Union Models: Support for Efficient Reasoning About Model Families Over Space and Time

Sanaa Alwidian and Daniel Amyot(✉)

School of EECS, University of Ottawa, Ottawa, Canada
{salwidia, damyot}@uottawa.ca

Abstract. For a given modeling language, a model family is a set of related models, with commonalities and variabilities among family members, that results from the variation/evolution of models over the space and time dimensions. With large model families, the analysis of individual models becomes cumbersome and inefficient. This paper proposes *union models* as a paradigm supporting the representation of model families (for time and space dimensions) using one generic model. Elements of a union model are annotated with information about time and space using a new *spatio-temporal annotation language* (STAL) in order to distinguish which element belongs to which model. We demonstrate empirically the usefulness of union models for analyzing a family of models, *all at once*, compared to individual models, *one model at a time*. Our experiments suggest that the use of union models facilitate efficient analysis in several contexts.

Keywords: GRL · Model analysis · Model evolution · Model family · Property checking · Union model

1 Introduction

In Model-Based Engineering (MBE), models are first-class artifacts used to represent and abstract knowledge and activities that govern a particular domain [1]. Models often undergo continuous change due to, for example, modifications in requirements or standards, or enhanced understanding of the domain to be modeled. Such change could happen over the course of *time* (i.e., evolution), resulting in one model evolving into a set of related versions. A model could also vary over the *space* dimension, where there could be several variants of the same model, all existing at the same time (e.g., to reflect different products or configurations). In both scenarios, a family of related models in the same language, where commonalities and variabilities between family members exist, is called a *model family*.

Change in an MBE context is inevitable. Hence, raising awareness to the phenomena of model families is of particular importance, especially in variant-rich domains such as cyber-physical systems, smart systems, or regulatory environments (where slightly different regulations need to be modeled for different regulated parties and jurisdictions). In any of these domains, models that are used to capture the domain's dynamic nature are subject to frequent variation and evolution. In other words, a modeler may start with an initial model version (v_0), which over *time* needs to

P. Fonseca i Casas et al. (Eds.): SAM 2019, LNCS 11753, pp. 200–218, 2019.
https://doi.org/10.1007/978-3-030-30690-8_12

be updated into a slightly different version (v_1) to reflect a changing requirement. This version may further evolve into versions v_2, v_3, and so on. In the space dimension, two or more modelers may need, *at the same time*, to create slightly different variations of an initial model to reflect different spaces called *configurations*. In such contexts, modelers often end up having a family of model versions and/or variations. *Analyzing and reasoning* about such models requires the modeler to load into a tool, analyze, and report on analysis results of each individual model, *separately*. This is a time-consuming and laborious process that becomes more critical as the number of models to analyze gets larger.

To alleviate these challenges, we propose to capture the set of individual models in a family using one generic model called *union model (M_U)*. M_U represents the union of all elements found in all family members, in both the space and time dimensions. The purpose behind the creation of M_U is to support efficient reasoning and analysis of a family of models, all at once, compared to analyzing individual models separately. At the core of M_U is the annotation of elements, which we realize by proposing a *spatio-temporal annotation language (STAL)*. The main purpose of STAL is to annotate elements of M_U with information about space and time, so as to distinguish which element belongs to which member model in the model family.

The rest of this paper is organized as follows: Sect. 2 discusses the motivation behind our work. Section 3 provides necessary background and formalisms that we rely on to formalize union models. In Sect. 4, we discuss how union models are formalized, constructed, and annotated. The potential benefits of using union models for reasoning and analysis are discussed in Sect. 5. Section 6 reports on experiments conducted to validate the potential efficiency of union models. Related work is discussed in Sect. 7. Finally, Sect. 8 concludes the paper and provides future directions.

2 Motivation

To explain and motivate our approach, we use a simple proof-of-concept example of a smart home environment, where we use the Goal-oriented Requirement Language (GRL) [2, 3] as a modeling language. In a smart home environment, stakeholders' goals, importance of goals, means to achieve goals and relationships between goals (e.g., contributions or decompositions) vary. This variation stems mainly from the existence of different configurations of a smart home that also evolve over time. In this example, we distinguish between two different configurations:

- Configuration A (*confA*): A smart home that is lived in by students who spend about 8 h of the day out of home.
- Configuration B (*confB*): A smart home that is lived in by retired senior persons (most likely sick), who spend most of their daytime and nighttime at home.

In these two configurations (see Fig. 1), a student's goals are slightly different from a senior's goals. For example, a student is more concerned about getting fresh air in her room by opening windows so as to reduce energy consumption (since a student's budget is usually tight). A retired senior, on the other hand, may focus more on getting her room's atmosphere refreshed using the most convenient option (regardless of cost),

e.g., by having the ventilator turned on most of the time. Also, the importance of achieving the "refresh air inside" goal differs between a senior person (100) and a student (80). Furthermore, to keep a senior's smart home secure, the home central operator could give the illusion that the house is lived in using several options, one of them being to keep lights always on. However, this may not be a feasible option for a student, since turning lights on all the time consumes energy beyond a student's budget affordability.

