arrow[ur, phantom, "="] & E_H \arrow[u, "tgt_H"']
\end{tikzcd}$$

Figure 5. Graph morphism between (directed) graphs.

3.1. Type Graphs and Typed Graphs

In graph theory (such as in strongly-typed programming languages, where each constant or variable is assigned a data type), it is often useful to determine the well-formedness of a graph by checking whether it conforms to a so-called *type graph*. A type graph is a distinguished graph containing all the relevant types and their interrelations [21]. This is analogous to the relationship between models and metamodels in the model-driven engineering world, where each model (e.g., a UML design model) needs to conform to a metamodel (e.g., of the UML language). The correspondence between both ideas is depicted in Figure 4.

Definition 3. ***Type graph (metamodel)***: *A type graph TG is a distinguished graph, where $TG = (N_{TG}, E_{TG}, src_{TG}, tgt_{TG})$, and N_{TG} and E_{TG} are types of nodes and edges, respectively.*

Definition 4. ***Typed graph (model)***: *A typed graph is a triple $G_{typed} = (G, type, TG)$ such that G is a graph (Definition 1) and type: $G \to TG$ is a graph morphism (Definition 2) called the typing morphism. The typed graph G_{typed} is called an instance (graph) of (the graph) TG, and we denote the set of all instance graphs of TG as Inst[TG].*

Figure 6b shows an example of a typed graph typed over the type graph to its left (Figure 6a). The type graph in Figure 6a represents the types of nodes and edges. There are two types of nodes here, namely, State and Transition, and two types of association edges, namely, source and target. Node names and types in the typed graph are depicted inside the nodes as name: type. For instance, in the node labeled with S1: State, S1 is the name of a node and State is the type of that node. For the edges, names and types of edges are represented in the same way.

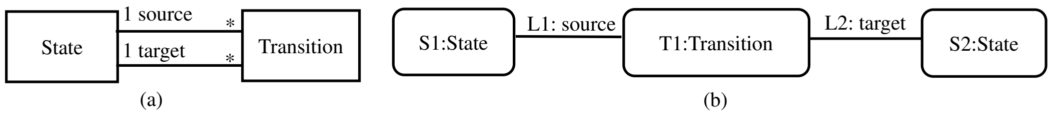


Figure 6. Type graph (**a**) and Typed graph (**b**). The star (*) indicates that zero or more instances of the class "Transition" are associated with one instance of the class "State".

Definition 5. ***Typed graph morphism***: *Given a type graph TG, and two typed graphs G_{typed} and H_{typed} (typed over TG), a typed graph morphism g: (G_{typed}, t_G: $G_{typed} \to TG$) $\to$ (H_{typed}, t_H: $H_{typed} \to TG$) is a graph morphism g: $G_{typed} \to H_{typed}$ that also preserves typing, i.e., $g \circ t_H = t_G$, as illustrated in Figure 7.*

$$\begin{tikzcd}
 & TG & \\
 & = & \\
G_{typed} \arrow[uur, "t_G"] \arrow[rr, "g"'] & & H_{typed} \arrow[uul, "t_H"']
\end{tikzcd}$$

Figure 7. Typed graph morphism.

3.2. Attributed Type(d) Graphs as E-Graphs

Given a typed graph (i.e., a model), it is useful in practice to attach additional information to nodes and edges by *attributing* them, such that each node and/or edge can contain zero or more attributes. In this case, we refer to *typed attributed graphs*, where attributes are typically a *name:value* pair that allows to attach a specific value to each attribute name. In addition, given a typed attributed graph that is typed over some *TG* (i.e., a metamodel), that *TG* needs to constrain the names and types of attributes that are allowed for certain types of nodes and edges. In this context, we refer to an *Attributed Type Graph (ATG)*, as shown in Figure 8, which extends the type graph illustrated in Figure 6a with node attributes.

As an example of a typed attributed graph, reconsider the typed graph in Figure 6b extended with attributes as in Figure 9, where nodes have attributes with names tname, sname.

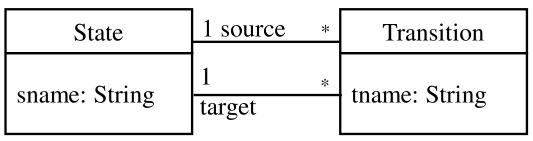


Figure 8. An attributed type graph (ATG). The star (*) indicates that zero or more instances of the class "Transition" are associated with one instance of the class "State".

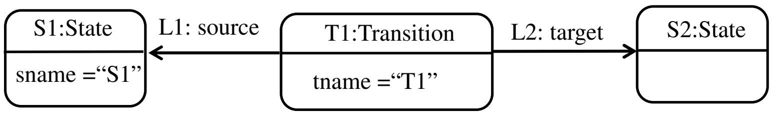


Figure 9. A typed attributed graph, typed over the ATG in Figure 8.

In this paper, we adopt the definition of Ehrig et al. [22] **for E-graphs to represent ATGs** (note that from now on in this paper, the concept of ATG implicitly means an attributed type graph that is represented as an E-graph), which allows attribution for both nodes and edges, where attributes of nodes (resp. edges) are represented as *special edges* between graph nodes (resp. graph edges) and data nodes that represent the data types of these attributes. Figure 10 visualizes the concept of E-graphs.

Definition 6. ***E-graph***: *An E-graph is a tuple EG = (N_G, N_D, E_G, E_{NA}, E_{EA}, (src_j, tgt_j) $j \in \{G, NA, EA\}$) where:*

- *N_G and N_D are graph nodes and data nodes, respectively;*
- *E_G, E_{NA}, and E_{EA} are graph edges, node attribute edges, and edge attribute edges, respectively;*
- *src_j and tgt_j, where $j \in \{G, NA, EA\}$ are source and target functions that assign to each of the three categories of edges (i.e., E_G, E_{NA}, and E_{EA}) a source and a target, as follows:*
  - *src_G: $E_G \to N_G$, tgt_G: $E_G \to N_G$ for graph edges;*
  - *src_{NA}: $E_{NA} \to N_G$, tgt_{NA}: $E_{NA} \to N_D$ for node attribute edges;*
  - *src_{EA}: $E_{EA} \to E_G$, tgt_{EA}: $E_{EA} \to N_D$ for edge attribute edges.*

$$\begin{tikzcd}
E_G \arrow[rr, bend left, "src_G"] \arrow[rr, bend right, "tgt_G"'] & & N_G \\
E_{EA} \arrow[u, "src_{EA}"] \arrow[r, "tgt_{EA}"'] & N_D & E_{NA} \arrow[u, "src_{NA}"'] \arrow[l, "tgt_{NA}"]
\end{tikzcd}$$

Figure 10. A visualization of an E-graph.

Example: The ATG in Figure 8 is represented as the E-graph illustrated in Figure 11. The figure shows the different categories of nodes and edges defined in Definition 6, namely:

- Graph nodes (N_G): State, Transition
- Data nodes (N_D): String
- Graph edges (E_G): source, target
- Node attribute edges (E_{NA}): sname, tname

In addition, attributes of nodes are represented as special edges between graph nodes and data nodes that represent the type that attributes. For example, the attribute *tname* is represented as an edge between graph node *Transition* and the data node *String*.

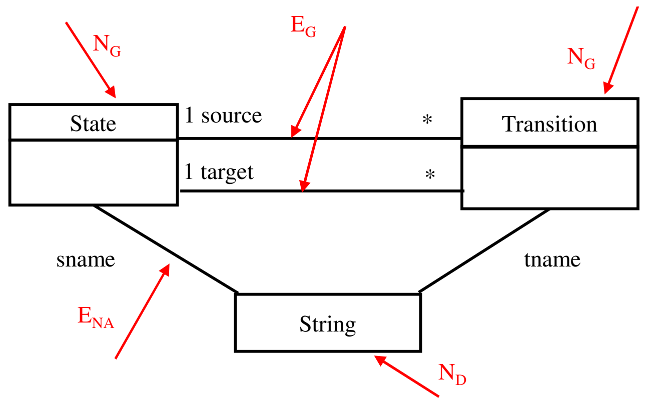


Figure 11. An E-Graph, EG, with node attributes represented as edges. The star (*) indicates that zero or more instances of the class "Transition" are associated with one instance of the class "State".

An instance typed graph ITG (i.e., model) of the attributed E-graph EG in Figure 11 is illustrated in Figure 12, where this figure is a representation of the typed attributed graph found in Figure 9. We use the notation Inst[EG] to denote the set of all graphs conforming to EG; hence, ITG ∈ Inst[EG].

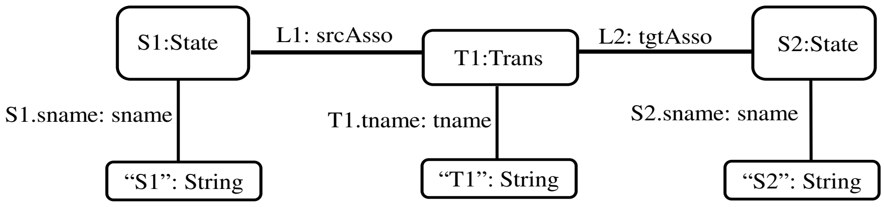


Figure 12. An instance typed graph ITG of the E-graph EG in Figure 11.

3.3. Formalization of Model Families and Union Models

In this paper, we consider model families that consist of an arbitrary set of *homogeneous* models that conform to the same metamodel. In that sense, we can define a model family as a set of typed attributed graphs that are instances of the same attributed type graph, ATG, and satisfy the constraints of that type graph.

Definition 7. ***Model Family***: *A model family is a tuple MF = ($\{G_{typed}\}$, type, ATG), where each $G_{typed} \in \{G_{typed}\}$ is in Inst[ATG] and type: $\{G_{typed}\} \rightarrow$ ATG.*

Definition 8. ***Union Model*** **(M_U)**: *Let MF be a model family with two models such that MF = ($\{G1, G2\}$, type, ATG), where type: G_1, $G_2 \rightarrow$ ATG (represented as an E-Graph) and where:*

- *G1 = (N_{G1}, N_{D_G1}, E_{G1}, E_{NA_G1}, E_{EA_G1}, (src_j,tgt_j), where $j \in \{G, NA, EA\}$, $type_{G1}$)*
- *G2 = (N_{G2}, N_{D_G2}, E_{G2}, E_{NA_G2}, E_{EA_G2}, (src_j,tgt_j), where $j \in \{G, NA, EA\}$, $type_{G2}$)*

  In addition, G1 and G2 satisfy the following conditions:

- ***Cond. 1***: *If two nodes have the same name and the same type, then these nodes are considered identical. Note that we assume that each node and each edge has its own unique identifier; for simplicity, we express this identity by means of a unique name.*
- ***Cond. 2***: *If two edges have the same name and the same type and if they connect between the same source and target nodes, these edges are considered identical.*

If the above conditions are satisfied, then the union model that represents MF is a model M_U = G1 ⊔ G2 = (N_U, E_U, src_U, tgt_U, $type_U$) such that N_U = ($N_{G1} \cup N_{G2} \cup N_{D\_G1} \cup N_{D\_G2}$), E_U = ($E_{G1} \cup E_{G2} \cup E_{NA\_G1} \cup E_{NA\_G2} \cup E_{EA\_G1} \cup E_{EA\_G2}$), and the functions src_U, tgt_U, and $type_U$ are:

$$src_U(e) = \begin{cases} src_{G1}(e) & if\ e \in E_{G1} \\ src_{NA\_G1}(e) & if\ e \in E_{NA\_G1} \\ src_{EA\_G1}(e) & if\ e \in E_{EA\_G1} \\ src_{G2}(e) & if\ e \in E_{G2} \\ src_{NA\_G2}(e) & if\ e \in E_{NA\_G2} \\ src_{EA\_G2}(e) & if\ e \in E_{EA\_G2} \end{cases}$$

$$tgt_U(e) = \begin{cases} tgt_{G1}(e) & if\ e \in E_{G1} \\ tgt_{NA\_G1}(e) & if\ e \in E_{NA\_G1} \\ tgt_{EA\_G1}(e) & if\ e \in E_{EA\_G1} \\ tgt_{G2}(e) & if\ e \in E_{G2} \\ tgt_{NA\_G2}(e) & if\ e \in E_{NA\_G2} \\ tgt_{EA\_G2}(e) & if\ e \in E_{EA\_G2} \end{cases}$$

$$type_U(elem) = \begin{cases} type_{G1}(e) & \textit{if elem} \in N_{G1} \cup N_{D\_G1} \cup E_{G1} \cup E_{NA\_G1} \cup E_{EA\_G1} \\ type_{G2}(e) & \textit{if elem} \in N_{G2} \cup N_{D\_G2} \cup E_{G2} \cup E_{NA\_G2} \cup E_{EA\_G2} \end{cases}$$

Provided that conditions Cond. 1 and Cond. 2 in Definition 6 are respected, the union operation can be generalized into an MF of arbitrary size. That is, given $MF = (\{G1, G2, \ldots, Gn\}, type, ATG)$, its union model is $M_U = G1 \sqcup G2 \sqcup G3 \ldots \sqcup Gn$.

