Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Characterizing Black-box Composition Operators via Generated Tailored Benchmarks

Benjamin Benni[a] Sébastien Mosser[b] Mathieu Acher[c]
Mathieu Paillart[a]

a. Université Côte d'Azur, I3S, CNRS, France

b. Université du Québec à Montréal, Québec, Canada

c. Université de Rennes 1, Inria/IRISA, France

Abstract The integration of a model composition operator into a system is a challenging task: the properties associated with such operators can drastically change how the developers will be able to use it. In this paper, we describe a modelling framework that allows a software developer, who is not an expert in model composition, to describe the interface of the operators she wants to use, and describe the properties she expects from them to fit her needs (*e.g.*, idempotence, commutativity, associativity). This abstract description is used to pilot a property-based testing approach on generated code. We applied the approach to two case studies: feature model composition and Git merging.

Keywords Software Composition; Testing; Composition Operator.

1 Introduction

The *Separation of Concerns* (SoC) paradigm advocates the design of large systems through the composition of elementary artifacts, tackling the intrinsic complexity of software-intensive systems. This paradigm is used in many different contexts, and at different levels of abstraction, to tame the complexity of nowadays' software development. However, decomposing a system is just the hidden part of the iceberg, and it intensively relies on composition operators that are buried in the core of the decomposition framework used. For example, one can decompose the modeling of a system into several UML diagrams, and rely on the UML package merge mechanism [ZDD06] to recompose the whole structure in the end. Similarly, one can decompose reusable behavior in a given system as aspects [KHH+01], and rely on the aspect weaving mechanism to integrate these new behaviors into the legacy system. If model-driven engineering often refer to these approaches as *merging* operators, we generalize it

Benjamin Benni, Sébastien Mosser, Mathieu Acher, Mathieu Paillart. *Characterizing Black-box Composition Operators via GeneratedTailored Benchmarks.* Licensed under Attribution 4.0 International (CC BY 4.0). In Journal of Object Technology, vol. 19, no. 2, 2020, pages 7:1–20.
doi:10.5381/jot.2020.19.2.a7

by refering to *composition*. For example, the intensive use of containers to deploy micro-services architectures [BMCR18] relies on a composition operator, as a container image is composed of its predecessors in the container build chain.

The software engineer who relies on the composition operator to implement the system under development expects guarantees on the operator, *i.e.*, how to integrate this existing implentation into her own development. Consider, for example, aspect weaving, where aspects $\alpha_i \in A$ are woven into a base program $p \in P$, using the aspect weaving operator. This operator, such as its implementation provided by AspectJ [KHH+01], is an exogenous composition operator that weaves a single aspect into a base program. Thus, by definition, it is ordered, and there is no guarantee that the weaving of two different aspects will commute. When an automated engine takes adaptation decisions with no ordering guarantee on the decision process (*e.g.*, designed using machine learning techniques), aspect commutativity must be ensured. In such a context, other composition operators exist, which are commutative by design. For example, HyperJ [TOJ02] considers a program p as the symmetric composition of program slices $\{s_1, \ldots, s_n\}$, *i.e.*, $p = \{s_1 \oplus \ldots \oplus s_n\}$, with $\oplus$ a commutative operator. Restricted to business processes, the ADORE approach [MB13] considers the definition of a business process p as the asymmetric weaving of a fragment f (like AspectJ), but provides a commutative fragment merge operator (denoted as μ), *i.e.*, $p' = p \leftarrow \mu(f_1, \ldots, f_n)$.

From a software engineering point of view, the algebraic properties ensured by the operators are essential to consider, as they shape how the composition method can be integrated in a system. Unfortunately, the internals of composition operators are rarely modeled, and their intrinsic complexity makes static analysis tools challenging to use in this context. Moreover, the software composition operators are not always defined by modeling experts, and software developers face the challenge of defining one for their very domain, or to chose one among several existing alternatives. On the one hand, it is difficult for developers who have no expertise in the narrow field of software composition to face all the associated challenges used to design or reuse such operators. As a consequence, the composition code is often complex, written in languages that were not explicitly designed for that, and must be considered as black-boxes. On the other hand, software testing is a well-known approach used to validate pieces of software, regularly used by software developers to address black-box validation.

In this paper, we propose to support software developers facing the challenge of integrating composition operators by providing a domain-specific language that models the elements to be composed, and the expected properties of such operators in a domain-independent way. We then use model transformations to transform these models into executable elements using state-of-practice tools to validate the modeled properties on the designed elements empirically.