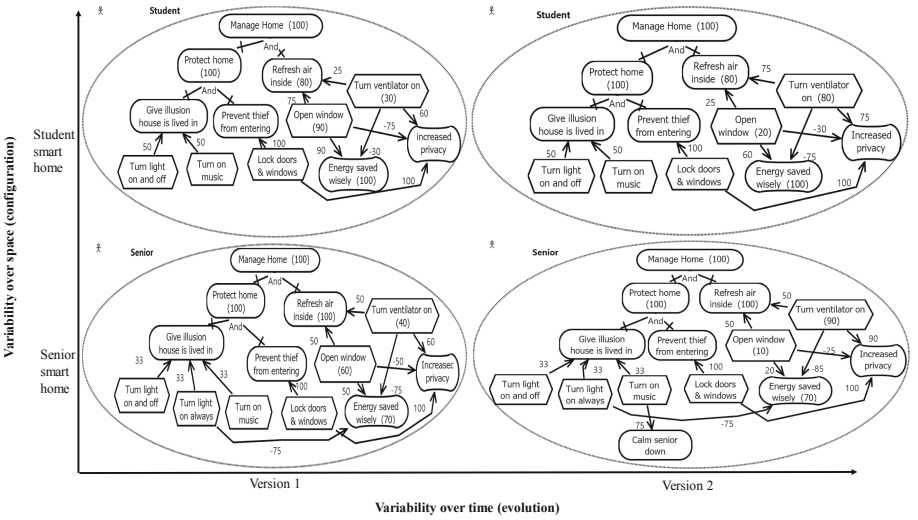

Fig. 1. Goal model family for smart home environments varying according to space and time

In addition to these "space-based" variations, goal models (in both configurations) could also evolve over time. In this example, we illustrate the evolution of models after several months (however, evolution could also happen over shorter periods of times). In such time-based evolution, goals and the means to achieve them (i.e., tasks) may differ between version 1 (produced in the summer) and version 2 (produced in the winter), due to changes in temperature, humidity, daylight duration, etc.

For a student's smart home (i.e., *confA*), the importance of goals/tasks and their impacts (i.e., contribution values) on other goals evolved from version 1 to version 2. For instance, the importance of task "Open window" in version 1 is 90 while it is 20 in version 2, as a student is able to open the window more often in the summer (assuming these models were produced in a Nordic country). Also, the impact of opening a window on the goal "energy saved wisely" is higher in the summer version (with contribution value = 90) than in the winter version (60). Finally, opening a window often in the summer has a higher negative impact on the "increased privacy" softgoal.

The previous evolutions are also applicable in the senior smart home environment (i.e., *confB*). As Fig. 1 shows, the "Open window" task is almost neglected at winter time (with importance = 10) compared to summer time (importance = 60). This is because a senior person is more vulnerable to get cold in the winter. Also, in version 2

(winter), the possibility for seniors to get depressed and anxious is higher (due to snow fall and short daylight duration). In this version, a smart home operator may calm the senior down by turning on soft music.

One important **challenge** implied by Fig. 1 is related to the complexity and effort required to analyze such family of models. Note that past versions may require analysis in case old versions of a product remain used by customers in the field. Assume a modeler plans to conduct satisfaction analysis (using the GRL forward propagation algorithm [2]) on each individual model, by assigning initial values to particular leaf goals, in order to study the impact of its satisfaction on the satisfaction of upper-level goals. She would end up running the same evaluation algorithm four times (in this example only), even though there are *many* common elements and computations among the four 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 in order of $M \times O(E)$. Such complexity becomes more significant if there are hundreds of models (or more), with hundreds of elements (or more) in each model. Moreover, the effort of loading a model into a tool, analyzing the model, saving analysis results, and then moving to the next model is *not negligible* in practice.

We aim to improve analysis complexity and reduce the effort of analyzing model families in arbitrary modeling languages (not only goal models), for both the space and time dimensions. This objective motivates us to find a way of representing model families other than using separate individual models. In this paper, we propose the use of a union model (M_U) as a single generic model that captures the entirety of a model family (in both dimensions of variability), in a comprehensive and exact way, such that all (and only) individual members of a family can be represented and analyzed.

3 Foundations

This section introduces relevant notations and background concepts related to graph-based modeling and propositional logic encodings.

3.1 Graph-Based Formalization of (Meta)Models

We formalize metamodels (resp. models) as type graphs (resp. typed graphs), as illustrated in Fig. 2.

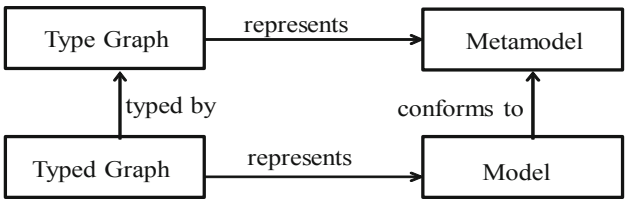

Fig. 2. Relationship between (meta)models and their graph representation

The following definitions, based on previous work by Ehrig et al. [4], are used as a basis for further formal definitions of model families and their union models.

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: $E_G \rightarrow N_G$ associate to each edge a source and a target node, respectively, such that e: x → y denotes an edge e with $src_G(e) = x$ and $tgt_G(e) = y$.

In graph theory (as in typed programming languages, where each element is assigned a 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 [4]. This is analogous to the relationship between models and metamodels in MDE [5], where each model needs to conform to a metamodel, as depicted in Fig. 2.

Definition 2–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.

Definition 3–Typed graph (model): A typed graph is a triple $G_{typed} = (G, type, TG)$ such that G is a graph (Def. 1) and type: G → TG is a graph morphism called the *typing morphism*. For example, the state machine model shown in Fig. 3 (right) is typed with the metamodel shown in Fig. 3 (left).