The union operation is *incremental*. That is, given a union model M_U already constructed for a particular family, any upcoming model M_i added to that family is unified incrementally with M_U such that the new union model becomes $M_{Unew} = M_U \sqcup M_i$. The annotations of M_U and M_i are unified as well, which is discussed in the next section. Consequently, the incremental nature of the union operation allows for the merging of two or more individual models, a model and a union model, or two union models.

It is important to mention here that even if the typed graphs used to construct models are well-formed, there is no guarantee that their union M_U will be a well-formed model. In fact, M_U will conform to the *typing constraints* imposed by the *ATG*, and may not conform to other constraints such as multiplicities of attributes and/or association ends, or to other external OCL constraints. This metamodel-level issue is discussed in more detail in [10].

4. Spatio-Temporal Annotation Language (STAL)

With an appropriate formalization of model families (as discussed in Section 3), the challenging part of constructing a union model is not necessarily in the union operation itself. Rather, the challenge is in being able to distinguish the models to which a particular element belongs. For example, in the goal model family shown in Figure 1, we need to distinguish that Fence rule 3 belongs only to aerodromes of TYPE 1 (i.e., configuration 1), Fence rule 2 belongs only to TYPE 2, etc. In addition, if there are common elements among all models (such as ACS rule 2) we need to indicate this in M_U in a precise and exploitable manner.

To achieve this, we propose a *Spatio-Temporal Annotation Language (STAL)* to annotate elements of each individual model with information about their versions and/or configurations in the form of <ver$_{\text{num}}$, conf$_{\text{info}}$>, where ver$_{\text{num}}$ denotes the version number (time dimension) of a particular model (e.g., 1st version, 2nd version, etc.), while conf$_{\text{info}}$ denotes space dimension-related information (e.g., organization type, size, location).

In the time dimension, models can evolve independently and asynchronously over distinct timepoints. Because timepoints can be correlated and compared, they naturally form a chronological order. Considering this inherent chronological nature of models evolution, a sequence of versions of a particular model can be annotated with sequential version numbers: ver$_1$, ver$_2$, ... ver$_n$. This creates an implicit temporal validity between model versions; for instance, we can say that ver$_1$ happened before ver$_2$. The timing information embedded in the <ver$_{\text{num}}$> format in STAL can represent version numbers or dates, a hierarchical version numbering scheme (e.g., versions 2.3.1 and 4.3.2, etc.), or ranges thereof.

The space dimension, on the other hand, is different and somewhat more complex. This stems from the fact that the space dimension is flat and has neither a chronological order nor a hierarchical nature (except in very specific domains, such as in provinces and their cities). In STAL, we usually use the naming conventions confA, confB, ..., confZ instead of conf1, conf2, ..., confn to reflect the lack of ordering semantics. If a configuration is simple, we use its syntactical description as a name for that configuration. For example, confA="airports in Ontario" and confB="airports in Quebec" are the names of the two different configurations of airports.

However, it is worth mentioning that information about configurations can be composite, i.e., it may consist of several pieces of information. For example, TYPE1 aerodromes may refer to those airports that are of medium-size, located in Ontario, and with national flights only. To represent this type of composite information in STAL in a way that keeps annotated models as simple as possible, we propose the use of look-up tables, and approach that provides mappings between configuration names and their real descriptions. Please note that, in this example, the numbering suffixes of TYPEs do not hold any ordering meanings; they are only descriptions of the configuration. Table 1 shows an example of a look-up table for the configurations in Figure 1.

Table 1. Mapping configurations to their detailed descriptions.

Config	Description
TYPE1	Size = M, Location = Ontario, Flight = national
TYPE2	Size = L, Location = Ontario, Flight = international
TYPE3	Size = M, Location = Quebec, Flight = national

In addition, in a model family it is possible that one model element belongs to several or all family members; see the full STAL grammar in Appendix A. For instance, assume that there is a model family with one model configuration (confA) that evolves into five versions (i.e., ver_1 to ver_5). Moreover, assume a node n that belongs to the five versions of that model. In this case, n will typically be annotated in the union model with five annotations: <ver_1, confA>, <ver_2, confA>, <ver_3, confA>, <ver_4, confA>, <ver_5, confA>. Such a style may lead to large amounts of annotations per element. To *simplify* annotations of union models, the representation of STAL annotations can be shortened such that a sequence of version annotations is represented as a range of values ([start:end]). In the above example, the annotation of n becomes <[ver_1: ver_5], confA>. Ranges are, however, unavailable for configurations, as they are usually not sortable.

In the same example, it could happen that an element, say, edge e, appears in all versions from ver_1 to ver_8 except in ver_4. In this scenario, a set of ranges can be used such that e is annotated with <{[ver_1: ver_3], [ver_5:ver_8]}, confA>. Furthermore, if an element x belongs to ver_1 to ver_3 of confA and to versions ver_1 to ver_4 of confB, then that element will be annotated as <[ver_1:ver_3], (confA, confB)>; <ver_4, confB>. Finally, if an element belongs to all versions and/or all configurations of a family, we annotate it with the keyword *ALL*.

The ranging mechanism used with versions may be applicable to configurations as well if their nature allows for continuous or discrete ranging, such as TYPE1, TYPE2, etc. However, to avoid confusion between versions and configurations we do not use ranges to annotate configurations in this paper. Rather, we use a comma-separated list to indicate a set of configurations, e.g., <(confA, confB, confD)>.

5. Propositional Encoding Language with Annotations (PELA)

To facilitate reasoning about models and to realize a simple graph union in practice, we encode typed graphs (i.e., models) as logical propositions. Such encoding provides a concrete syntax for defining models and metamodels. To encode a model m into propositional logic, we need to first map elements in m into propositional variables and then join them. To achieve this, we propose a *propositional encoding language with annotations* (PELA), which defines specific naming conventions for the propositional encoding of variables, where propositions themselves are annotated with STAL (see Section 4). While defining the syntax of PELA, we took into consideration that this language should be reversible, that is, a modeler should be able to retrieve (or decode) a model back from its propositional encoding. This operation, however, is not supported by the tools in this paper; such tool support, albeit not difficult to implement, is left for future work.

5.1. Definitions

The propositional encoding of models using PELA is defined as follows:

Definition 9. ***ElementToPropositionWithAnnotation***: *Given a model M = (G, type, ATG), where G = (N_G, N_D, E_G, E_{NA}, (src_j, tgt_j), $j \in \{G, NA, EA\}$) (see Definition 6), together with* *a STAL annotation specifying version numbers and configuration information, the mapping of elements of M into propositions with annotation ElementToPropositionWithAnnotation(elem) is defined according the following syntactical rules (along with their semantics):*

- *A graph node $n \in N_G$ of type $t \in N_{ATG}$ is mapped into a propositional variable "t–<ver_{num}, $conf_{info}$>" to express the semantics: "a model (with ver_{num} and $conf_{info}$) contains a node n of type t". Formally: n–t iff $\exists n \in N_G \wedge type(n) = t$.*
- *A graph edge $e \in E_G$ of type $t \in E_{ATG}$ with source node x and target node y is mapped into a propositional variable "e–x–y–t–<ver_{num}, $conf_{info}$>" to express the semantics: "a model (with ver_{num} and $conf_{info}$) contains an edge e of type t from node x to node y". Formally: e–x–y–t iff $\exists e \in E_G \wedge type(e) = t \wedge src_G(e) = x \wedge tgt_G(e) = y$.*
- *A data node $dn \in N_D$ of type $t \in dataType$ owned by a graph node $n \in N_G$ is mapped into a propositional variable "dn–n–t–<ver_{num}, $conf_{info}$>" to express the semantics: "a model (with ver_{num} and $conf_{info}$) contains a node n that owns a data node dn of type t". Formally: dn–n–t iff $\exists n \in N_G \wedge owner(dn) = n \wedge type(dn) = t$.*
- *A node attribute edge $nae \in E_{NA}$ of type $t \in attribute\_name$ that is represented as a special edge between a graph node $n \in N_G$ and a data node $dn \in N_D$, where n is also the owner of that attribute, is mapped into a propositional variable "nae–n–n–dn–t–<ver_{num}, $conf_{info}$>" to express the semantics: "a model (with ver_{num} and $conf_{info}$) contains a node n which owns an attribute nae of type t, and this nae is represented as an edge from graph node n to data node dn". Formally: nae–n–n–dn–t iff $\exists nae \in E_{NA} \wedge owner(nae) = n \wedge type(nae) = t \wedge src(nae) = n \wedge tgt(nae) = dn$. It is worth clarifying that the "n–n" part in the pattern nae–n–n–dn–t represents the same element n. The first n indicates the owner of the attribute, and the second n indicates that this owner is also a source of the edge. We use this syntax to distinguish the node attribute edge as a special edge distinct from the ordinary graph edge.*

For example, if model M in Figure 12 is the second version of an initial model, say, M_0, and represents configuration X, then the propositional encoding of state S1 in that model is *S1–State–<ver_2, $conf_X$>* and the propositional encoding of a node attribute edge S1.sname is *S1.sname–S1–S1–"S1"–name–<ver_2, $conf_X$>*.

It is important to emphasize here that model M could be a union model rather than an individual model. According to Definition 8, because a union model M_U is a model, the propositional encoding rules discussed above should be applicable to M_U, with the exception that the annotations of M_U elements are expressed as full STAL annotations instead of single model annotations. This means that the four rules stated in Definition 9 preserve their syntax and semantics when applied to M_U, except that the annotation format changes from single ver_{nums} and single $conf_{info}$ into ranges, sets, or lists of ver_{num} and lists of $conf_{info}$.

Based on Definition 9, we can now define the mapping of an entire graph to propositions (with annotations), as follows:

Definition 10. ***GraphToPropositionWithAnnotation****: Given a typed graph G, a mapping of G's elements into a set of propositions is GraphToPropositionWithAnnotation(G) = $\{ElementToPropositionWithAnnotation(elem) \mid elem \in N_G \cup N_D \cup E_G \cup E_{NA} \cup E_{EA}\}$.*

For example, the propositional encoding of the model in Figure 12 (repeated here for convenience) is shown to the left of Figure 13. We assume that the model here represents the second version of a given initial model and represents configuration X; that is, each of its elements is annotated with *<ver_2, $conf_X$>*.

```
GraphToPropositionwithAnnotation(M) = {
  S1–State–<ver2, confx>,
  S2–State–<ver2, confx>,
  T1–Trans–<ver2, confx>,
  "S1"–S1–String–<ver2, confx>,
  "T1"–T1–String–<ver2, confx>,
  "S2"–S2–String–<ver2, confx>,
  L1–T1–S1–src–<ver2, confx>,
  L2–T1–S2–tgt–<ver2, confx>,
  S1.sname–S1–S1–"S1"–sname–<ver2, confx>,
  T1.tname–T1–T1–"T1"–tname–<ver2, confx>,
  S2.sname–S2–S2–"S2"–sname–<ver2, confx>}
```

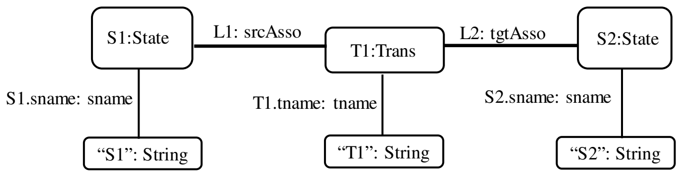


Figure 13. *GraphToPropositionWithAnnotation* encoding of model M in Figure 12.

5.2. Union of Propositional Encodings of Models

Given the propositional encoding of models discussed in the previous section, the union operation simply becomes the union of the propositional encodings of individual models, as follows:

Definition 11. ***Proposition Encoding Union (PE_U)***: *Let MF be a model family of two models G1 and G2 (Definition 8), where G1 and G2 are attributed typed graphs with the same metamodel ATG, and let GraphToPropositionWithAnnotation(G1) and GraphToPropositionWithAnnotation(G2) be their propositional encodings (Definition 10); then, the union of the propositional encodings with annotations G1 and G2 is: PE_U = GraphToPropositionWithAnnotation(G1) ∪ GraphToProposition-WithAnnotation(G2).*

We can generalize the above definition to a set of arbitrary encoded models, where the union of the propositionally encoded models is annotated according to the grammar of STAL following the approach in Section 4. Furthermore, because the union operation is incremental, a proposition encoding union (PE_U) can be unified with other individual models or even with other PE_Us. For instance, given a PE_U of a set of propositionally encoded models and a new model M_i encoded as *GraphToPropositionWithAnnotation(M_i)*, their union becomes $PE_{Unew} = PE_U \cup PE_{M_i}$. The annotations of PE_U and M_i are unified according to STAL.