2 Motivating example: building a catalogue of products

We consider here the situation of a software engineer who needs to build a catalog of existing products. This use case is typical in variability management and is classically addressed by merging *Software Product Lines* (SPLs). According to this approach, each product is considered as an SPL that does not contain any variability (as it is a single product). Then, an union operator is used to compose all the non-variable artifacts into an SPL that models all the input products. To perform such a task, one

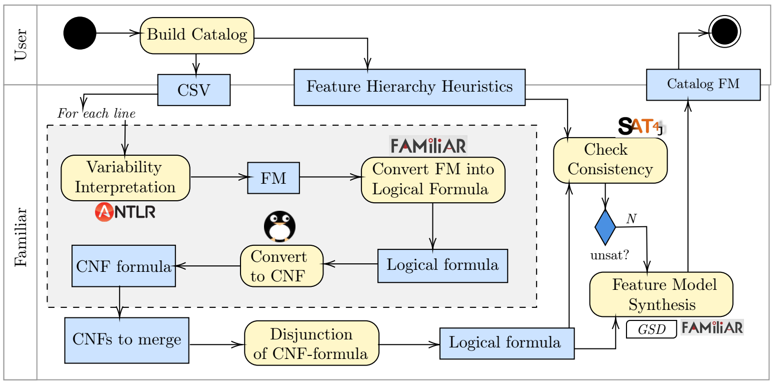

Figure 1 – Simplified activity diagram of building a catalog with Familiar

can rely on the Familiar language [ACLF13]. This language is designed to model SPLs as *Feature Models* (FMs), a modeling approach tailored for software variability that organizes features hierarchically using trees. The language is tooled with operators to manipulate FMs and also provides a Java API to let engineers embed Familiar inside their software. Familiar is a good example of *a black-box software composition*: it relies on a complex process to achieve feature model composition, such as union or intersection. Figure 1 describes a simplified version of the catalog-building process using the *merge union* operator. Considering a set of products stored in tabular data (*i.e.*, CSV files), the process converts each product into an FM (*i.e.*, a tree-structure modelling the characteristics of the product). Then, the FMs are transformed into logical formulas represented in *Conjunctive Normal Form* (CNF), and the merge operator operates on the CNFs to create a new formula that represents the union ($\cup$) of the input ones (a disjunction). Finally, as the FM to CNF transformation is not bijective, heuristics are used to transform the merged CNF into a human-readable FM. It also involves a SAT-solving step to check the consistency of the merged FM.

This process strongly relies on the fact that the *merge union* operator is *(i) idempotent* (as the very same product can be present several times in the input dataset), *(ii) commutative* (as the input order of the products is not guaranteed) and *(iii)* associative (as the merge operator is binary, then being applied in sequence to merge a set of products). However, considering the complexity of the mechanisms involved in the operator implementation and the fact that it uses heuristics, it is not possible to demonstrate that the composition operator guarantees such properties at the *implementation* level. Unit tests can be defined to validate chosen cases, but it is up to the operator's developer to invest in such a testing effort. In Familiar, the test suite[1] represents 125 test files that implement 626 unit tests ($\sim 18,500$ lines of codes). As a consequence, a developer who wants to integrate Familiar to build product catalogue has no guarantee about how the opeprator will behave in her context, and it is up to her to setup an experiment to validate that these three properties are respected by Familiar.

[1] https://github.com/FAMILIAR-project/familiar-language/tree/master/familiar.test

3 Background

In the previous section, we presented an real example of composition operator, and the assumption that one makes when using it regarding its algebraic properties. In this section, we perform a mapping between algebraic properties defined at the mathematical level and the impact of such properties on how a composition operator can be used in an operational context.

3.1 Algebraic foundations of composition laws

We saw in the previous section that a simple process (*i.e.*, using an existing operator to apply it to our context) triggered challenges that were not expected at the beginning. Also, according to the complexity of the composition operator, there is no formal proof that the operator's implementation respects the fundamental algebraic properties. We underline that the previous section depicted an example, and that these characteristics are not tied-up to the FAMILIAR use-case. In this section, we define composition operators from an algebraic point of view and map the associated properties to benefits available for software engineers.