Fig. 3. Type graph (left) and typed graph (right)

The *vocabulary* (or scope) of a typed graph G_{typed} is a set $Voc = \{N_G \cup E_G\}$ of its typed nodes and edges [4]. For example, the vocabulary of the model in Fig. 3 (right) consists of nodes S1 and S2 of type *State*, node T1 of type *Transition*, and edges L1 and L2 of types *src* association (referred next as *srcAssoc*) and *tgt* association (referred next as *tgtAssoc*), respectively. We refer to the set of nodes and edges that are in the vocabulary of a model as the *elements* of that model.

3.2 Propositional Encoding of Models

In order to facilitate reasoning about models, and also to define a simple graph union, we represent typed graphs (i.e., models) as logical propositions. To encode a model in propositional logic, we first map elements in their vocabulary into propositional variables and then conjoin them. The mapping of graph elements into propositional variables is performed according to the following naming conventions:

- A node element $n \in N_G$ of type $t \in N_{TG}$ is mapped to a propositional variable *"n-t"*. Formally: *n-t iff* $\exists n \in N_G \wedge type(n) = t$

- An edge element $e \in E_G$ of type $t \in E_{TG}$ with source node x and target node y is mapped to a propositional variable *"e-x-y-t"*. Formally: *e-x-y-t iff* $\exists e \in E_G \wedge \text{type}(e) = t \wedge \text{src}_G(e) = x \wedge \text{tgt}_G(e) = y$.

For instance, the propositional encoding (*PE)* of the model (*m)* in Fig. 3 (right) is the conjunction of its propositional variables, described as follows:

```
PE(m) = S1-State ∧ T1-Transition ∧ S2-State ∧
        L1-S1-T1-srcAssoc ∧ L2-T1-S2-tgtAssoc
```

4 Union Models

The union model M_U of a model family (MF) is the union of all elements in all individual models of that family. The subsequent sections formally define union models (based on Defs. 1 to 3 and Sect. 3.2), and discuss how to construct an M_U and how to distinguish its elements by means of annotations using our annotation language (STAL).

4.1 Union Model Formalism

Definition 4–Union Model (M_U): Let MF be a model family with two models (i.e., typed graphs), such that MF = {G1, G2}, where G1 = ((N_{G1}, E_{G1}, src_{G1}, tgt_{G1}), $type_{G1}$, TG) and G2 = ((N_{G2}, E_{G2}, src_{G2}, tgt_{G2}), $type_{G2}$, TG). Their union model is a typed graph M_U = ((N_U, E_U, src_U, tgt_U), $type_U$, TG), such that: $N_U = N_{G1} \cup N_{G2}$, $E_U = E_{G1} \cup E_{G2}$, and the functions src_U, tgt_U, and $type_U$ are:

$$src_U(e) = \begin{cases} src_{G1}(e), \text{if } e \in E_{G1} \\ src_{G2}(e), \text{if } e \in E_{G2} \end{cases} \qquad tgt_U(e) = \begin{cases} tgt_{G1}(e), \text{if } e \in E_{G1} \\ tgt_{G2}(e), \text{if } e \in E_{G2} \end{cases}$$

$$type_U(elem) = \begin{cases} type_{G1}(elem), \text{if } elem \in V_{G1} \cup E_{G1} \\ type_{G2}(elem), \text{if } elem \in V_{G2} \cup E_{G2} \end{cases}$$

We can apply the above definition of graph union to sets of typed graphs of arbitrary sizes. Note however that even if the typed graphs used to construct union models are well-formed, there is no guarantee that their M_U is also a well-formed model. In fact, a union model M_U could respect the typing constraints imposed by the TG, but not the multiplicity constraints of attributes or association ends, or OCL constraints. We have already highlighted this general issue in [6], which is outside the scope of this paper.

4.2 Union of Propositional Encodings of Models

Given the propositional encoding of models discussed in Sect. 3.2, the union operation simply becomes the union of the propositional encodings of individual models.

Definition 5–Proposition Encoding Union (PE_U): Let MF = {G1, G2} be a model family, where G1 and G2 are typed graphs with the same metamodel TG, and PE(G1) and PE(G2) be their propositional encodings, such that they satisfy these conditions:

- *Cond. 1*: If two nodes have the same name and type, then these nodes are considered identical. We assume here that each node and each edge have 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 type and connect between the same source and target nodes, then edges are considered identical.

Then, the union of their propositional encoding becomes: $PE_U = PE(G1) \cup PE(G2)$.

Again, provided that the two conditions are satisfied, we can generalize the propositional encoding union (Def. 5) to a set of arbitrary encoded models.

4.3 Spatio-Temporal Annotation Language (STAL)

The challenging part of constructing a union model is not in the union operation itself (as expressed in Def. 5), but in being able to distinguish to which models a particular element belongs. To address this challenge, we propose a *spatio-temporal annotation language (STAL)* to annotate elements of each individual model with space/time information in the form of $<ver_{num}, conf_{info}>$, where ver_{num} denotes the version number of a particular model (e.g., 1st version, and so on), while $conf_{info}$ denotes space dimension-related information (e.g., smart home configuration, organization type or size, etc.).