5.3. Example

This section provides a simple yet complete example for two versions of state transition diagrams, *M1* and *M2*. The example illustrates the formalization of both models as E-graphs. In addition, it illustrates their encoding into propositional variables as well as their union. In this example, *M1* and *M2* are assumed to be the first and the second version of a model that together represent configuration A. Hence, the elements of *M1* and *M2* are respectively annotated with $<ver_1, conf_A>$ and $<ver_2, conf_A>$.

Figure 14 represents *M1* in the conventional representation of state transition diagrams, Figure 15 illustrates the representation of *M1* as a canonical typed attributed E-graph, and Figure 16 represents *M1*'s propositional encoding.

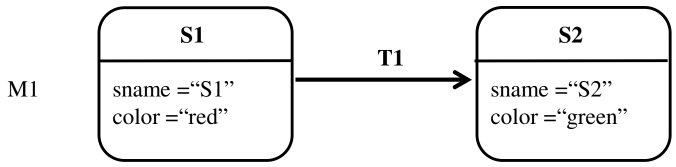


Figure 14. First version of a state transition diagram, M1.

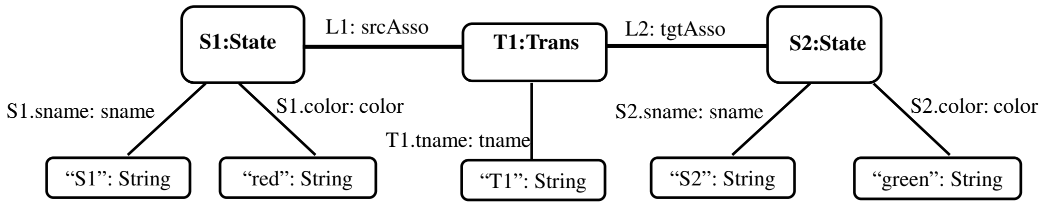


Figure 15. Representation of M1 as an E-graph.

GraphToPropositionwithAnnotation(M1) = {
S1–State–<ver$_1$, conf$_A$>, S2–State–<ver$_1$, conf$_A$>,
"S1"–S1–String–<ver$_1$, conf$_A$>, "red"–S1–String–<ver$_1$, conf$_A$>,
"S2"–S2–String–<ver$_1$, conf$_A$>, "green"–S2–String–<ver$_1$, conf$_A$>,
T1–Trans–<ver$_1$, conf$_A$>, "T1"–T1–String–<ver$_1$, conf$_A$>,
L1–T1–S1–srcAsso–<ver$_1$, conf$_A$>, L2–T1–S2–tgtAsso–<ver$_1$, conf$_A$>,
S1.sname–S1–S1–"S1"–sname–<ver$_1$, conf$_A$>,
S1.color–S1–S1– "red"–color–<ver$_1$, conf$_A$>,
S2.sname–S2–S2– "S2"–sname–<ver$_1$, conf$_A$>,
S2.color–S2–S2–"green"–color–<ver$_1$, conf$_A$>,
T1.tname–T1–T1– "T1"–tname–<ver$_1$, conf$_A$> }

Figure 16. Propositional encoding of M1.

In the same manner, Figure 17 shows the conventional representation of *M2* as a state transition diagram, Figure 18 represents it as an E-graph, and Figure 19 details its propositional encoding.

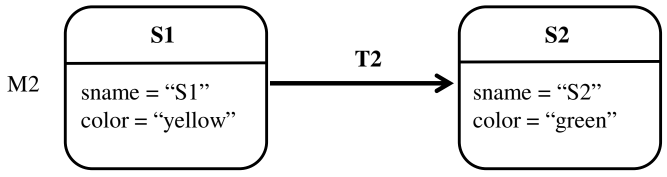


Figure 17. Second version of a state transition diagram, M2.

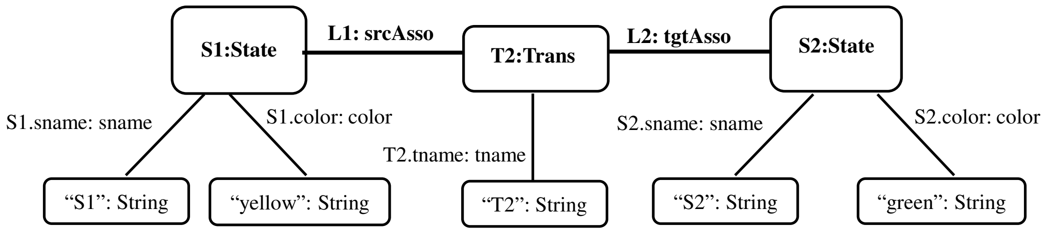


Figure 18. Representation of M2 as an E-graph.

GraphToPropositionwithAnnotation(M2) = {
S1–State–<ver$_2$, conf$_A$>, S2–State–<ver$_2$, conf$_A$>,
"S1"–S1–String–<ver$_2$, conf$_A$>, "yellow"–S1–String–<ver$_2$, conf$_A$>,
"S2"–S2–String–<ver$_2$, conf$_A$>, "green"–S2–String–<ver$_2$, conf$_A$>,
T2–Trans–<ver$_2$, conf$_A$>, "T2"–T2–String–<ver$_2$, conf$_A$>,
L1–T2–S1–srcAsso–<ver$_2$, conf$_A$>, L2–T2–S2–tgtAsso–<ver$_2$, conf$_A$>,
S1.sname–S1–S1–"S1"–sname–<ver$_2$, conf$_A$>,
S1.color–S1–S1–"yellow"–color–<ver$_2$, conf$_A$>,
S2.sname–S2–S2–"S2"–sname–<ver$_2$, conf$_A$>,
S2.color– S2–S2–"green"–color–<ver$_2$, conf$_A$>,
T2.tname–T2–T2–"T2"–tname–<ver$_2$, conf$_A$> }

Figure 19. Propositional encoding of M2.

After encoding *M1* and *M2*, we can now construct their union model M_U as the union of their propositional encodings with annotations (Definition 11), as shown in Figure 20:

PE_U=*GraphToPropositionWithAnnotation(G1)*∪ *GraphToPropositionWithAnnotation(G2)* = {
S1–State–<ALL>, S2–State–<ALL>,
"S1"–S1–String–<ALL>,
"red"–S1–String–<ver$_1$, conf$_A$>, "yellow"–S1–String–<ver$_2$, conf$_A$>,
"S2"–S2–String–<ALL>, "green"–S2–String–<ALL>,
T1–Trans–<ver$_1$, conf$_A$>, "T1"–T1–String–<ver$_1$, conf$_A$>,
T2–Trans–<ver$_2$, conf$_A$>, "T2"–T2–String–<ver$_2$, conf$_A$>,
L1–T1–S1–srcAsso–<ver$_1$, conf$_A$>, L2–T1–S2–tgtAsso–<ver$_1$, conf$_A$>,
L1–T2–S1–srcAsso–<ver$_2$, conf$_A$>, L2–T2–S2–tgtAsso–<ver$_2$, conf$_A$>,
S1.sname–S1–S1–" S1"–sname–<ALL>, S1.color–S1–S1–"red"–color–<ver$_1$, conf$_A$>,
S1.color–S1–S1–"yellow"–color–<ver$_2$, conf$_A$>, S2.sname–S2–S2–"S2"–sname–<ALL>,
S2.color–S2–S2–"green"–color–<ALL>,
T1.tname–T1–T1–"T1"–tname–<ver$_1$, conf$_A$>,
T2.tname–T2–T2–"T2"–tname–<ver$_2$, conf$_A$> }

Figure 20. Union of the propositional encodings of M1 and M2.

The representation of the union model as an E-graph is depicted (with annotations) in Figure 21, and its conventional representation as an ordinary state transition diagram (again with annotations) is shown Figure 22.

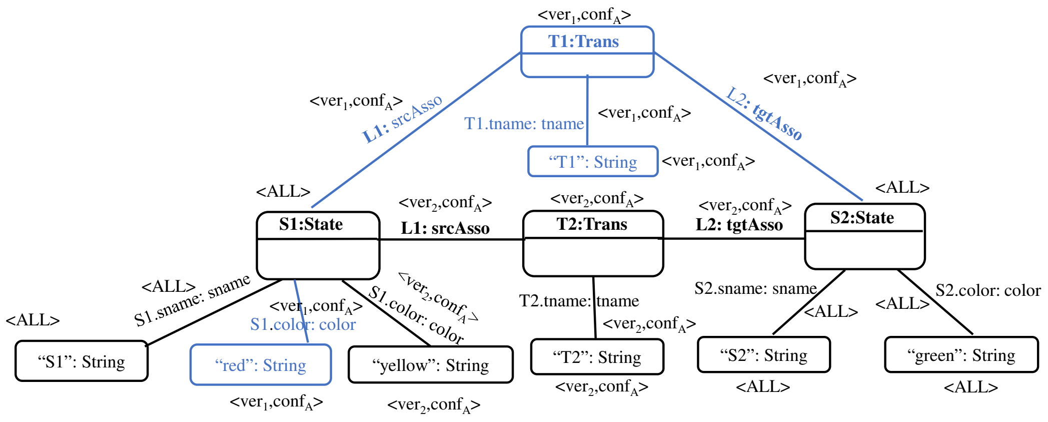


Figure 21. Representation of M_U as an E-graph.

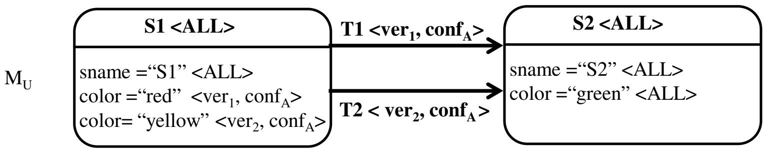


Figure 22. Conventional representation of M_U as an annotated state transition diagram.

6. Analysis and Reasoning with Model Families

This section explores the research question: *How efficient is reasoning and analysis with a group of models all at once using M_U in comparison to the use of individual models?*

To this end, Section 6.1 defines three reasoning tasks in order to evaluate their performance, first using union models and then using individual model several times. Section 6.2 discusses our experimental setup, methodology, and implementation. The next chapter then presents our results. A preliminary version of this section and the next one has previously been published in [12].

6.1. Reasoning Tasks

To answer the research question mentioned above, we need to describe how a union model can facilitate analysis and reasoning with sets of models instead of only single models. To achieve this, we consider three *reasoning tasks* (RTs), namely, *property checking*, *trend analysis*, and *commonality analysis*. Then, we compare the performance of the three RTs using M_U as opposed to using individual models.

Although these kinds of analyses can be performed using individual models several times, one model at a time, our objective here is to make these analyses more efficient using M_U. In addition, we aim to reduce the effort needed for loading each model into a tool, analyzing the model, saving the analysis results, and then moving to the next model, especially as this effort *cannot* be neglected with a large number of models. These manual steps are, however, not considered in our results; as such, our results and performance improvements are conservative.

6.1.1. RT1: Property Checking

Property checking of models aims to verify whether or not a model satisfies a particular property. Given a model m and a property p, the result of property checking is either True if m satisfies p or False otherwise. For instance, a modeler may want to check whether or not a group of state machine diagrams contains self-looping edges, or she may check whether there exist two or more different actors in a GRL model family that contain the same goal. In these scenarios, property checking is beneficial to help modelers understand, for example, what is common between model versions or variations that violate a property.

In this section, we limit ourselves to language-independent syntactic properties (which describe the structure of models) other than semantic properties (which describe the behavior of models, e.g., traces). The rationale behind this scoping is because our approach aims to be applicable to any metamodel-based modeling language. However, while there exists a standard approach for defining the syntax of a modeling language (i.e., through metamodeling), there is no common approach for specifying semantics. Thus, we limit our approach to checking those properties related to a language syntax independently from any language specificity. Hence, "property" here means "syntactic property". Alwidian [10] further discusses and evaluates semantic aspects for one particular language.

To perform property checking, we assume that a property p (expressed in any constraint language such as FOL or OCL) can be grounded over the vocabulary of models. Hence, a corresponding propositional formula Φp can be obtained. For example, given a well-formedness constraint $\Phi p = \{\forall t : Transition, \exists s : State \mid t.src = s\}$, it can be grounded over the vocabulary of the model in Figure 23 as follows:

$$T1\text{-}S1\text{-}S2\text{-}Transition \implies S1\text{-}State \wedge T2\text{-}S2\text{-}S3\text{-}Transition \implies S2\text{-}State$$

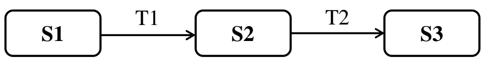


Figure 23. An example of propositional encoding of a model (Φm).

As can be noticed, the example considers the graphical representation of the state machine presented in the canonical form in Figure 6b, where transitions T1 and T2 are represented here as directed edges between states and not as nodes.