From an algebraic point of view, a *composition law* is defined as a binary operation, denoted as $\bullet$, defined over three sets of elements representing the elements to be composed: its domain ($E \times F$) and its codomain (G): $\bullet : E \times F \rightarrow G$. When $E = F = G$, $\bullet$ defines an *internal law*, for example when two models (*e.g.*, UML class diagrams) are merged to produce a new one. If $E \neq F \wedge (G = E \vee G = F)$, the operator $\bullet$ defines an *external law*, which can be *right-external* (*i.e.*, $E \neq F \wedge G = E$) or *left-external* ($E \neq F \wedge G = F$). For example, in ADORE, weaving is a right-external law (where fragments operate on business processes to yield enriched processes), and the merge operator μ is an internal law that operates on fragments to compose them:

$$\begin{array}{rcccc} \leftarrow: & \mathit{Process} & \times & \mathit{Fragment} & \rightarrow & \mathit{Process} \\ \mu: & \mathit{Fragment} & \times & \mathit{Fragment} & \rightarrow & \mathit{Fragment} \end{array}$$

For the sake of concision, we consider in this paper only internal laws. However, it is possible to extend this work to external laws, but it makes the notations cumbersome and triggers a lot of duplicated definitions according to the left or right "externality" of the law. Thus, we consider here a *magma* $M = (E, \bullet)$, defined as a set E equipped with an internal composition law. By definition of an internal law, the application of $\bullet$ yields an element of E. If the law is not total (*i.e.*, not defined for every element of E), the magma is considered as partial, but it does not impact its compositional properties.

To assess properties, one needs to compare models. A prerequisite of model comparison is the definition of a *preorder*: $\sim \subseteq E \times E$. A preorder is a binary relation defined as reflexive ($\forall e \in E, e \sim e$) and transitive ($\forall x, y, z \in E^3, (x \sim y \wedge y \sim z) \Rightarrow x \sim z$). If the preorder is antisymmetric ($\forall x, y \in E^2, (x \sim y \wedge y \sim x) \Rightarrow y = x$), it defines an *order relation*, denoted as $\leq$. On the contrary, if the preorder is symmetric ($\forall x, y \in E^2, x \sim y \Rightarrow y \sim x$), it defines an *equivalence relation*, denoted as $\equiv$.

In the FAMILIAR ecosystem, as described in the first subsection of this section, the magma is in our case defined as $M_F = (FM, \cup)$, the *merge union* operator (denoted as $\cup$) and the set of features models FM. The equivalence relation used to assess properties is defined on top of the `compare` function available in the language, which can detect that an FM is a *refactoring* of another one, meaning that they model the very same set of products even if their shape is different.

3.2 From software engineers' needs N_i to algebraic properties P_i

Considering a magma $M = (E, \bullet)$ and an equivalence relation $\equiv$, we can address several needs encountered by software developers in terms of algebraic properties to be guaranteed by the operator. Even if this list would cover all the classical properties from a mathematical point of view, it cannot cover all the domain-specific needs that can exist for each developer. The idea here is to define a “standard library" of properties to be used by developers, and we will describe in the technical contribution section how this library can be extended to fit more specific needs.

N_1**: Support multiple equivalent elements.** Composition might be required in an environment where the dataset to be composed is unreliable. For example, in our motivating example the set of products to be composed might contains the same product several times. Another relevant point here is the consideration of a composition system exposed over a network. In distributed systems, as message delivery is unreliable by design, techniques such as the multiple sending of the very same message can be applied to reduce the rate of message loss. If the operator is compatible with these requirements, it helps the design of the composition process: each received message will trigger a composition. If this property is not ensured, it is up to the software developers to build a mechanism that will take care of duplicated elements. From an algebraic point of view, this property relates to *idempotence* (P_1), defining that for an idempotent law, an element composed with itself yields the initial element. This property can also be used to simplify composition equations, as it is useless to involve the same model multiple times in the composition (saving CPU time, for example).

$$\forall x \in E, x \bullet x \equiv x \qquad \textit{(idempotent)} \qquad (P_1)$$

In our previous example, the *merge union* operator on feature model is expected to be idempotent, as its semantics defines that considering $f = f' \cup f''$, then f models all the products contained in f' and all the ones contained in f'', and no other else. Thus, $g = f \cup f$ must contains all the products contained in f, and no other else, meaning that $g \equiv f$. Classically, *package merge* in UML is also expected to be idempotent by definition. An example of a non-idempotent composition operator is *aspect weaving*, where, for example, weaving multiple times a logging aspect will result in multiple logs at runtime.