Syntax and Semantics of STAL. In the time dimension, models can evolve (independently and asynchronously) over distinct *timepoints*. Since timepoints can be corelated and compared, they naturally form a *chronological order*. Given this inherent chronological nature of models' evolution, a sequence of versions of a particular model can be annotated with sequential version numbers: $ver_1, ver_2, ver_3 ... ver_n$. 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 with the ver_{num} format in STAL could represent version numbers or dates, 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 use the naming conventions $conf_X, conf_Y, ..., conf_Z$ (instead of $conf_1, conf_2, ..., conf_n$) 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, in Fig. 1, we used the names $conf_A$ = "Student smart home" and $conf_B$ = "Senior Smart home" as the names of the two different configurations of smart homes. However, it is worth mentioning that information about configurations could be composite (i.e., consists of several pieces of information). For example, if we want to model different configurations or types of smart houses (similar to TYPE1, TYPE2, and TYPE3 in [7]), where each type refers to a home of a specific size, location, and occupant kind, then we need to take these information into

consideration. For instance, TYPE1 refers to homes that are of medium size, located in Ontario, and meant for seniors. 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* (see Table 1), which provide 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 and they are just descriptions of the configuration.

Table 1. Mapping configurations to their descriptions

Configuration	Description
TYPE1	Size = Medium, Location = Ontario, Occupants = Seniors
TYPE2	Size = Large, Location = Ontario, Occupants = Students
TYPE3	Size = Medium, Location = Quebec, Occupants = Seniors

Annotating the Propositionally-Encoded Models. We annotate the propositional encodings of model elements with information about their versions and/or configurations. For example, assume that the model *m* in Fig. 3 (right) represents a second version (ver_2) and a configuration X ($conf_X$) of a particular model. Then, the propositional encoding of *m* with annotation $PE(m_{annot})$ of this model becomes:

$$PE(m_{annot}) = S1\text{-}State\text{-}\langle ver_2, conf_X\rangle \wedge T1\text{-}Transition\text{-}\langle ver_2, conf_X\rangle \wedge S2\text{-}State\text{-}\langle ver_2, conf_X\rangle \wedge L1\text{-}S1\text{-}T1\text{-}srcAssoc\text{-}\langle ver_2, conf_X\rangle \wedge L2\text{-}T1\text{-}S2\text{-}tgtAssoc\text{-}\langle ver_2, conf_X\rangle$$

Given a set of annotated, propositionally-encoded models and based on Def. 5, the union of these models is the union of their annotated propositional variables:

$$PE_{Uannot} = PE(G1_{annot}) \cup PE(G2_{annot}) \ldots \cup PE(Gn_{annot})$$

Annotating the Union of Propositionally-Encoded Models with STAL. In a model family, it is possible one model element belongs to several or all family members. For instance, assume that there is a model family with one model configuration ($conf_A$) that evolves into five versions (i.e., ver_1 to ver_5). Assume also a node *n* that belongs to the five versions of that model. Now, to construct a union model, we need to unify the annotated propositional variables of these five versions. In this case, *n* will be annotated in the union model with five annotations: $\langle ver_1, conf_A\rangle$, $\langle ver_2, conf_A\rangle$, $\langle ver_3, conf_A\rangle$, $\langle ver_4, conf_A\rangle$, $\langle ver_5, conf_A\rangle$. Such style may lead to large amounts of annotations.

To simplify annotations of union models, STAL represents a sequence of version annotations as a range of values (*[start:end]*). In the above example, the annotation of *n* becomes $\langle [ver_1: ver_5], conf_A\rangle$. Many sequences can also be used, e.g., for ver_1 to

ver_7 skipping ver_4, we get:<*[ver1: ver3] [ver5: ver7]*, $conf_A$> . If an element belongs to all versions and all configurations of a family, we annotate it with the keyword *ALL*.

Example. We use a simple state machine example, with two versions of a model, as shown in Fig. 4. The union model combining these two versions is expressed as follows:

PEU_{annot} = S1-State_<ALL> ∧ T1-Transition-<ver >$_1$∧ S2-State-<ALL>
∧ T2-Transition-<ver$_2$> ∧ L1-S1-T1-srcAssoc-<ver$_1$>
∧ L2-T1-S2-tgtAssoc-<ver$_1$> ∧L1-S1-T2-srcAssoc-<ver$_2$>
∧ L2-T2-S2-tgtAssoc-<ver$_2$>

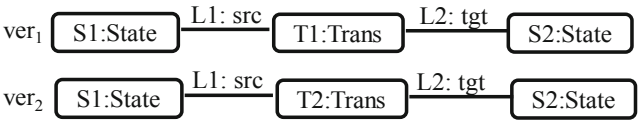

Fig. 4. Two versions of a state machine diagram

In this paper, we limit ourselves to simple type graphs, where attributes of model elements have to be expressed structurally with named nodes and edges. In future work, we will also assess the benefits of extending the definitions of basic type and typed graphs with explicit attributes, for instance using Ehrig's attributed type graphs (or E-graphs) [4], where special edges are used for attributes.

5 Reasoning and Analysis with Union Models

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 answer this research question, we consider three reasoning tasks (RTs), namely: property checking (which is already known in the literature), trend analysis, and significance analysis (which we proposed for this work). Then we compare the performance of the three RTs using M_U as opposed to using individual models.