Formally speaking, given the propositional encoding of both models (Section 5) and properties, the task of property checking can be defined as follows:

Definition 12. ***Property Checking**: Given a model m, property p, and their respective propositional encodings Φm and Φp, we check whether the expression $\Phi m \wedge \Phi p$ is satisfiable or not using a SAT solver.*

For RT1, inspired by van der Straeten et al. [23], we checked the "*cyclic composition property*", which ensures that "the model does not contain self-looping edges", i.e., $\Phi p = \{\forall e : E_G, src_G(e) \neq tgt_G(e)\}$. It is important to highlight here that checking the self-looping edge property in individual models and in their M_U is a simple reasoning task that always produces the same result; that is, if a self-looping edge exists in any of the individual models, then it will be detected in the M_U that captures these individual models, and vice versa. This case, however, is not necessarily true for other properties, such as for example the general acyclicity property.

Figure 24 shows two simple state machine models that are acyclic (i.e., cycle-free) along with their union model M_U that is not acyclic (i.e., has a cycle). In this scenario, checking the acyclicity property in individual models M1 and M2 produces a different result than checking the same property in M_U. To address this issue, there is a need to explicitly consider annotations on elements during property checking to determine the actual occurrence of cycles. For instance, if the link in model M1 is annotated with $<v_1>$ and the link in M2 is annotated with $<v_2>$, then both links in M_U will be considered as different and not as an actual cycle, even though they seem to constitute a structural cycle. A real cycle only exists when the intersection of the annotations of the links involved is not empty.

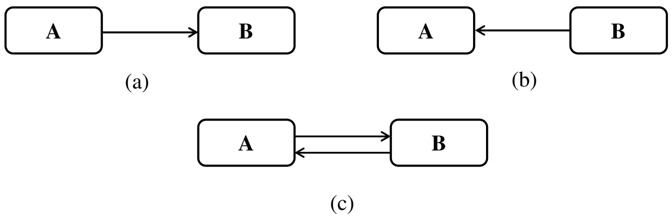


Figure 24. (**a**) An acyclic model M1, (**b**) an acyclic model M2, and (**c**) a union model of M1 and M2 that is cyclic. Note: these three models satisfy Φp, as they have no self-loops.

In the presence of annotations, property checking is more complicated and further research is needed to investigate its complexity. That is, we need to characterize the conditions under which this kind of property checking is beneficial when using M_U in comparison to using individual models multiple times. The findings of Famelis et al. [6] regarding the analysis of syntactic properties of partial models indicate that the size of the model family might be an important factor; in smaller model families, it is better to analyze each individual model separately. In the future we want to investigate whether other factors, such as the degree of redundancy across the family or the shape of the models and property expressions, might play a role. In addition, counterexamples produced by model-checkers for union models that would violate a property would be harder to interpret than usual, as they might cover multiple models in the family. How to interpret such counterexamples in terms of individual models is left for future work.

6.1.2. RT2: Trend Analysis

This analysis aims to search for a particular element across members of a model family and to study the *trend* of that element, i.e., the behavior of elements over space/time. In other words, a trend analysis studies how properties of elements change over the course of time or across configurations. For instance, a modeler may need to search for a particular goal, say, GoalX, in all members of a GRL family in order to conduct a trend analysis about the properties of that goal (e.g., its importance/satisfaction value) and to observe how that value changes across model versions/variations in order to obtain insights into its evolution pattern.

6.1.3. RT3: Commonality Analysis

We suggest this type of analysis to enable modelers to check for those elements that are common in all (or part) of versions or variations of models in a family. This type of analysis is aligned with the commonality-based analysis in the SPL domain, where commonality is a key metric that indicates the reuse ratio of a feature across the SPL [24]. Following the same rationale, in this analysis it can be inferred that elements found to be common among the majority of models are important. For example, if a modeler is investigating several design options of a particular system and needs to know which elements are important in design options, then she would conduct this analysis *once* using the M_U of the model family she has at hand instead of carrying out a pairwise search on each version/variation of individual models. To be fair, such analysis obviously needs to take into consideration the construction time of the union model.

6.2. Analysis and Experiments

We assessed the feasibility of reasoning using M_U empirically by running experiments with parameterized random inputs that simulated different settings of various reasoning and analysis categories. In this chapter, we build on the formalization of union models in Section 3.3 and use formalized GRL models and state machine models. Our approach, however, is not bound to these languages, and is applicable to other metamodel-based languages.

6.2.1. Methodology

To evaluate the feasibility of using M_U with the three reasoning tasks RT1, RT2, and RT3, we first measured the total time (in seconds) needed to perform each one of the RTs on each individual model one model at a time; we refer to this time as T_{ind}. Next, we measured the time needed to accomplish the same task with M_U; we refer to this time as T_{MU}. Then, we computed the performance improvements using the time metrics *speedup* (as used by Famelis in [25], defined originally as *speedup* = T_{ind} / T_{MU}) and *time saving* (in minutes for RT1 and in seconds for RT2 and RT3, calculated as *TimeSaving* = ($T_{ind} - T_{MU}$)).

Here, we define $T_{construct}$ as the time needed to construct M_U. Although $T_{construct}$ is usually quite small and can be performed once before being amortized over multiple analyses, we distinguish two categories of experiments, named **Exp.1** and **Exp.2**. In Exp.1, we consider $T_{construct}$, such that the speedup is calculated as *speedup_with_constrTime*= T_{ind} / ($T_{construct}$ + T_{MU}). In Exp.2, on the other hand, we neglect the time needed to construct M_U, where the speedup is calculated here (same as in [25]) as *speedup_without_constrTime* = T_{ind}/T_{MU}. In both experiments, a speedup larger than 1 is a positive result, and the larger the speedup, the better the improvement.

The reason why $T_{construct}$ is considered here is to be more fair and realistic in the experiments, especially for large models, where it becomes necessary to not neglect the time needed to construct $M_{U.}$ Another reason is to compare the results of both categories of experiments in order to reach a conclusion on whether or not to always neglect $T_{construct}$. As mentioned, the time that an analyst would need to analyze models individually in a realistic context by loading the model in a tool, performing the analysis, and saving the results is not taken into consideration in T_{ind}.

For both experiments, we considered the following experimental parameters: (1) the size of individual models (*SIZE*), which represents the number of elements (i.e., nodes and edges) in each individual model, and (2) the number of individual models in a model family (*INDV*). To control the possible combinations of parameters *SIZE* and *INDV* and to facilitate reporting, we followed the methodology proposed by Famelis et al. [6,25] to discretize the parameters' domain into categories, where the ranges of values for the *INDV* parameter were set based on pilot experiments conducted by Famelis et al. [6] to determine reasonable ranges of individual models that constitute a family. Regarding the ranges of values for the *SIZE* parameter, we relied on our own experience with goal models, where models with a few dozen elements are considered to be small, while models with many hundreds or more elements are deemed to be extra-large.

For parameter *SIZE*, four categories were defined based on the number of nodes and edges, as follows: small (S), medium (M), large (L), and extra-large (XL). To calculate the ranges of each size category, we performed experiments with a seed sequence (0, 5, 10, 20, 40). The boundaries of each category were calculated from successive numbers of the seed sequence using the formula $n \times (n + 1)$. Using the same formula, a representative exemplar of each category is calculated by setting n to be the mean (rounded up) of two successive numbers in the seed sequence. We followed the same methodology for the number of individual models, *INDV*, using a seed sequence (0, 4, 8, 12, 16). The four size categories (S, M, L, XL) are shown in Table 2. The ranges of all categories of *SIZE* and *INDV* and the selected exemplars for each category are shown in Table 3. These ranges (generated from the seeds mentioned above) are in line with our own real experience dealing with goal models and state machines of various sizes. The same can be said for the number of individual models; a family is considered small when it contains a handful of models, and it is considered very large when it contains hundreds of models or more.

Table 2. Categories of SIZE parameter (number of elements in a model).

#Elements/Model (SIZE)	**(0, 30]**	**(30, 110]**	**(110, 420]**	**(420, 1640]**
Exemplar	12	56	240	930
Category	S	M	L	XL

Table 3. Categories of INDV parameter (number of individual members in a family).

# of Indiv. Models (INDV)	**(0, 20]**	**(20, 72]**	**(72, 156]**	**(156, 272]**
Exemplar	6	42	110	210
Category	S	M	L	XL

To evaluate the property checking task (i.e., RT1), each annotated individual model m in a model family MF was encoded as a propositional logic formula, namely, $\Phi m = \bigwedge ElementToPropositionWithAnnotation(e_i), e_i \in m$, where e_i are elements of the model m. A union model M_U of that MF was encoded as $\Phi M_U = \bigwedge ElementToPropositionWithAnnotation(e_i), e_i \in M_U$. Furthermore, the property to be checked was encoded into a propositional formula Φp. Afterwards, an SAT solver was used to check whether or not the encodings of each of the individual model and their union model satisfied the property. In particular, a formula $\Phi m \wedge \Phi p$ was constructed for each individual model. The property was said to hold in any model if and only if this formula was satisfiable. Similarly, a formula $\Phi M_U \wedge \Phi p$ was constructed and checked against whether the property was satisfiable. In both experiments, using the same computer settings, the time it took to check a property on individual models (T_{ind}) was recorded and compared to the time needed to carry out the check on union models (T_{MU}).

6.2.2. Implementation

To validate our approach, we used the NetworkX 2.2 Python library [26] to implement attributed typed graphs (according to Definition 5); we implemented our own union algorithm on top of that library to construct M_U (based on Definitions 9 and 11). NetworkX 2.2 is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex graphs [26]. It is enriched with a variety of features, from the support of graph data structures and algorithms to analysis measures to visualization options.

To ensure that we had a family of state machine models, NetworkX's graph generator was applied to randomly generate valid attributed typed graphs with different *SIZE* and *INDV* parameters. These graphs corresponded to typed state machines with *likely evolutions*, i.e., a sequence of random but typical manipulations on state machine models that lead to different versions. A sample of the generated graphs was manually checked to make sure that we had generated likely changes to existing models rather than generating completely independent models. Due to the completely random nature of the graph generators, we found that although the amount of changes in state machine models could be controlled, the topology of the resulting graph could not; that is, the same node in a model could change its incoming and/or outgoing edges randomly, leading to a different model. Such deviations between individual models can lead to a union model with large variations, indicated by the number of annotations on each element. Although this is not the best or even the typical case of state machine model families, we decided to perform experiments on the generated models in order to examine the complex families, for which the performance is unlikely to be worse.

For GRL models, the generation of models was less random; we took a set of real GRL models of smart homes published in [12] as a starting point and enlarged them according to the different combinations of the *SIZE* and *INDV* parameters. While growing the models, we created a set of random realistic modifications that involved adding and/or deleting intentional elements and/or element links or modifying their attributes. We then constructed $\mathrm{G_U}$ from the generated graphs using our union algorithm. $\mathrm{G_U}$ is the union of a set of typed graphs; hence, $\mathrm{G_U}$ corresponds to M_U.

It is worth mentioning here that the random generation of graphs was only used to check the scalability of the approach in order to provide a proof of concept for the suggested approach. Properties checked on such random graphs may be satisfied or not; this is not important here as long as the results for the model family are the same as the results for individual models, which is the case here.

For RT1, we checked the cyclic composition property inspired by [23], which ensures that the model does not contain self-looping edges. A propositional formula ($\Phi p = \{\forall e : E_G, src_G(e) \neq tgt_G(e)\}$) was generated for this property. The propositional encodings were generated according to the rules discussed in Section 5, and were fed as literals to the *MiniSAT* solver included in the *SATisPY* package [27]. SATisPy is a Python library that provides an interface for various SAT solver applications.

To build confidence about the property checking results, the graphs and their union model were tested to check for the existence of any self-looping edges. Output solutions retrieved for individual models of a particular family were compared to the solutions returned for the corresponding union model. The results were the same. All experiments were executed on a laptop with an Intel Core i5-8250U (8th Gen) 1.6GHz quad-core CPU and 8 GB RAM running Windows 10 $\times$ 64.