N_2**: Foldable operators.** Operators are mathematically related to composition laws, defined as binary entities. It is also a classical decomposition technique applied by software developers, to solve the problem for two elements and then iterate over the set of elements to handle to eventually build the expected results. This approach is strongly advocated by functional languages that implement *fold* operators to compose binary functions. For example, in Scala, one can build the sum of the first 100 integers by applying the `foldLeft` operator to the list of elements and the binary `+` operator.

$$\sum_{i=0}^{100} i = 0 + 1 + \ldots + 100 \qquad \texttt{(1 to 100).foldLeft(0)(\_+\_)}$$

Even if this approach seems natural, it can only be applied in a legit way if the developers have guarantees on the fact that the sequential application of the operator will yield the expected result. From an algebraic point of view, this is related to the notion of *alternativity* and *associativity*. *Alternativity* (P_2) is a weaker form of associativity, considering only two models instead of three. A law is alternative when defined as left-alternative (P_{2l}) and right-alternative (P_{2r}). *Associativity* (P_3) is a stronger form of alternativity, where the operator is also left-associative (P_{3l}) and right-associative (P_{3r}) at the same time.

$$\begin{array}{lll}
\forall x, y \in E^2, (x \bullet x) \bullet y \equiv x \bullet (x \bullet y) & \textit{(left-alternative)} & (P_{2l}) \\
\forall x, y \in E^2, y \bullet (x \bullet x) \equiv (y \bullet x) \bullet x & \textit{(right-alternative)} & (P_{2r}) \\
\forall x, y, z \in E^3, (x \bullet y) \bullet z \equiv (x \bullet y) \bullet z & \textit{(left-associative)} & (P_{3l}) \\
\forall x, y, z \in E^3, (x \bullet y) \bullet z \equiv x \bullet (y \bullet z) & \textit{(right-associative)} & (P_{3r})
\end{array}$$

In the previous example, knowing that the binary `+` operator is associative allows the developer to use both `foldLeft` and `foldRight` functions to work on sets of integers. In the Adore example, the fragment merge operator μ has to be associative to support the composition of n fragments (with $n > 2$) at the very same location in the code. According to its definition, the Familiar merge-union operator is also expected to be associative. The same goes for the UML package merge.

N_3: **Avoid useless compositions.** Performing a composition can be time and resource consuming, considering the complexity of the task to be achieved. For example, the UML package merge relies on name matching in models, which is known to be slow when applied to large models. The Adore approach relies on structural mappings to compose fragments, relating the composition to sub-graph isomorphisms detection, which is known to be a NP-Complete problem. In Familiar, the merge union relies on SAT-solving logical formulas, and even considering the recent advances in this domain, it remains a resource-expensive computation.

We already saw in N_1 how the idempotence property P_1 can address this need, by avoiding to trigger a composition if the element is going to be composed with itself. Three additional properties can be taken into account in this context : *regularity* (P_4), *identity* (P_5), and *absorption* (P_6). Regularity follows an idea similar to idempotence, stating that if two elements y and z composed with a base element x yield equivalent elements, then $y \equiv z$. The identity property relies on the exhibition of a remarkable element n that is neutral for the considered composition. For example, considering the addition of integers, 0 is a neutral element. In feature modeling, the empty feature model (*i.e.*, a feature model that does not contain any product) is neutral considering the merge union operator. The absorption property works the other way around, through the exhibition of an absorbing element z (also named a *zero*) that will absorb any other elements composed of itself. For example, the number zero is an absorbing element when considering the multiplication of integers. In feature modeling, the feature model that contains all the possible products is absorbing w.r.t the merge union operator. Like for alternativity and associativity, identity and absorption are defined as left- or right- properties. It is easy to demonstrate that when an element is both left-identical and right-identical (respectively absorbing), then it is unique.

$$\begin{array}{lll}
\forall x, y, z \in E^3, (x \bullet y \equiv x \bullet z) \Rightarrow y \equiv z & \textit{(left-regular)} & (P_{4l}) \\
\forall x, y, z \in E^3, (y \bullet x \equiv z \bullet x) \Rightarrow y \equiv z & \textit{(right-regular)} & (P_{4r}) \\
\forall x \in E, \exists i_l \in E, i_l \bullet x \equiv x & \textit{(left-identical)} & (P_{5l}) \\
\forall x \in E, \exists i_r \in E, x \bullet i_r \equiv x & \textit{(right-identical)} & (P_{5r}) \\
i_l \equiv i_r \bullet i_l \equiv i_r & \textit{(identity unicity)} & (P_5) \\
\forall x \in E, \exists z_l \in E, z_l \bullet x \equiv z_l & \textit{(left-absorbing)} & (P_{6l}) \\
\forall x \in E, \exists z_r \in E, x \bullet z_r \equiv z_r & \textit{(right-absorbing)} & (P_{6r}) \\
z_l \equiv z_r \bullet z_l \equiv z_r & \textit{(zero unicity)} & (P_6)
\end{array}$$