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

RT1: Property Checking. Property checking on models aims to verify if a model satisfies a particular property or not. 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 a group of state machine diagrams contains self-looping edges or not, or she may check if 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 paper, 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. So, we limit our approach to checking those properties related to language syntax, independently from any language specificity. Hence, "property" here means "syntactic property". 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 Φ`c:∀t:Transition ∃s:State|t.src=s`, it can be grounded over the vocabulary of the model in Fig. 5. as follows:

`T1-S1-S2-Transition⇒ S1-State ∧T2-S2-S3-Transition⇒ S2-State`

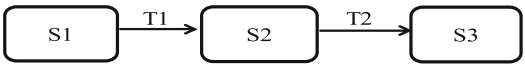

Fig. 5. An example of propositional encoding of a property

It is important to emphasize here that the example in Fig. 5. is just a proof of concept and it does not adhere to our formalization of typed graphs (Sect. 3). As can be noted, the example considers the graphical representation of the state machine presented in the canonical form in Fig. 3. (right), 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 (Sect. 3.2) and properties, the task of property checking can be defined as follows:

Definition 6–Property Checking: Given model *m* and a property *p*, and their propositional encodings Φm and Φp, respectively, we check if the expression Φm ∧ Φp is satisfiable or not using a SAT solver.

RT2: Trend Analysis. The idea of this analysis is to search for a particular element across members of a model family and study the *trend* of that element. By "trend" we mean 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 in all members of a GRL family to conduct a trend analysis about the properties of that goal (e.g., its importance value, or satisfaction value) and observe how that value changes across model version/variations to get some insights about its evolution pattern.

RT3: Significance Analysis. We suggest this type of analysis to enable modelers to check for those elements that are common in all (or part) of versions (i.e., time) or variations (i.e., space) of a model family. Elements that are common among all models can be inferred to be essential or significant. For example, if a modeler is investigating several design options of a particular system, and she needs to know which elements are significant (i.e., mandatory for design) in all design options, then she would conduct this analysis *once* using the M_U of the model family she has at hand (instead of doing a pairwise search on each version/variation of individual models).

6 Experiments

We assess the feasibility of reasoning using M_U empirically. We ran experiments with parameterized random inputs that simulate different settings of various reasoning and analysis categories. In this paper, we build on the formal semantics of union models (Sect. 3) and use formalized GRL models and state machine models. However, our approach is also applicable to other metamodel-based languages.

6.1 Methodology

We ran two experiments (named Exp.1 and Exp.2) to evaluate the feasibility of using M_U with RT1, RT2, and RT3. In Exp.1, we measured the total time needed to perform one task on each individual model, one model at a time. We refer to this time as T_{ind}. In Exp.2, we measured the time needed to accomplish the same task with M_U. We refer to this time as T_{MU}. Then, we compute improvements with a metric called time *speedup* (as used in [8]), defined originally as: speedup_old = T_{ind}/T_{MU}. However, to be fairer and more realistic in our experiments (especially for large models), we decided not to neglect the time needed to construct M_U (although it is quite small and can performed once before being amortized over multiple analyses). We call $T_{construct}$ the time needed to construct M_U, and the time speedup is then calculated as: speedup = $T_{ind}/(T_{construct + TMU})$. A speedup larger than 1 is a positive result, and the larger the speedup, the better.

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 (*I*). To control the possible combinations of parameters SIZE and I, we discretized their domains into categories (following Famelis' methodology [8]). For parameter SIZE, we defined four categories 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, we calculated the representative exemplar of each category by setting n to be the median of two successive numbers in the seed sequence. The ranges of the categories and the selected exemplars for each category are shown in Table 2. These numbers are in line with our own experience dealing with goal models and state machines of various sizes.

We followed the same methodology for the number of individual models, *I*, using a seed sequence (0, 4, 8, 12, 16). The four size categories (S, M, L, XL) are shown in Table 3.

Table 2. Categories of parameter SIZE (size of 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 parameter I (number of individual members in a model family)

# of individual models (*I*)	(0, 20]	(20, 72]	(72, 156]	(156, 272]
Exemplar	6	42	110	210
Category	S	M	L	XL

To evaluate the property checking task (RT1), we encoded each annotated individual model *m* in a model family *MF* as a propositional logic formula $\Phi m = \bigwedge e_i \in PE(m_{annot})$. We also encoded a union model M_U of that MF as $\Phi M_U = \bigwedge e_i \in PE_{Uannot}$. Furthermore, the property to be checked was encoded into a propositional formula Φp. Then, a SAT solver was used to check if the encodings of each of the individual model and their union model satisfy (or not) the property. In particular, for each individual model, we constructed a formula $\Phi m \wedge \Phi p$. The property is said to hold in any model if and only if this formula is satisfiable. Similarly, we constructed the formula $\Phi M_U \wedge \Phi p$ and checked whether the property was satisfiable. In both experiments (using the same computer), we recorded the time it took to check a property on individual models (T_{ind}) and compared it to the time needed to do the check on union models (T_{MU}).

6.2 Implementation

To validate our approach, we had a prototype implementation in Python 3.6 to represent models (GRL and state machine models) as typed graphs (based on Defs. 3 and 4), and construct their M_U according to Def. 5. We used NetworkX 2.2 [9] to implement typed graphs, and we implemented our own union algorithm to construct M_U by adapting NetworkX's built-in union function as a building block. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex graphs [9]. It is enriched with a variety of features from the support of graph data structures and algorithms to analysis measures to visualization options. We used NetworkX's graph generators to randomly generate valid typed graphs (with different parameters SIZE and I) with likely evolutions. We checked a sample of the generated graphs manually to make sure that we are generating likely changes to existing models rather than generating independent models. Then, we assigned attributes to nodes and edges of these generated graphs to reflect attributions

and typing information of real models. We then constructed G_U from the generated graphs using our union algorithm. G_U is the union of a set of typed graphs, and hence G_U corresponds to M_U.