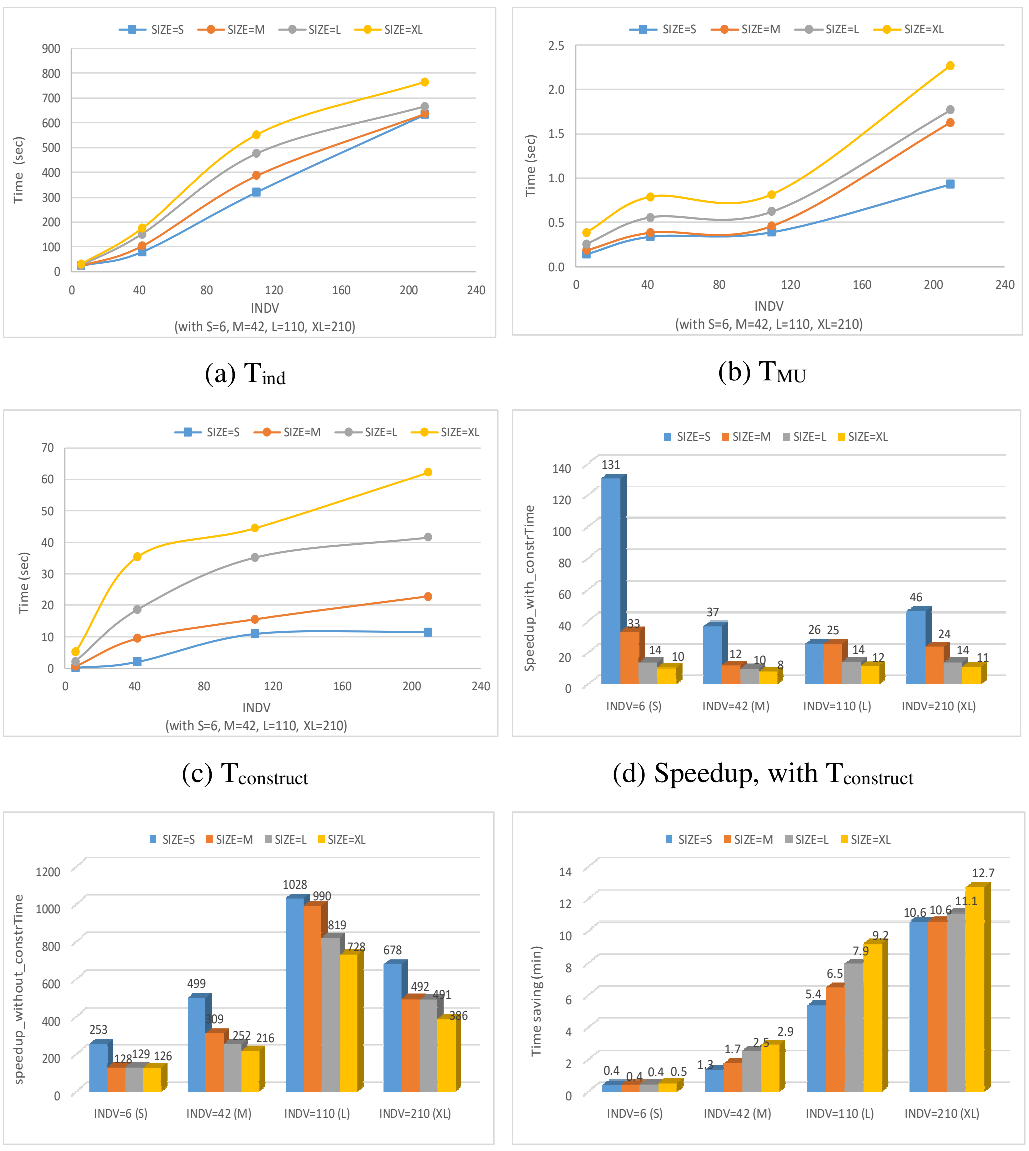
The next three sections are organized according to the experiments conducted to evaluate RT1, RT2, and RT3. All figures illustrated below represent a summary of the average results of fifteen runs represented for all *SIZE* and *INDV* categories together.

7. Results and Discussion

The results of the three reasoning tasks defined earlier are reported in Sections 7.1–7.3. Section 7.4 highlights the potential threats to validity. Finally, Section 7.5 provides a summary and a discussion of important points related to the current results and means of improving them in the future.

7.1. Results for Property Checking (RT1)

This section reports on our empirical results for the property checking reasoning task. Figure 25a illustrates $\mathrm{T_{ind}}$, which is the total time performing property checking on each individual model, for all *SIZE* and *INDV* categories. Figure 25b reports on $\mathrm{T_{MU}}$, which is the time needed to perform property checking on union models that capture individual models of different *SIZE* and *INDV*. In addition, Figure 25c shows the time $\mathrm{T_{construct}}$ needed to construct these union models. Figure 25d shows the time speedup with $\mathrm{T_{construct}}$, i.e., *speedup_with_constrTime* = T_{ind} /($T_{construct}$ + T_{MU}), while Figure 25e shows the speedup without considering $\mathrm{T_{construct}}$, calculated as *speedup_without_constrTime* = T_{ind} / T_{MU}. Finally, Figure 25f highlights the time saving in minutes achieved by using M_U to perform property checking calculated as *TimeSaving* = ($T_{ind} - T_{MU}$).



(a) T_{ind} (b) T_{MU}

(c) $T_{construct}$ (d) Speedup, with $T_{construct}$

(e) Speedup without $T_{construct}$ (f) Time saving

Figure 25. Results for property checking (RT1) for all *INDV* and *SIZE* categories.

Figure 25a–c shows that T_{ind}, T_{MU}, and $T_{construct}$ increase when the size of models (i.e., *SIZE*) or the number of models in a family (i.e., *INDV*) increase. In addition, it can be noticed that the increase of T_{ind} is important for each *SIZE* category, as the *INDV* parameter grows from *INDV* = S to *INDV* = XL. For instance, with *SIZE* = XL, we observe 30.85 s on average for *INDV* = S to 765 s on average for *INDV*=XL. On the other hand, the increase of T_{MU} as *INDV* grows is marginal for each *SIZE* category. For example, for XL-sized models T_{MU} increases from 0.39 s (for *INDV* = S) to 2.25 s (for *INDV* = XL).

Furthermore, it can be inferred from Figure 25d,e that the use of M_U for property checking achieves a noticeable time speedup compared to performing the same task on a set of individual models separately. For *speedup_with_constrTime* (Figure 25d), the highest speedup (=131) was observed with a small number of individual models (i.e., *INDV* = S) that are of a small size (i.e., *SIZE* = S). The smallest speedup (=8) was observed when *INDV* = M and *SIZE* = XL. In addition, Figure 25d shows that there is a noticeable pattern of speedup degradation for each *INDV* category as the number of elements per individual model (i.e., *SIZE*) increases. This is due in part to the increase of $T_{construct}$ as the *SIZE* increases. Nevertheless, the speedup never falls below 1, which means that even with very large models (with *INDV* = XL and *SIZE* = XL) the time to perform property checking on a group of such models (using M_U), considering the time to construct M_U, is better than performing property checking on all individual models. Figure 25e shows that the time speedup becomes more significant when $T_{construct}$ is ignored, where the highest *speedup_without_constrTim* = 1028 is with models of *SIZE* = S and *INDV* = L. This high value of 1028, which is higher than the number of models in family (110), is caused in part because of a low denominator value with low resolution (e.g., 1028 = 308.4/0.3), as well as by the time taken by the Python environment to load before executing the code of each individual model, which is not negligible, particularly for small models.

It is important to emphasize here that the erratic behavior of the time speedup with and without $T_{construct}$ across all categories of *SIZE* and *INDV* does not necessarily mean that one category is superior to the other. This is because speedup reflects the ratio between T_{ind} and T_{MU} (and $T_{construct}$ in case of calculating *speedup_with_constrTime*), and could fluctuate because of different densities of variability in the models.

It is not necessary that the speedup with *INDV* = S should be always better than that of *INDV* = M, *INDV* = L, etc., or vice versa, as the topology of the models generated may influence efficiency and this aspect is not controlled in this experiment. To this end, the speedup metric is used in this paper to demonstrate the improvements achieved by using M_U in general, without regarding of the particular behavior of this improvement.

The *TimeSaving* metric, however, can be relied on to observe the behavior of the time gained from using M_U to perform a particular task as opposed to using individual models multiple times. Figure 25f clearly illustrates a consistent pattern of time savings that increases when *SIZE* and *INDV* increase, which is the result that we were hoping for.

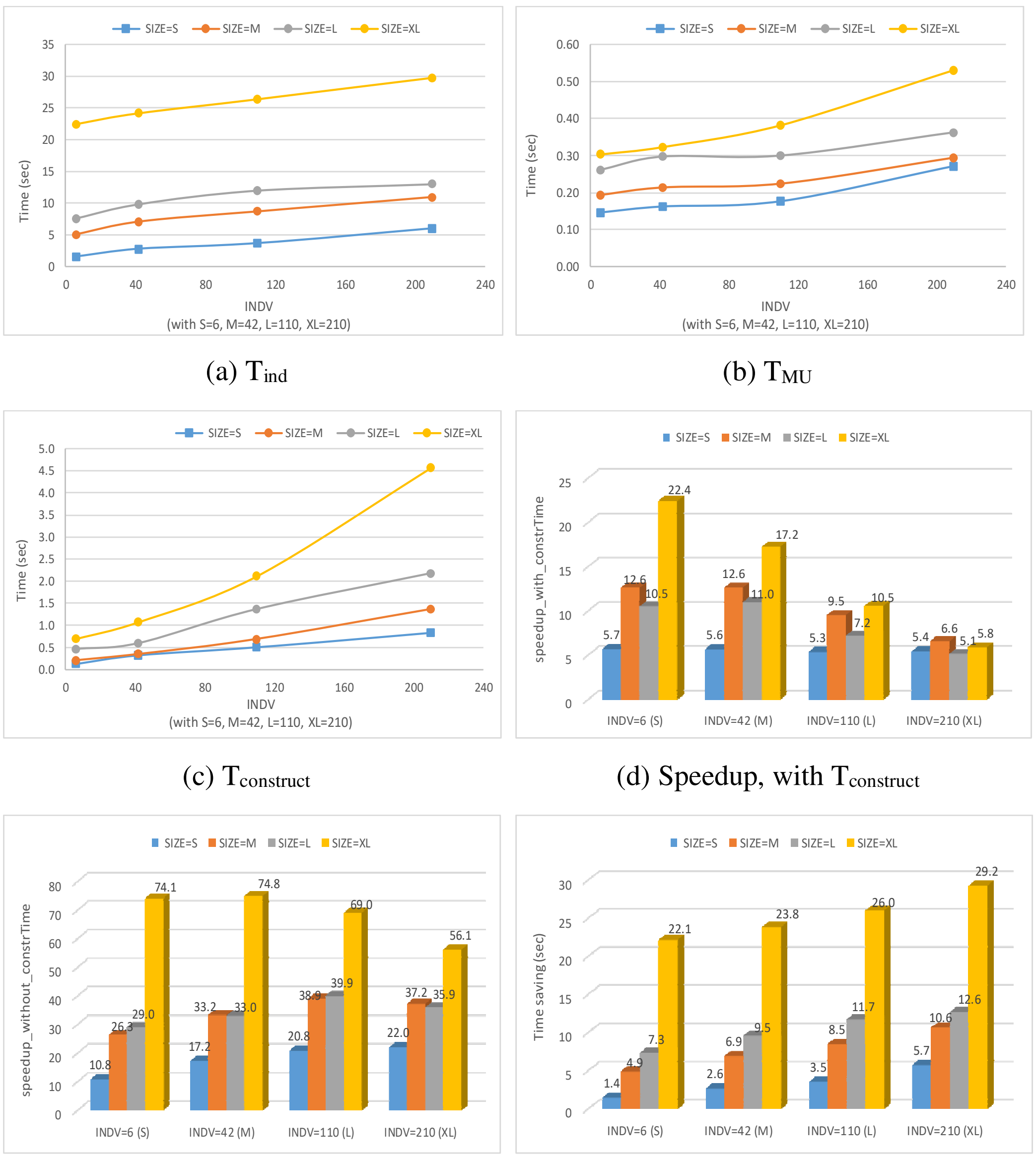
7.2. Results for Trend Analysis (RT2)

In this experiment, a trend analysis is conducted on an element named X-Goal from a set of individual GRL models and their union model M_U. The purpose of this analysis is to study the trend of this goal's *importance value* attribute and analyze how this value changes over time. Performing this analysis on M_U implies retrieving an element named X of type Goal, annotated with any version number <versions>, which could be a single version, a set of versions, or a range of versions that the element may belong to. With individual models, the search for and retrieval of X-Goal involve each individual model, where the laborious process in practice would involve opening each individual model, searching the desired element, observing its importance value, and closing the current model.

Figure 26a–c shows the respective T_{ind}, T_{MU}, and $T_{construct}$ utilised in this experiment, while Figure 26d–f illustrates the time speedups, with and without $T_{construct}$ and the time saved. The results in Figure 26a–c clearly illustrate a linear increase of T_{ind}, T_{MU}, and $T_{construct}$ as *SIZE* or *INDV* increases. This is to be expected, as the searching task, which is the core of trend analysis, has a linear time complexity.

From Figure 26d, it can be noticed that for one *SIZE* category, *SIZE* = XL, the speedup decreases with the increase of the number of individual models in a family, i.e., *INDV*. This decrease is mainly due to the consideration of $T_{construct}$ while computing the speedup. Nevertheless, the achieved speedups with $T_{construct}$ are very positive and important. Without considering $T_{construct}$ the speedup becomes more substantial, with a generally increasing pattern as *INDV* or *SIZE* increases, as depicted in Figure 26e.

Finally, Figure 26f demonstrates that the use of M_U reduces the time needed to search for elements that belong to a group of models compared to traversing each individual model separately. This is clearly illustrated by the time savings, which substantially increase when *SIZE* or *INDV* increases.