N_4**: Order-Independence.** As we described in the introduction, order matters when composing things. Considering an operator that is order-dependent, it is up to the developer to take care of the order of composition. This problem is a combinatorial one, considering that the selection of k elements among n (denoted as A_n^k) has an order of magnitude of $n!$ in the worst case. Unfortunately, when a composition is required, the models to be composed have already been selected beforehand. It means that they all need to be composed (so $n = k$, denoted as A^n), pushing the combinatorial space to its worst case, *e.g.*, five elements to compose lead to 120 ordered sequence candidates, and it is up to the developer to take this into account.

$$A_n^k = \frac{n!}{(n-k)!} \qquad A^n = A_n^n = \frac{n!}{0!} = n!$$

This situation is classically encountered, for example, when weaving aspects into a base program using AspectJ. The language provides an ordering mechanism to define a partial order among aspects to be woven on shared join points. A similar mechanism is defined in the J2E ecosystem, where one can define so-called interceptors to change component behavior. When multiple interceptors are deployed on the very same component, the runtime engine follows an approach close to a Decorator pattern and executes the interceptors in the declared order. State-of-the-art code rewriters (*e.g.*, Spoon [PMP⁺15], Coccinelle [PLHM08]) rely on a declared application order when several rewriting must be done at the sale location in the source code. The opposite approach is to define an order-independent operator. This is the case for the Familiar merge union, which is expected to yield the same result for any ordered input sequence, as its semantics rely on the set union. Adore follows a similar approach, and the fragment composition operator is order-independent by design. Being order-independent can also simplify runtime execution. For example, in the GreyCat [FHMC18] database engine, we demonstrated that using an order-independent operator allowed us to remove a time-consuming reconciliation step when processing requests. It multiplied by 20 the number of supported queries per second, handling $\sim 20,000$ queries per second instead of $1,000$ on reference benchmarks. From an algebraic point of view, it relates to the *commutativity* (P_7) property.

$$\forall x, y \in E^2, x \bullet y \equiv y \bullet x \quad \textit{(commutative)} \qquad (P_7)$$

N_5**: Compose composition laws.** The previous needs are defined for a given magma, in isolation. However, we saw in the introduction that classical composition approaches could rely on several operators. For example, Adore relies on *(i)* a weaving operator to integrate fragments into a business process, and *(ii)* a merge operator to compose fragments when required. The Familiar language is a language defined to manipulate feature models and then provides several merge operators as well as intersection, aggregation and slicing operators. The software developer needs to take care of the composition of the different laws, as it might impact the way the different elements can be composed together. A classical example is to consider multiplication and addition over integers, where it is possible to propagate a multiplication over an addition: $a \times (b + c) = (a \times b) + (a \times c)$. But it is not legal to perform the propagation the other way around: $a + (b \times c) \neq (a + b) \times (a + c)$. In the feature modeling context, for example, the intersection of feature models is expected to support propagation over the union operator (according to set theory).

From an algebraic point of view, it leads to the property of *distributivity* (P_8) and the definition of a *ringoid*. Considering two magmas $M = (E, \oplus)$ and $M' = (E, \otimes)$, a *ringoid* $R = (E, \oplus, \otimes)$ defines an algebraic structure where $\otimes$ distributes over $\oplus$. To be distributive, $\otimes$ must be left-distributive and right-distributive at the very same time (this property is then trivial to demonstrate when $\oplus$ is commutative). Still considering feature models (FM) union ($\cup$) and intersection ($\cap$), the *ringoid* is in this case $R_F = (FM, \cup, \cap)$.

$$\forall x, y, z \in E^3, x \otimes (y \oplus z) \equiv (x \otimes y) \oplus (x \otimes z) \qquad \textit{(left-distributive)} \qquad (P_{8l})$$
$$\forall x, y, z \in E^3, (y \oplus z) \otimes x \equiv (y \otimes x) \oplus (z \otimes x) \qquad \textit{(right-distributive)} \qquad (P_{8r})$$

4 A Language to support developers during characterization

In the previous section, we described a set of classical needs encountered by software developers who have to perform tasks related to software composition regardless of the domain in which it is used. These needs are valid in different contexts: *(i)* a software developer who wants to integrate an existing composition operator into a piece of software and *(ii)* a software developer facing the challenge to define a composition operator from scratch. When developing a composition operator from scratch, it is most likely that this operator will be included in a bigger system, as a portion of it. As depicted in Fig. 1, developing a operator from scratch may involve various steps, different external libraries, and heuristics-based algorithm. In this context, the need to characterize the composition operator and the listed needs still hold but our proposition will act as a canvas to ease the development from scratch of such operator.