For RT1, we manually generated propositional formulas for state machine diagrams (both individual models and their M_U). We checked the "*cyclic composition property*" inspired from [10], which ensures that "the model does not contain self-looping edges". A propositional formula was also generated for this property. The propositional encodings were generated according to the rules discussed in Sect. 3.2, and they were fed as literals to the MiniSAT solver included in the SATisPY package [11]. SATisPy is a Python library that aims to be an interface to various SAT solver applications.

6.3 Results

This section is organized according to the experiments conducted to evaluate RT1, RT2, and RT3. All figures illustrated in this section represent the average results of 15 runs.

Results for RT1. Figure 6 illustrates the time speedup of performing property checking, first with a set of individual state machine diagrams (represented as typed graphs) and then with their M_U. In this experiment, we checked the satisfiability of the cyclic composition property (Sect. 6.2). Figure 6 shows that the use of M_U for property checking achieves a significant time speedup compared to performing the same task on a set of individual models separately. The highest speedup (=365) was observed with a large number of individual models (i.e., I = L) that are of a small size (i.e., SIZE = S). The smallest speedup (=2.54), on the other hand, was observed when both I, and SIZE parameters are of category XL. In addition, for all categories of I, there is a noticeable pattern of speedup degradation 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 went below 1, which means that even with very large models (with I = XL and SIZE = XL), the time to perform property checking on a

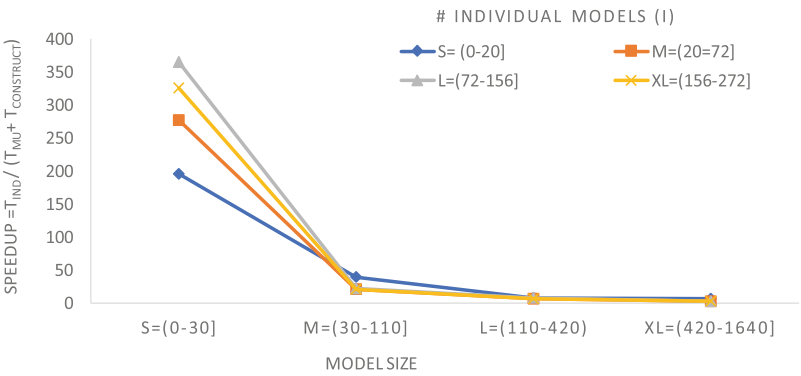

Fig. 6. Average speedups achieved by using MU to perform property checking (RT1)

group of such models (using M_U), with considering the time to construct M_U, is still better than performing property checking on individual models.

Results for RT2. In this experiment, we conducted a trend analysis on an element named GoalX from a set of GRL individual models and their 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. To perform this analysis on M_U, we simply searched for and retrieved an element named X of type Goal, annotated with any version number (i.e., `X-Goal-<{ver`$_i$`}>`), where {ver$_i$} reflects the set 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 reality involves opening each individual model, searching about the desired element, observe its importance value, and close the current model, iteratively for each model. Figure 7 illustrates the time speedup gained in this experiment. The results illustrated in this figure show a pattern close to the results of RT1 (i.e., property checking). This is somewhat expected as both the property checking task and the searching task (which is the core of trend analysis) have a linear time complexity. From Fig. 7, it can be noticed that the use of M_U reduces the time to search for elements that belong to a group of models instead of traversing each individual model, separately. The highest speedup (=296) was achieved when I = XL and SIZE = S, and the lowest (=5.9) when I = S and SIZE = XL. The decrease pattern of speedup gained in this experiment is almost close to the one illustrated in Fig. 6.

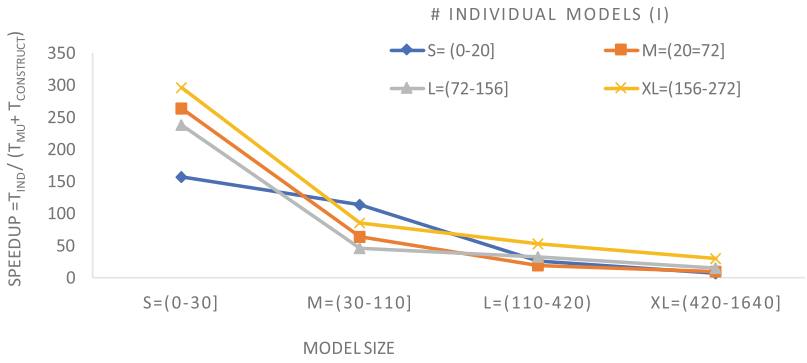

Fig. 7. Average speedups achieved by using MU to perform trend analysis (RT2)

Results for RT3. Figure 8 shows the time speedup for significance analysis on a set of GRL models and their M_U. In this experiment, we searched for all elements that are common between all model versions. This is a tedious task, especially when the number and the size of models increase. Searching a set of *M* individual models, with *N* elements each to find elements in common between all models has a complexity of $O(M \times N^2)$. However, with M_U, we only use one model to search elements in common, where the search task is reduced here into searching for elements annotated with <*ALL*> . The time speedup gained in this experiment is more significant than in the