(a) T_{ind} (b) T_{MU}

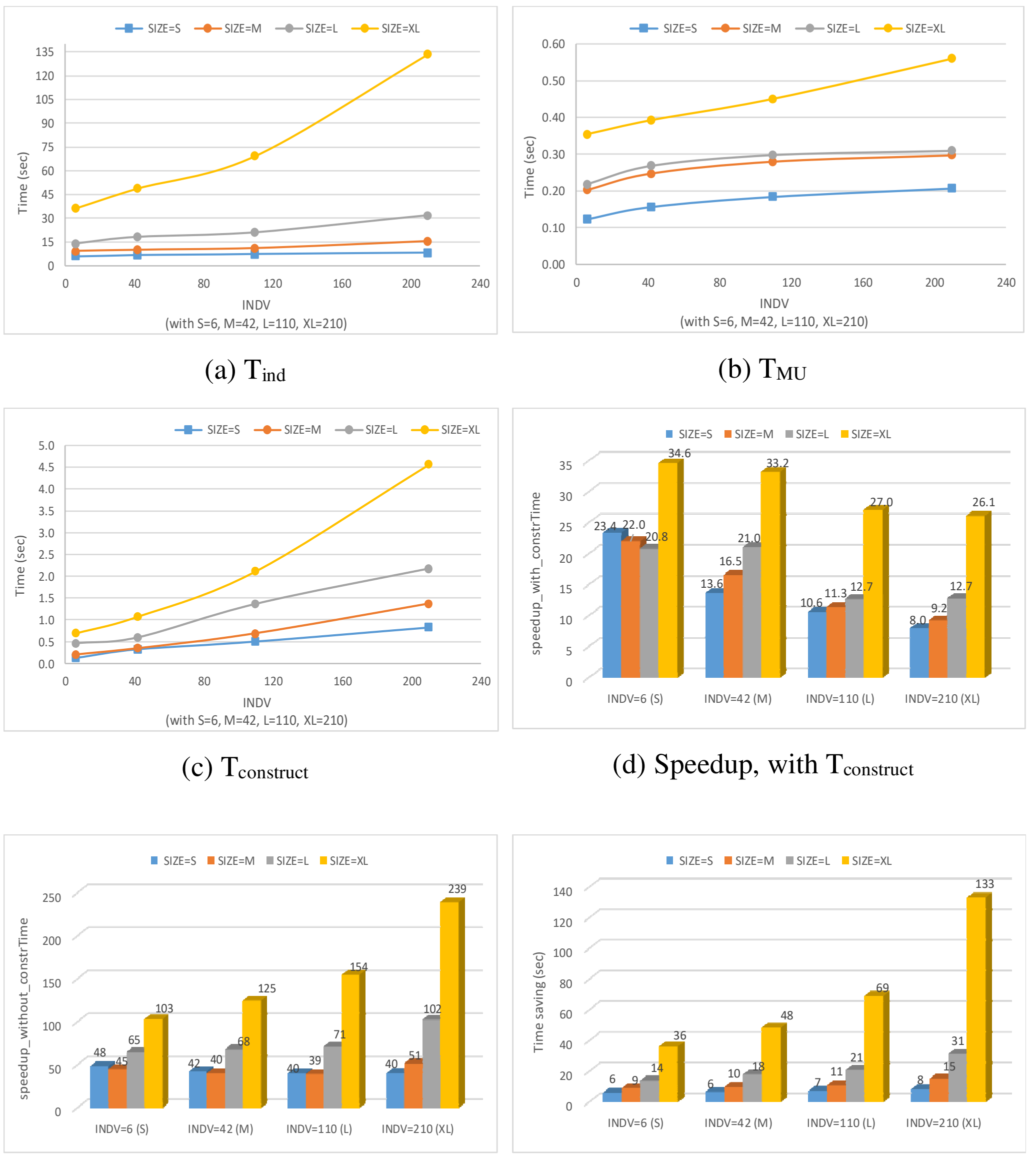
(c) $T_{construct}$ (d) Speedup, with $T_{construct}$

(e) Speedup without $T_{construct}$ (f) Time saving

Figure 26. Results for trend analysis (RT2) for all *INDV* and *SIZE* categories.

7.3. Results for Commonality Analysis (RT3)

Figure 27 shows the results of conducting commonality analysis on a set of GRL models and their M_U. This experiment required searching for all elements that are common between all model versions. This is a tedious task, especially when the number/size of models increases. Searching a set of M individual models with N elements each to find elements in common between all models has a complexity of $\mathcal{O}(M \times N^2)$. However, with M_U we only use one model to search for elements in common, where the task here is to search for elements annotated with <*ALL*>.



(a) T_{ind} (b) T_{MU}

(c) $T_{\text{construct}}$ (d) Speedup, with $T_{\text{construct}}$

(e) Speedup without $T_{\text{construct}}$ (f) Time saving

Figure 27. Results for commonality analysis (RT3) for all *INDV* and *SIZE* categories.

It can be noticed from Figure 27a that T_{ind} in this experiment is at least two times larger than T_{ind} in the previous experiment (i.e., RT2). T_{MU} and $T_{\text{construct}}$, on the other hand, grow at the same pace. Again, times T_{ind}, T_{MU}, and $T_{\text{construct}}$ increase as *SIZE* or *INDV* increases.

The time speedup achieved by using M_U is always positive, regardless of whether or not $T_{\text{construct}}$ is considered. Figure 27d illustrates that *speedup_with_constrTime* shows a pattern close to that in RT2, that is, for one *SIZE* category, e.g., *SIZE* = S or *SIZE* = M, the speedup decreases as *INDV* increases. On the other hand, *speedup_without_constrTime* is more substantial, and increases as *INDV* or *SIZE* increases, as shown in Figure 27e. Finally, the time savings in this experiment (Figure 27f) are more significant than in the experiments for RT2, as the potential gain here is quadratic rather than linear, with about 133 s saved for extra-large families of extra-large models.

7.4. Threats to Validity

One major threat to the validity of our empirical evaluation stems from relying on randomly generated inputs, both graphs and experimental parameters. This threat can be alleviated by using more realistic parameters, e.g., using real-world model families.

Another threat is related to the experimental parameters, where we used only *SIZE* and *INDV*. We recognize that we need to examine the impact of the variability of models on reasoning. For example, we could consider the number of different annotations per element to describe how similar or different the members are. The topology of the graphs (e.g., depth, number of linked nodes, etc.) could benefit from specific experiments. The complexity of a property to be checked might be another parameter to consider.

Our experiments need to be elaborated further for more complex properties and analysis types, some of which might not exploit the STAL annotations as the ones used here could, and should be compared to other approaches that handle variability in the time dimension alone for goal models, including the work of Aprajita et al. [28] and that of Grubb and Chechik [29]. Furthermore, the current validation covers two modeling language, goal models and state machines, and should be extended to other types that are more structural, e.g., class diagrams, or behavioral, e.g., process models. Finally, the usefulness of our approach needs to be assessed and demonstrated with more significant examples or real-world case studies and a better quantitative comparison with existing approaches where there are overlapping analysis functionalities.

7.5. Summary and Discussion

Thus far, this paper has explored a research question related to the performance of union models by empirically evaluating the efficiency of reasoning and analysis tasks for modeling families using union models in comparison to the use of individual models. We defined three general reasoning tasks and evaluated their performance, first using union models and then using individual models several times. We have discussed the experimental methodology, setup, and implementation and reported on the empirical results. Our experiments demonstrate the usefulness and performance gains of union models for analyzing a family of models all at once compared to individual models. Finally, we have identified several threats to validity as caveats.

The speedups, whether the time for constructing the union model is considered or not, are always in favor of the union model, and these speedups generally increase as models or families grow larger. However, there is erratic behavior regarding the increase of the time speedup. For instance, the speedup for L-sized models in certain experiments was larger than the speedup of M-sized models, while in other experiments it was smaller than that for M-sized models. This behavior is due in part to the possibility of having a footprint for the creation of M_U, which is not negligible. Moreover, in order for the behavior of the speedup to be more stable and less erratic, we may need to consider additional experimental parameters such as the ratio of variability across models and the topology of the models at hand.

In addition, it is worthwhile to mention here that in certain individual experiments related to the calculation of *speedup_with_constrTime* we encountered counter-intuitive results, specifically, where the speedup was less than one. In such experiments, a speedup value less than one suggests that the use of M_U, when taking its construction time into account, is slower than the use of individual models. Again, these slowdowns where obtained in individual experiments only, and they were amortized by calculating the average results of fifteen runs.

In terms of concrete time saved, which ranges from half a second to a few minutes depending on the experiment, the savings may not seem large at first glance; however, there are several practical implications:

- The time savings accumulate as many analyses are performed on a same union model, especially as the union model construction time is amortized over multiple analyses;

- The savings doe not take into account the concrete time required when a practitioner uses a tool to open a model, perform the analysis, and save the results, which this may take many seconds per model and is prone to human error;
- The savings become essential in a context where union model construction and verification are offered *as a service*, i.e., as an online application where multiple users can concurrently upload, merge, and analyze their model families.

While this paper focuses on analysis types that are language *independent*, the first author's previous work in [10,30] explored the adaptation or *lifting* of language-*dependent* analyses for a given language, namely, forward and backward propagation of satisfaction values for GRL model families, again with positive performance gains. The GRL-based analysis algorithms and their lifted version were implemented using an optimizer (IBM CPLEX) with speedups up to 23 times faster when using the union model compared to individual models, and for a much lower additional memory cost. In practical terms, this means that analysis on model families is no longer limited to the verification of behavioral properties often seen in existing approaches.

In addition, while conducting property checking our focus was not on the type of properties to be checked (i.e., semantic vs. syntactic properties); rather, the focus was on examining the improvement of performance in terms of time speedup for performing any kind of property checking (syntactic in this paper) when using union models once as compared to performing the same task using many models in the model family one model at a time. In the future, it would be possible to differentiate between the types of properties to be checked and assess the impact of the property type on the overall performance of the analysis.

8. Related Work

In the literature, few approaches have been proposed to support model families. Shamsaei et al. [9] used GRL to define a generic goal model family for various types of organizations in the legal compliance domain. They annotated models with information about organization types to specify which were applicable to which family members. Different from our work, the work of [9] handled only variation of models in the space dimension, and did not consider evolution over time. In addition, the authors focused only on maintainability issues and did not propose union models to improve analysis complexity and reduce analysis effort. Palmieri et al. [8] elaborated further on the work of [9] to support more variable regulations. The authors integrated GRL and feature models to handle regulatory goal model families as software product lines (SPLs) by annotating a goal model with propositional formula related to features in a feature model. Unlike [9], Palmieri et al. considered further dimensions such as the organization size, type, the number of people, etc. However, they did not consider the evolution of goal models over time, and did not introduce union models.

Our work has strong conceptual resemblances with the domain of SPL engineering, which aims to manage software variants in order to efficiently handle families of software [31,32]. The notion of a feature is central to variability modeling in SPL [33], where features are expressed as variability points. Feature models (FMs) [16] are a formalism commonly used to model variability in terms of optional, mandatory, and exclusive features organized in a rooted hierarchy and associated with constraints over features. FMs can be encoded as propositional formula defined over a set of Boolean variables, with each variable corresponding to a feature. FMs are deemed to be very useful to represent feature dependencies, describe precisely allowed variability between products in a product line, and guide feature selection to allow for the construction of specific products [34,35]. However, it is important to emphasize here that FM is different from our proposed union models M_U in both usage and formalism. The differences between both artifacts can be summarized as follows:

- A feature model represents variability at an abstract "feature level", which is separate from software artifacts (such a grammar of possible configurations), whereas M_U represents the variability of all existing models at the "artifact level" itself [36];
- While a feature model defines all possible valid configurations of products along with constraints on their possible configurations, an M_U provides a complete view of the solution space that makes it explicit for modelers which particular element belongs to which model without necessarily modeling dependencies or constraints between elements of one model or across several models of a family;
- The purpose behind using both artifacts is different, in that FMs are mainly used to ensure that the derived individual models are valid through valid feature configurations, with the possibility of generating new models or products. On the other hand, M_U is proposed to perform domain-specific analysis beyond configuration validation more efficiently on a group of existing models than on individual models;
- Finally, an M_U enables the extraction of individual members of a model family by means of selecting particular time and/or space annotations, while FM enables model extraction by means of selecting valid combination of dependency rules and cross-tree constraints between features.

There exist verification approaches that target many valid configurations of feature models [37]. For instance, Classen et al. [38] explored the use of model checking on a family of behavioral models captured by a feature model, with resulting gains in verification time. Similar approaches have been developed for real-time SPLs [39], for symbolic model checking [40], and for probabilistic model checking [41], among others [42]. These approaches are, however, limited to the space dimension (no evolution of models over time) and to behavioral properties, e.g., they cannot be used to reason about satisfaction propagation in a family of goal models, which is allowed by the usage of M_U [30].