Even if our proposition can be applicable in both of these contexts, it may appear to be more useful when helping a developer already facing a legacy composition operator that needs to be characterized before being used.

In this section, we describe the technical part of our contribution, *i.e.*, the definition of a *Domain-Specific Language* (DSL) to support developers when facing the previously described situations and associated needs. We designed this DSL with three objectives:

- Provide a way for a regular software developer to model composable elements, relations and operators in a language-independent way;
- Provide a set of standard properties for composition operators extensible according to the user needs;
- Automate the generation of the glue code to allow software developers to focus on their added-value, *i.e.*, the description of the expected properties and the binding to existing operators.

4.1 Modelling Composable Elements

As the targeted user for the DSL is a *regular* software developer, we chose to leverage a very restricted version of the MOF, where users can only model `Concepts` that contain named `Attributes` (Fig. 2). This is the least minimum for the definition of a composition operator and can be integrated into classical MDE tools (*e.g.*, transformed into an EMF metamodel) thanks to a model transformation. We made this choice to provide a lightweight language to the developers, whose abstract syntax can be integrated into state-of-the-art approaches only when necessary. To support the reuse of structural parts of the designed elements, we provide a generalization relation (using the `<:` symbol), and `Traits` to model abstract concepts that cannot be instantiated by themselves. Traits are used at the model level to organize the elements and avoid

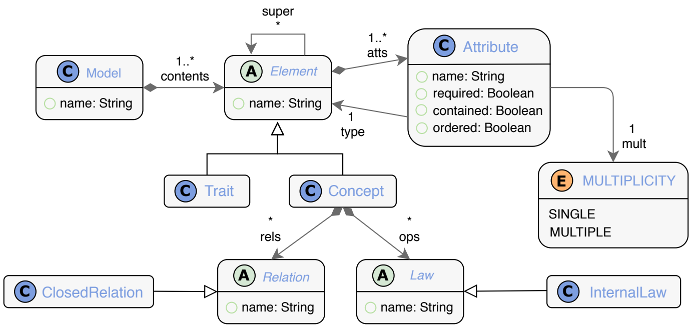

Figure 2 – Simplified meta-modelling language dedicated to internal composition

Listing 1 – Example of composable elements described with the DSL

```
1 model example
2 trait A         { required name: String }
3 concept B <: A  { value: Integer }
4 concept C       { elems: B[0..n] unordered contained }
5
6 operator merge: B x B -> B
7 operator union: C x C -> C
8 relation equivalence: C x C
```

duplication but are not kept at composition time, where only concrete elements can be used.

The language is developed using the ANTLR tool. Based on the designed grammar, we parse the textual representation of the elements, and rely on a VISITOR pattern to generate two artifacts: *(i)* a UML-like representation of the designed elements using PLANTUML [2] (for documentation purposes), and *(ii)* a set of Java classes that implement the structure of the composable elements. The toolchain is driven thanks to the definition of a Maven plugin, which looks for a description file in the `src/resources` directory of the project and triggers the compiler and the associated code generator at compile time.

We depict in LIST. 1 a simple example modelled using the DSL. The context is as follows: the developer has written her concepts and relation using the DSL (as depicted in LIST. 1), and using only this description, our proposition generates a Java framework in which the developer can bind her already existing implementations of the described concepts, or start developing it if necessary. She implements the generated-interfaces that provide a strong guidance for her to focus on what is necessary to assess the property she expects. At the implementation level, she can bridge the generated classes to her legacy operator, or implement the missing parts when necessary.

We describe in FIG. 3 as a UML class diagram, the Java framework generated from the toy example. The code is designed for extension, as it intensively relies on interfaces to implement the modelled concepts (described LIST. 1 l.3,4). A FACTORY is provided to instantiate concrete classes based on these interfaces, following the same