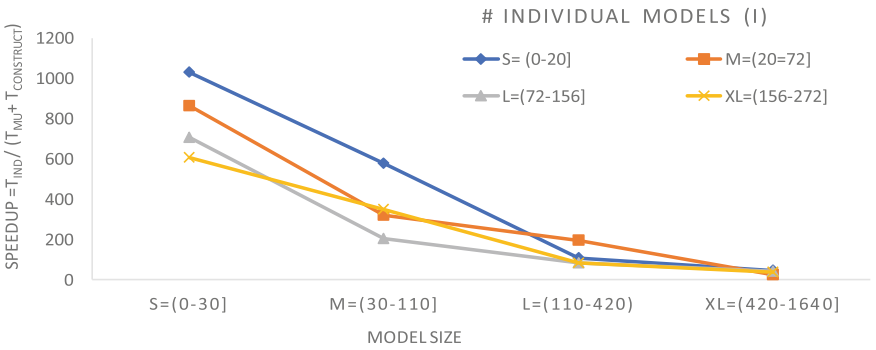

Fig. 8. Average speedups achieved by using MU to perform significance analysis (RT3)

experiments for RT1 and RT2, as the potential gain here is quadratic rather than linear. Again here, there is a decrease in the speedup as the model SIZE increases.

We noticed in some experiments with particular settings (related to variation of models, size, number of models in a family) that the time saving achieved from using M_U was a few minutes (about 15 min for some model families).

6.4 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., from real-world model families.

Another threat to validity is related to the experimental parameters, where we used only *SIZE* and *I*. 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 complexity of a property to be checked might also be another parameter to consider.

Our experiments need to be elaborated further for more complex analysis techniques found in typical goal modeling such as top-down and bottom-up satisfaction propagation. The results could also be compared to approaches that handle some variability in the time dimension (only) for goal models, including the work of Aprajita et al. [12] and of Grubb and Chechik [13]. Furthermore, the current analysis covers two modeling language (goal models and state machines) and it 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.

7 Related Work

There are few approaches proposed in the literature to support *model families*. Shamsaei et al. [7] defined a generic goal model family (using GRL) for various types of organizations in a legal compliance domain. They annotate models with information

about organization types to specify which ones are applicable to which family member. Different from our work, the work of [7] handles only variation of models in the space dimension and does not consider evolution over time. Also, the authors focused only on maintainability issues and did not propose union models to improve analysis complexity and reduce analysis effort. Palmieri et al. [14] elaborated further on the work of [7] 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 [7], Palmieri et al. considered further dimensions such as the organization size, type, the number of people, etc. However, they did not consider 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 to efficiently handle families of software [15]. Although both of our work and the SPL domain have the concept of "families", their usages are different. In essence, the goal of SPL engineering is to plan for "proactive reusability", which means to strategically maintain a set of modeling artifacts (with high-level features) to exploit what variants have in common to derive or create new desirable products. The goal of our work, however, is not to plan for reusability but to analyze families of models more efficiently using union models.

The notion of a *feature* is central to variability modeling in SPL, 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, where each variable corresponds to a feature. FMs characterize the valid combinations of features as a configuration. A configuration defines, at a conceptual level, one product which can be extracted from the SPL. Yet, a FM is different from an M_U in both usage and formalism. A FM represents variability at an abstract "feature level" which is separate from the software artifacts (like a grammar of possible configurations), whereas M_U represents variability of all existing models at the "artifact level" itself.

To express variability, annotative approaches are commonly used in the literature, as in the work of Czarnecki and Antkiewicz [17], where 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 to derive or create a concrete software system as variant of the SPL. Using a negative variability mechanism [17], annotative approaches define a so-called 150% model that superimposes all possible variations of for the entire SPL. The 150% model is used to derive a particular variant, while other irrelevant parts are removed. While union models have some similarities to 150% models, the usage of both models, the domains they are used in, and the way of annotating them are different.

The approaches proposed by Seidl et al. [18] and Lity et al. [19] are closely related to ours. They considered variation of software families in space and time, and explicitly annotated variability models with time and space information to distinguish between the different versions and variations of software artifacts. However, this work is done from the SPL perspective (where FMs are essential), while FMs are not used here.

Famelis et al. [20] proposed partial models to capture a set of possible alternative design models with uncertainty. While the idea of capturing models in one partial model is close to our idea of representing models of a family in one union model, our proposed approach is different in two major aspects: the context and the purpose. In essence, we propose union models to enable a more efficient reasoning of multiple models compared to individual models. Partial models, however, are used to describe the observable behavior of a system and to reduce design-time uncertainty.

Mussbacher [21] and Aprajita et al. [12] extended the metamodel of GRL to document explicit changes (additions/deletions) of model elements to specific versions of a metamodel. Although a model family can then be captured, this approach is specific to one language and currently incomplete in the kinds of changes to versions it can accommodate.

Grubb et al. [22] introduced the concepts of "dynamic intentions" into goal models to model alternatives on multiple time scales. The authors proposed a tool-supported method for specifying changes in intentions over time which uses simulation for asking a variety of 'what if' questions about models that evolve over time. Unlike our work, Grubb's approach is limited to goal models (Tropos and *i** in particular), and does not cover variations of models over the space dimension.