To express variability, annotative approaches are commonly used in the literature, such as in the work of Czarnecki and Antkiewicz [34], in which variability points are represented as presence conditions. These conditions are propositional expressions over features. Annotations of features can be used as inputs to a variability realization mechanism in order to derive or create a concrete software system as variant of the SPL. Using a negative variability mechanism, annotative approaches define a so-called 150% model that superimposes all possible variations for the entire SPL. The 150% model is used to derive a particular variant, while other irrelevant parts are removed. While union models have similarities to 150% models, the uses of both models, the domains they are used in, and the ways of annotating them are all different.

Ananieva et al. [43–45] proposed an approach for consistent view-based management of variability in space and time. In particular, the authors studied and identified concepts and operations of approaches and tools dealing with variability in space and time. Furthermore, the authors identified consistency preservation challenges related to view-based evolution of variable systems composed of heterogeneous artifacts and provided a technique for (semi-)automated detection and repair of variability-related inconsistencies.

Mahmood et al. [46] presented an empirical assessment of annotative and compositional variability mechanisms for three popular types of models, namely, class diagrams, state machine diagrams, and activity diagrams. The authors provided recommendations to language and tools developers and discussed findings from a family of three experiments with 164 participants in total, in which they studied the impact of different variability mechanisms during model comprehension tasks. The authors recommended that annotative techniques lead to better developer performance and noted that the use of the compositional techniques correlates with impaired performance. In addition, for all the experiments it was found that annotative variability is preferred over compositional variability by a majority of the participants for all task types and in all model types.

The approaches proposed by Seidl et al. [14], Ananieva et al. [47], Michelon et al. [48,49], and Lity et al. [15] are closely related to ours. In the context of SPL engineering, they considered variation of software families in both space and time, and explicitly annotated variability models with time and space information to distinguish between the different versions and variations of software artifacts.

Different from approaches for managing the variability and evolution, Wittler et al. [50] have introduced a variability model for both software and hardware that captures variability in both space and time as well as the dependencies between loosely coupled software and hardware components. The authors refined the Unified Conceptual Model by introducing system generations to reflect the implicit dependencies between software and hardware components in product lines.

Even though the concept of a "family" is shared between our work and the SPL domain, the main difference is one of purpose and scope. At its core, SPL engineering is a software engineering methodology for systematic *proactive reusability*. The overarching concern is to strategically design, plan, and maintain a set of software artifacts according to points of functionality (features). Then, by exploiting commonalities between variants, engineers can efficiently derive or create software products with desirable features. It is not the goal of our work to plan for reusability or to derive new models or products. Characteristically, union models do not depend on modeling common functionalities in a feature model. Such a model is expressly created to facilitate proactive reusability, and is a central concept in SPLs.

Instead, we use union models to analyze families of *existing* models *irrespective* of their provenance or the intent for which they were constructed. As an object-oriented technology, union models generalize SPLs by decoupling family modeling from the particularities and exigencies of proactive reusability. Union models are a step towards a more basic and fundamental idea: the representation and analysis of families of models.

Famelis et al. [6] proposed partial models to capture a set of possible alternative models with design-time uncertainty. The emphasis of this work was to create a methodology for the lifecycle of design-time uncertainty. This includes articulating modelers' uncertainty about design decisions, maintaining this uncertainty, and supporting systematic decision-making via refinement [51]. Dhaouadi et al. [52] addressed the challenges of design-time uncertainty by proposing *DRUIDE* (Design and Requirements Uncertainty Integrated Development Environment), a language and workflow for articulating design time uncertainty. DRUIDE provides modelers with the ability to explicitly articulate uncertainty and compose completely heterogeneous models by linking uncertainties. This work is related to decision-oriented Product Line Engineering (PLE) [53], mainly as design uncertainty often involves modeling sets of design alternatives. Furthermore, DRUIDE is related to research on creating representations of sets of related models, which is in turn related to our work on model families. Compared to the work of Famelis et al. [6] and Dhaouadi et al. [52], the focus of our work is more fundamental, looking at efficient reasoning about model families by leveraging redundancies across members of the family. Naturally, the three approaches are complementary, and we are actively researching various synergies.

The use of a single model to represent multiple interrelated systems was proposed by Stünkel et al. [54]. Similar to our approach, they used typed graphs to represent commonalities between models. The authors focused on interoperability, consistency specification, and consistency restoration, and did not consider the representation of modeling families or variability representation in space and time.

Aprajita et al. [28,55] extended the GRL metamodel to document explicit changes (additions/deletions) of elements to specific versions of a model. Although a model family can be captured, this approach is specific to one language and is currently incomplete in terms of the kinds of changes to versions that it can accommodate.

Grubb et al. [56,57] introduced the concept of *dynamic intentions* into goal models to capture alternatives on multiple time scales. The authors proposed a tool-supported method for specifying changes in intention over time which uses simulations to ask a variety of "what-if" questions about models that evolve over time.

Hablutzel et al. [58] investigated the problem of model merging for Tropos goal models. In particular, the authors proposed a formal approach to address the problem of automatically merging the attributes of intentions and actors in which both static models and models with timing information were considered. In their work, Hablutzel et al. looked at the initial creation of a goal model through merging, rather than tracking model evolution over space and time through a merged representation using union models.

9. Conclusions and Future Work

In this paper, we propose union models as a first-class generic artifact to capture and represent model families. The paper provides a graph theory-based formalization of model families and their union models. In particular, model families are formalized as a set of attributed typed graphs in which all models are typed over the same metamodel. Metamodels are formalized as attributed type graphs, whereas union models are formalized as the union of all graph elements in the set of typed attributed graphs which constitute a model family. Included in the union model are the graph elements annotated by the models they are occurring. The purpose of this formalization is to permit representation of model families and their union models *independent* of any modeling language used in the context of MBSE. In addition, we propose a Spatio-Temporal Annotation Language (STAL) to support the representation of variability in model families in both the space and time dimensions and to facilitate reasoning about union models.

We demonstrate how the properties of these formalisms can be used to support the construction of union models as well as to perform typical reasoning tasks, such as trend analysis and property checking, on union models. We empirically illustrate the potential of using union models to improve analysis and reasoning over a set of models all at once as opposed to analyzing single models separately one at a time. Positive results are observed even when the time taken to construct union models is taken into consideration; in practice, the construction time can be amortized over many analyses.

The artifacts related to this paper are available online. These include Excel files for attributed typed graphs defined according to the formalization provided in Section 3. In addition, GRL model families generated for four SIZE categories (i.e., small, medium, large, X-large) for our experiments, and the Python programs used for the union algorithm, for generating models, and for measuring the reasoning tasks defined in Section 6 are included. See https://bit.ly/UnionModelsExtra, accessed on 8 February 2023.

For future work, there are many opportunities to follow up on several directions:

- Studying the effects of variation and topology on reasoning techniques: it is necessary to describe how sensitive reasoning and analysis are to the degree of variation in a union model, as well as where this variation is located (topology). The degree of variation of an M_U can be inferred from the number of annotations and the number of annotated elements that exist in a model family and are represented by that M_U.
- Improved tool support: although a prototype tool exists for creating union models, the software ecosystem could be greatly enhanced by providing conversion from metamodels (in EMF or MOF) to type graphs and from models to typed graphs expressed with PELA. Tools to better visualize annotated union models in the original language's syntax whenever possible would be very useful. Support for more rigorous lifting analysis methods and editors would contribute to the practical adoption of the proposed approach.
- Studying the usability of the approach: there is a need to assess which practitioners this approach is useful to and which parts require usability enhancements in order to improve adoption in industry, especially regarding automation. We envision offering merging and several types of reasoning as online services.
- Exploration of further synergies with techniques for representing and managing design-time uncertainty using partial models [51]. Specifically, we would like to adapt model transformation lifting to union models and to develop operators for managing union models across their lifecycle.

Author Contributions: Conceptualization: S.A., D.A. and Y.L.; methodology: S.A. and D.A.; software: S.A.; validation: S.A.; formal analysis: S.A., D.A. and Y.L.; investigation: S.A., D.A. and Y.L.; resources: S.A. and D.A.; data curation: S.A.; writing—original draft preparation: S.A.; writing—review and editing: D.A. and Y.L.; visualization: S.A., D.A. and Y.L.; supervision: D.A.; project administration: D.A.; funding acquisition: S.A. and D.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Ontario Trillium Scholarship program and the BMO Financial Group Graduate Bursaries (for S. Alwidian), as well as the NSERC Discovery program (for D. Amyot).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors thank Michalis Famelis, Nan Niu, and Dorina Petriu for their useful feedback on this work.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACS	Access Control System
ATG	Attributed Type Graph
DSML	Domain-Specific Modeling Language
FM	Feature Model
FOL	First-Order Logic
GRL	Goal-oriented Requirement Language
$INDV$	Number of individual models in a family
ITG	Instance Typed Graph
MBSE	Model-based Software Engineering
MF	Model Family
M_U	Union Model
OCL	Object Constraint Language
PELA	Propositional Encoding Language with Annotations
PLE	Product Line Engineering
RT	Reasoning Task
$SIZE$	Size of individual models
SPL	Software Product Line
STAL	Spatio-Temporal Annotation Language
TG	Type Graph
UML	Unified Modeling Language
URN	User Requirements Notation

Appendix A. STAL Grammar Definition

This appendix provides the grammar of the Spatio-Temporal Annotation Language (STAL) discussed in Section 4.

Appendix A.1. Metagrammar

The following elements describe the metagrammar and styles used to define the STAL grammar.

- <> for rules
- ::= for definitions
- { }∗ for 0 to many
- { }+ for 1 to many
- | for alternatives

- **bold** for terminal symbols
- NATURAL for non-negative integers
- IDENTIFIER for Strings without spaces
- # *text* for comments

Appendix A.2. STAL Grammar

```
# ALL here means all versions and configurations.
<STAL> ::= <annotation> { ; <annotation>}∗ | ALL
<annotation> ::= < <versions> , <configurations> >
# ALL here means all versions.
<versions> ::= <singleversion> | <listversions> | <rangeversions> | ALL
# In versions, ver1, ver2, ver3, . . . are sorted.
<singleversion> ::= verNATURAL
# Nested lists, if any, are flattened.
<listversions> ::= ( <versions> { , <versions>}+ )
# The first version value must be lower than the second version value.
<rangeversions> :: = [ <singleversion> : <singleversion> ]
# ALL here means all configurations.
<configurations> ::= <singleconfig> | <listconfigs> | ALL
<singleconfig> ::= IDENTIFIER
<listconfigs> ::= ( <singleconfig> { , <singleconfig>}+ )
```

Note that certain modelers use a hierarchical version numbering scheme (versions 2.3.1 and 4.3.2, etc.); for simplification, these hierarchical numbers can be mapped to simple integers while keeping the same ordering.

References

1. Stahl, T.; Voelter, M.; Czarnecki, K. *Model-Driven Software Development: Technology, Engineering, Management*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2006.
2. van Deursen, A.; Klint, P.; Visser, J. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.* **2000**, *35*, 26–36. [CrossRef]
3. Taentzer, G.; Mantz, F.; Arendt, T.; Lamo, Y. Customizable Model Migration Schemes for Meta-model Evolutions with Multiplicity Changes. In Proceedings of the Model-Driven Engineering Languages and Systems, Miami, FL, USA, 29 September–4 October 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 254–270. [CrossRef]
4. Henderson, K.; Salado, A. Value and benefits of model-based systems engineering (MBSE): Evidence from the literature. *Syst. Eng.* **2021**, *24*, 51–66. [CrossRef]
5. Domingo, Á.; Echeverría, J.; Pastor, Ó.; Cetina, C. Evaluating the Benefits of Model-Driven Development. In *Advanced Information Systems Engineering*; Dustdar, S., Yu, E., Salinesi, C., Rieu, D., Pant, V., Eds.; Springer: Berlin/Heidelberg, Germany, 2020; pp. 353–367. [CrossRef]
6. Famelis, M.; Salay, R.; Chechik, M. Partial Models: Towards Modeling and Reasoning with Uncertainty. In Proceedings of the 34th International Conference on Software Engineering, ICSE'12, Zurich, Switzerland, 2–9 June 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 573–583. [CrossRef]
7. Michelon, G.K.; Obermann, D.; Assunção, W.K.G.; Linsbauer, L.; Grünbacher, P.; Egyed, A. Managing Systems Evolving in Space and Time: Four Challenges for Maintenance, Evolution and Composition of Variants. In Proceedings of the 25th ACM International Systems and Software Product Line Conference (SPLC '21)—Volume A, Leicester, UK, 6–11 September 2021; ACM: New York, NY, USA, 2021; pp. 75–80. [CrossRef]
8. Palmieri, A.; Collet, P.; Amyot, D. Handling Regulatory Goal Model Families as Software Product Lines. In Proceedings of the Advanced Information Systems Engineering, Stockholm, Sweden, 8–12 June 2015; Springer International Publishing: Cham, Switzerland, 2015; pp. 181–196. [CrossRef]
9. Shamsaei, A.; Amyot, D.; Pourshahid, A.; Braun, E.; Yu, E.; Mussbacher, G.; Tawhid, R.; Cartwright, N. An Approach to Specify and Analyze Goal Model Families. In *System Analysis and Modeling: Theory and Practice, Innsbruck, Austria*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 34–52. [CrossRef]
10. Alwidian, S. Union Models: Support of Variability Modeling and Efficient Reasoning About Model Families over Space and Time. Ph.D. Thesis, University of Ottawa, Ottawa, ON, Canada, 2020. [CrossRef]