[2] `https://plantuml.com`

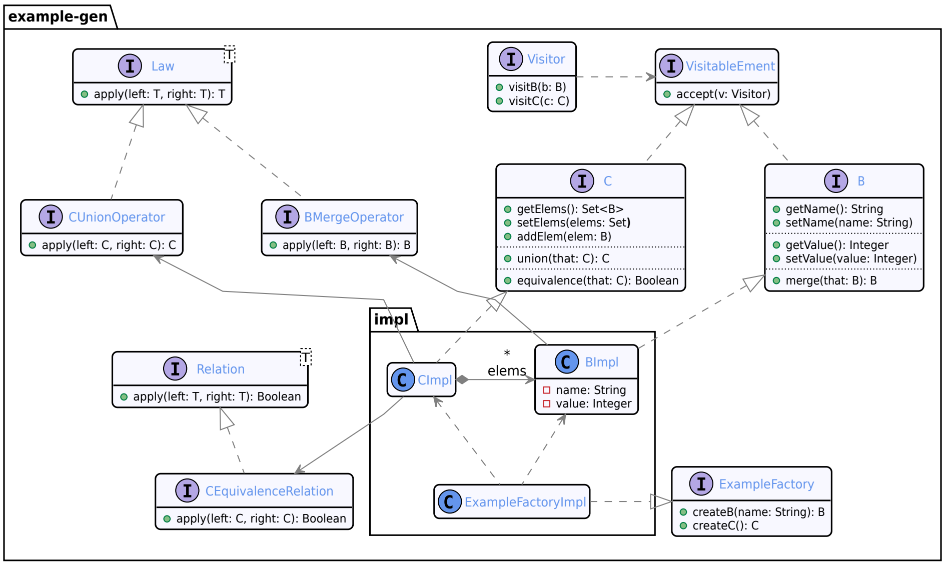

Figure 3 – Java framework generated based on the model described in List. 1

approach as the EMF to provide an extensible back-end. Operators and relations (described in List. 1 l.6-8) are reified as dedicated concepts, and also embedded into the composable elements to provide syntactical sugar for the end-user (*i.e.*, if an operator `op` is defined on a concept `C`, considering `c1` and `c2` two instances of `C`, one can call `c1.op(c2)`). The trait `A` does not exist anymore, as it was only used to support code decomposition at the model level.

4.2 Binding interfaces to composition operators

Based on the previously generated framework, the software engineer can now focus on the definition of the concrete implementation of the operators and relations, as the boilerplate code was automatically generated. To perform this task, one has to extend the generated interfaces, as described in Fig. 4. According to this approach, a composition operator is a *function class* that defines a binary method named `apply`, creating a new element based on the input ones. We decided to avoid side effects in the definition of a composition, as it is classically safer when designing a composition operator to yield a new element instead of modifying one of the inputs. To create the composed element, the software engineer can use any method, *e.g.*, creating a new algorithm from scratch, or calling an external tool or an API. Thanks to the code generated at the previous step, calling the operator on the concrete concept will automatically call the actual operation implemented by the software engineer. If a developer have to create a new operator from scratch, we also leverage the DSL to generate an set of action-based commands that allows her to manipulate her composable elements.

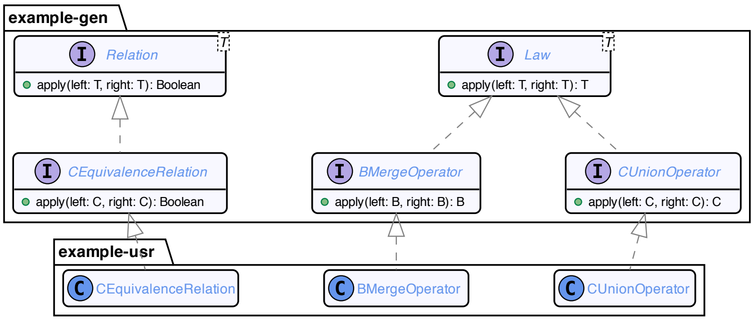
example-gen
T
Relation
apply(left: T, right: T): Boolean
T
Law
apply(left: T, right: T): T
CEquivalenceRelation
apply(left: C, right: C): Boolean
BMergeOperator
apply(left: B, right: B): B
CUnionOperator
apply(left: C, right: C): C
example-usr
CEquivalenceRelation
BMergeOperator
CUnionOperator

Figure 4 – Making a model composable using interface realization

Listing 2 – Example of properties described with the DSL

```
/* Property to be associated to an operator */
property associative(op: Operator, =: Relation, {a,b,c}):
    op(op(a,b), c) = op(a, op(b,c))

/* Property to be associated to a relation */
property transitive(=: Relation, {a,b,c}):
    a = b && b = c && a = c
```

4.3 Ensuring properties on composition operators

Thanks to the language used to model the elements and the extensible framework generated concerning operators and relations, we now reach a state where one can reason about the artifacts at the algebraic level. Using the same DSL as the one used to model concepts, relations and properties, one can design properties, and bind these properties to operators and relations. Thanks to this approach, we provide in a standard library a definition of the properties $\{P_1, \ldots P_8\}$ that address the classical needs of software developers. A developer can also use the same mechanism to design a domain property, specific for domain-requirements (List. 2).