The concept of difference and union/merging of models is well investigated in the context of version control systems [23, 24]. For instance, Alanen and Porres [25] proposed an approach to calculate the difference between two models, represented as a sequence of operations, and then extend the difference calculation to form a union algorithm. The union algorithm calculates the union of two models based on their differences from a given original/base model, where two separate modifications are made to a base model, and the union algorithm combines both differences into one model by interleaving the operations from the latter difference with the former difference. For example, given a base model M_{base} and two alternative model versions M_1 and M_2, the union of these models, denoted as M_{final}, is calculated as: $M_{final} = M_{base} + (M_1 - M_{base}) + (M_2 - M_{base})$. This mechanism, known as *three-way merge*, is mainly concerned with tracking and highlighting the changes that happen across models, and calculates the final model based on the *differences* from the original model, without backward traceability to the source of the changes. This is different from our union algorithm, where we calculate the union model by taking all elements that belong to all versions of models, with an additional feature that annotates elements to indicate to which version they belong.

8 Conclusion and Future Work

This paper proposed union models as a modeling paradigm to support the representation of model families (for time and space dimensions) using one generic model. Elements of a union model are annotated with information about time and space using a new spatio-temporal annotation language (STAL) in order to distinguish which element belongs to which model. The paper is contributing a formalization of union models that simplifies the creation of such models while enabling several types of efficient analyses.

Our experiments indeed demonstrate the usefulness of union models for analyzing a family of models, all at once, compared to individual models.

For future work, we plan to extend our empirical evaluation by having more experimental inputs, parameters, and tasks, by using existing model families, and also by considering other categories of analysis techniques (such as GRL top-down propagation) and other modeling languages. We expect that some analysis techniques will need to be adapted from a single-model context to a model-family context; the circumstances imposing such adaptation and the effort required to adapt the analysis techniques need to be identified and better understood. Usable tool support is also being developed.

Acknowledgement. We would like to thank the anonymous reviewers, as well as Prof. Michalis Famelis, for their comments and feedback, which helped us improve the presentation of this paper. We also thank the Ontario Trillium Scholarship program, the NSERC Discovery program, and the BMO Financial Group Graduate Bursaries for their financial support.

References

1. Micouin, P.: Model Based Systems Engineering: Fundamentals and Methods. Wiley, Hoboken (2014)
2. ITU-T: Recommendation Z.151 (10/18) User Requirements Notation (URN) – Language definition (2018). https://www.itu.int/rec/T-REC-Z.151/en
3. Amyot, D., Mussbacher, G.: User requirements notation: the first ten years, the next ten years. J. Softw. **6**(5), 747–768 (2011)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of algebraic graph transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
5. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47884-1_16
6. Alwidian, S., Amyot, D.: Relaxing metamodels for model family support. In: 11th Workshop on Models and Evolution (ME 2017), vol. 2019, pp. 60–64. CEUR-WS (2017)
7. Shamsaei, A., et al.: An approach to specify and analyze goal model families. In: Haugen, Ø., Reed, R., Gotzhein, R. (eds.) SAM 2012. LNCS, vol. 7744, pp. 34–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36757-1_3
8. Famelis, M.: Managing design-time uncertainty in software models. Doctoral dissertation, University of Toronto, Canada (2016)
9. NetworkX. https://networkx.github.io/. Accessed 05 June 2019
10. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45221-8_28
11. SATisPY Solver. https://github.com/netom/satispy. Accessed 15 June 2019
12. Aprajita, Luthra, S., Mussbacher, G.: Specifying evolving requirements models with TimedURN. In: Proceedings of the 9th International Workshop on Modelling in Software Engineering, pp. 26–32. IEEE Press (2017)

13. Grubb, A.M., Chechik, M.: Modeling and reasoning with changing intentions: an experiment. In: 2017 IEEE 25th International Requirements Engineering Conference (RE), pp. 164–173. IEEE CS (2017)
14. Palmieri, A., Collet, P., Amyot, D.: Handling regulatory goal model families as software product lines. In: Zdravkovic, J., Kirikova, M., Johannesson, P. (eds.) CAiSE 2015. LNCS, vol. 9097, pp. 181–196. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19069-3_12
15. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005). https://doi.org/10.1007/3-540-28901-1
16. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. **51**(2), 456–479 (2007)
17. Czarnecki, K., Antkiewicz, M.: Mapping features to models: a template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005). https://doi.org/10.1007/11561347_28
18. 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), vol. 1, pp. 22–31. ACM (2014)
19. 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), pp. 27–34. ACM (2018)
20. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: 34th International Conference on Software Engineering (ICSE), pp. 573–583. IEEE CS (2012)
21. Mussbacher, G.: TimedGRL: specifying goal models over time. In: IEEE International Requirements Engineering Conference Workshops (REW), pp. 125–134. IEEE CS (2016)
22. Grubb, A.M., Chechik, M.: Looking into the crystal ball: requirements evolution over time. In: 24th International Requirements Engineering Conference (RE), pp. 86–95. IEEE CS (2016)
23. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. Int. J. Web Inf. Syst. **5**(3), 271–304 (2009)
24. Förtsch, S., Westfechtel, B.: Differencing and merging of software diagrams–state of the art and challenges. In: Filipe, J., Helfert, M., and Shishkov, B. (eds.) Second International Conference on Software and Data Technologies (ICSOFT), pp. 90–99. INSTICC Press (2007)
25. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45221-8_2