11. Alwidian, S.; Amyot, D. Inferring Metamodel Relaxations Based on Structural Patterns to Support Model Families. In Proceedings of the MODELS Companion 2019 (ME 2019), Munich, Germany, 15–20 September 2019; IEEE CS: Piscataway, NJ, USA, 2019; pp. 294–303. [CrossRef]
12. Alwidian, S.; Amyot, D. Union Models: Support for Efficient Reasoning About Model Families Over Space and Time. In Proceedings of the System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0, Munich, Germany, 16–17 September 2019; Springer International Publishing: Cham, Switzerland, 2019; pp. 200–218. [CrossRef]
13. International Telecommunication Union. Recommendation Z.151 (10/18) User Requirements Notation (URN)—Language Definition. 2018. Available online: https://www.itu.int/rec/T-REC-Z.151/en/ (accessed on 17 December 2022).
14. Seidl, C.; Schaefer, I.; Aßmann, U. Integrated Management of Variability in Space and Time in Software Families. In Proceedings of the 18th International Software Product Line Conference (SPLC '14)—Volume 1, Florence, Italy, 15–19 September 2014; ACM: New York, NY, USA, 2014; pp. 22–31. [CrossRef]
15. Lity, S.; Nahrendorf, S.; Thüm, T.; Seidl, C.; Schaefer, I. 175% Modeling for Product-Line Evolution of Domain Artifacts. In Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2018), Madrid, Spain, 7–9 February 2018; ACM: New York, NY, USA, 2018; pp. 27–34. [CrossRef]
16. Famelis, M.; Rubin, J.; Czarnecki, K.; Salay, R.; Chechik, M. Software Product Lines with Design Choices: Reasoning about Variability and Design Uncertainty. In Proceedings of the 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), Austin, TX, USA, 17–22 September 2017; IEEE CS: Piscataway, NJ, USA, 2017; pp. 93–100. [CrossRef]
17. Ehrig, H.; Ehrig, K.; Prange, U.; Taentzer, G. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*; Springer: Berlin/Heidelberg, Germany, 2006. [CrossRef]
18. Taentzer, G.; Rensink, A. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In Proceedings of the Fundamental Approaches to Software Engineering, Edinburgh, UK, 4–8 April 2005; Springer: Berlin/Heidelberg, Germany, 2005; pp. 64–79. [CrossRef]
19. Biermann, E.; Ermel, C.; Taentzer, G. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Softw. Syst. Model.* **2012**, *11*, 227–250. [CrossRef]
20. Ehrig, K.; Küster, J.M.; Taentzer, G.; Winkelmann, J. Generating Instance Models from Meta Models. In *Formal Methods for Open Object-Based Distributed Systems*; Gorrieri, R., Wehrheim, H., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; pp. 156–170. [CrossRef]
21. de Lara, J.; Bardohl, R.; Ehrig, H.; Ehrig, K.; Prange, U.; Taentzer, G. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.* **2007**, *376*, 139–163. Fundamental Aspects of Software Engineering. [CrossRef]
22. Ehrig, H.; Prange, U.; Taentzer, G. Fundamental Theory for Typed Attributed Graph Transformation. In *Graph Transformations*; Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 161–177. [CrossRef]
23. Van Der Straeten, R.; Mens, T.; Simmonds, J.; Jonckers, V. Using Description Logic to Maintain Consistency between UML Models. In *UML 2003—The Unified Modeling Language. Modeling Languages and Applications*; Stevens, P., Whittle, J., Booch, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; pp. 326–340. [CrossRef]
24. Heradio-Gil, R.; Fernandez-Amoros, D.; Cerrada, J.A.; Cerrada, C. Supporting commonality-based analysis of software product lines. *IET Softw.* **2011**, *5*, 496–509. [CrossRef]
25. Famelis, M. Managing Design-Time Uncertainty in Software Models. Ph.D. Thesis, University of Toronto, Toronto, ON, Canada, 2016. Available online: http://hdl.handle.net/1807/72997 (accessed on 8 February 2023).
26. NetworkX. Network Analysis in Python. 2022. Available online: https://networkx.org/ (accessed on 17 December 2022).
27. Tamás László, F. SATisPY Solver. 2018. Available online: https://github.com/netom/satispy (accessed on 28 March 2020).
28. Aprajita; Luthra, S.; Mussbacher, G. Specifying Evolving Requirements Models with TimedURN. In Proceedings of the IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE), Buenos Aires, Argentina, 21–22 May 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 26–32. [CrossRef]
29. Grubb, A.M.; Chechik, M. Formal Reasoning for Analyzing Goal Models That Evolve over Time. *Requir. Eng.* **2021**, *26*, 423–457. [CrossRef]
30. Alwidian, S.; Amyot, D. "Union is Power": Analyzing Families of Goal Models Using Union Models. In Proceedings of the MODELS'20: 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '20), Montreal, QC, Canada, 18–23 October 2020; ACM: New York, NY, USA, 2020; pp. 252–262. [CrossRef]
31. Pohl, K.; Böckle, G.; Linden, F. *Software Product Line Engineering: Foundations, Principles, and Techniques*; Springer: Berlin/Heidelberg, Germany, 2005. [CrossRef]
32. Chimalakonda, S.; Hyung Lee, D. A family of standards for software and systems product lines. *Comput. Stand. Interfaces* **2021**, *78*, 103537. [CrossRef]
33. Berger, T.; Steghöfer, J.P.; Ziadi, T.; Robin, J.; Martinez, J. The State of Adoption and the Challenges of Systematic Variability Management in Industry. *Empir. Softw. Engg.* **2020**, *25*, 1755–1797. [CrossRef]
34. Czarnecki, K.; Antkiewicz, M. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Proceedings of the Generative Programming and Component Engineering, Tallinn, Estonia, 29 September–1 October 2005; Springer: Berlin/Heidelberg, Germany, 2005; pp. 422–437. [CrossRef]

35. Beuche, D.; Papajewski, H.; Schröder-Preikschat, W. Variability management with feature models. *Sci. Comput. Program.* **2004**, *53*, 333–352. [CrossRef]
36. Schobbens, P.Y.; Heymans, P.; Trigaux, J.C.; Bontemps, Y. Generic semantics of feature diagrams. *Comput. Netw.* **2007**, *51*, 456–479. [CrossRef]
37. Thüm, T.; Apel, S.; Kästner, C.; Schaefer, I.; Saake, G. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* **2014**, *47*, 6. [CrossRef]
38. Classen, A.; Heymans, P.; Schobbens, P.Y.; Legay, A.; Raskin, J.F. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)—Volume 1, Cape Town, South Africa, 2–8 May 2010; ACM: New York, NY, USA, 2010; pp. 335–344. [CrossRef]
39. Cordy, M.; Schobbens, P.Y.; Heymans, P.; Legay, A. Behavioural Modelling and Verification of Real-Time Software Product Lines. In Proceedings of the 16th International Software Product Line Conference (SPLC '12)—Volume 1, Salvador, Brazil, 2–7 September 2012; ACM: New York, NY, USA, 2012; pp. 66–75. [CrossRef]
40. Classen, A.; Cordy, M.; Heymans, P.; Legay, A.; Schobbens, P.Y. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.* **2014**, *80*, 416–439. [CrossRef]
41. Chrszon, P.; Dubslaff, C.; Klüppelholz, S.; Baier, C. ProFeat: Feature-oriented engineering for family-based probabilistic model checking. *Form. Asp. Comput.* **2018**, *30*, 45–75. [CrossRef]
42. Cordy, M.; Devroey, X.; Legay, A.; Perrouin, G.; Classen, A.; Heymans, P.; Schobbens, P.Y.; Raskin, J.F. A Decade of Featured Transition Systems. In *From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 285–312. [CrossRef]
43. Ananieva, S. Consistent Management of Variability in Space and Time. In Proceedings of the 25th ACM International Systems and Software Product Line Conference (SPLC '21)—Volume B, Leicester, UK, 6–11 September 2021; ACM: New York, NY, USA, 2021; pp. 7–12. [CrossRef]
44. Ananieva, S.; Kühn, T.; Reussner, R. Preserving Consistency of Interrelated Models during View-Based Evolution of Variable Systems. In Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2022), Auckland, New Zealand, 6–7 December 2022; ACM: New York, NY, USA, 2022; pp. 148–163. [CrossRef]
45. Ananieva, S.; Greiner, S.; Krueger, J.; Linsbauer, L.; Gruener, S.; Kehrer, T.; Kuehn, T.; Seidl, C.; Reussner, R. Unified Operations for Variability in Space and Time. In Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '22), Florence, Italy, 24–25 February 2022; ACM: New York, NY, USA, 2022. [CrossRef]
46. Mahmood, W.; Strüber, D.; Anjorin, A.; Berger, T. Effects of Variability in Models: A Family of Experiments. *Empir. Softw. Engg.* **2022**, *27*, 72. [CrossRef]
47. Ananieva, S.; Greiner, S.; Kühn, T.; Krüger, J.; Linsbauer, L.; Grüner, S.; Kehrer, T.; Klare, H.; Koziolek, A.; Lönn, H.; et al. A Conceptual Model for Unifying Variability in Space and Time. In Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A (SPLC '20), Montreal, QC, Canada, 19–23 October 2020; ACM: New York, NY, USA, 2020. [CrossRef]
48. Michelon, G.K.; Assunção, W.K.G.; Obermann, D.; Linsbauer, L.; Grünbacher, P.; Egyed, A. The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time. In Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2021), Chicago, IL, USA, 17–18 October 2021; ACM: New York, NY, USA, 2021; pp. 2–15. [CrossRef]
49. Michelon, G.K.; Obermann, D.; Assunção, W.K.G.; Linsbauer, L.; Grünbacher, P.; Fischer, S.; Lopez-Herrejon, R.E.; Egyed, A. Evolving software system families in space and time with feature revisions. *Empir. Softw. Eng.* **2022**, *27*, 112. [CrossRef]
50. Wittler, J.W.; Kühn, T.; Reussner, R. Towards an Integrated Approach for Managing the Variability and Evolution of Both Software and Hardware Components. In Proceedings of the 26th ACM International Systems and Software Product Line Conference—Volume B (SPLC '22), Graz, Austria, 12–16 September 2022; ACM: New York, NY, USA, 2022; pp. 94–98. [CrossRef]
51. Famelis, M.; Chechik, M. Managing design-time uncertainty. *Softw. Syst. Model.* **2019**, *18*, 1249–1284. [CrossRef]
52. Dhaouadi, M.; Spencer, K.; Varnum, M.H.; Grubb, A.M.; Famelis, M. Towards a generic method for articulating design uncertainty. *J. Object Technol.* **2021**, *20*, 3. [CrossRef]
53. Dhungana, D.; Grünbacher, P.; Rabiser, R. Domain-specific adaptations of product line variability modeling. In Proceedings of the Situational Method Engineering: Fundamentals and Experiences, Geneva, Switzerland, 12–14 September 2007; Springer: Boston, MA, USA, 2007; pp. 238–251. [CrossRef]
54. Stünkel, P.; König, H.; Lamo, Y.; Rutle, A. Multimodel correspondence through inter-model constraints. In Proceedings of the Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, 9–12 April 2018; ACM: New York, NY, USA, 2018; pp. 9–17. [CrossRef]
55. Aprajita; Mussbacher, G. TimedGRL: Specifying Goal Models over Time. In Proceedings of the 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW), Beijing, China, 12–16 September 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 125–134. [CrossRef]
56. Grubb, A.M.; Chechik, M. Looking into the Crystal Ball: Requirements Evolution over Time. In Proceedings of the 2016 IEEE 24th International Requirements Engineering Conference (RE), Beijing, China, 12–16 September 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 86–95. [CrossRef]

57. Grubb, A.M.; Chechik, M. Reconstructing the past: The case of the Spadina Expressway. *Requir. Eng.* **2020**, *25*, 253–272. [CrossRef]
58. Hablutzel, K.R.; Jain, A.; Grubb, A.M. A Divide & Concur Approach to Collaborative Goal Modeling with Merge in Early-RE. In Proceedings of the 2022 IEEE 30th International Requirements Engineering Conference (RE), Melbourne, Australia, 15–19 August 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 14–25.