To model properties, we used a functional approach. A given property is then defined as a function that takes as input elements, operators and relations and assembles them into a boolean function. We show in List. 2 an example of the *associative* property (P_3) for an operator. As stated in Sec. 3, it is also important to ensure properties on the designed relations, as an operator assessment mechanism entirely relies on the assumption that the relations conform to their algebraic definitions. It is then possible to express properties to be associated with relations using the same approach in the language (*e.g.*, relation transitivity in List. 2). In both cases, a property takes as input a set of free variables, being operators, relations, or model elements.

Defining properties according to free definition allows us to define a standard library of properties, without being tied to a particular implementation. We use binding mechanisms to bind the free variables exposed (by the definition) to concrete elements.

Listing 3 – Binding properties to operators and relations

```
import stdlib.ace
import example.ace

declare C::equivalence as transitive
declare C::union as associative with equivalence
declare C::union as commutative with equivalence
```

In List. 3, we use the `import` keyword to load[3] the definitions of the standard library and the model shown in List. 1. Then, we `declare` three bindings, to associate the properties (here associativity and commutativity) to the operator (here union) according to a given relation (here equivalence). We also declare that the equivalence relation is expected to be transitive.

According to our assumptions, the software developer can use many techniques to implement the contents of the composition operators and relations, leading to arbitrary complex code. In this context, we can only consider a validation approach, instead of a verification one such as static code analysis. We decided to rely on a *Property-Based Testing* (PBT) approach using the reference framework QuickCheck [CH00] at the implementation level. In a PBT approach, one implements the expected properties as tests taking "elements" as parameters. Then, dedicated generators are used to feed the tests with concrete elements, and the validation approach relies on a probabilistic approach: if a property was true in thousands of instances, one could reasonably think that this property will still be true in real-life usage. Nevertheless, if one counterexample is identified, then the framework can exhibit the specific case that breaks the property.

Based on this approach and the modelled elements, we leverage the models to generate tailored QuickCheck code that will act as characterization benchmarks for the considered operators and relations. For each declaration, we generate a dedicated test using the QuickCheck API. To feed the test with instances, we generate code skeletons, for instance, generators based on the arity of models elements used in the property definition. An example of such code is shown in List. 4.

According to our approach, the validation of a composition operator by a software developer is now restricted to its minimum: *(i)* describe the elements to be composed, *(ii)* describe the signature of the operators and relations, *(iii)* provide an implementation of these elements using any necessary means, and finally *(iv)* fill in the blanks the instance generator generated to feed the characterization benchmark.

5 Case studies: Familiar & Git

The previous section described the internal mechanisms of our contribution from a technical point of view. In this section, we validate the approach by leveraging the framework to assess an existing operator, the Familiar *merge union* used in our motivation example, and on the Git merge use-case to show why our proposition is

[3] The importation semantics is the same as the `#include` C pre-processor directive.

Listing 4 – Example of tailored characterization benchmark using QuickCheck

```
@RunWith(JUnitQuickcheck.class)
public class C_Union_Is_Commutative { // aka CUIC
  @Property(mode = EXHAUSTIVE)
  public void holds(@From(CUIC_Gen.class) C_2_Tuple inputs) {
    C a = inputs.getC1();
    C b = inputs.getC2();
    assertTrue(a.merge(b).equivalence(b.merge(a)));
  }
}

public class CUIC_Gen extends Generator<C_2_Tuple> {
  public CUIC_Gen() { super(C_2_Tuple.class); }

  public C_2_Tuple generate(
      SourceOfRandomness sourceOfRandomness,
      GenerationStatus generationStatus) {
    C_2_Tuple instance = new C_2_Tuple();
    // Generation code goes here
    return instance;
  }
}
```

not tied-up to Familiar. We described in the previous section how one could use the language and the generated framework to create a new operator.

5.1 Composing Feature Models using Familiar

5.1.1 Modelling an existing ecosystem

This modelling step starts with the definition of the elements to be composed (here feature models), and the signatures of the associated operators and relations. For the sake of concision, we do not demonstrate here how one can extensively assess the operator. We will instead zoom in details into the validation of two properties, *associativity* and *commutativity*. In List. 5, we use our modelling language to describe the signatures of the equivalence relation and the *merge union* operator. We then bind the associative and commutative properties to the operator. As the fundamental idea of the approach is to be lightweight and easy to integrate, we model a feature model $f \in FM$ as a named element containing a serialized version of the feature model, using Familiar textual syntax. This is a strength of the approach, as we can use the language to characterize any operator (assuming that one can call it from Java code).

5.1.2 Integrating the operators and relations

We now extend the associated interfaces generated by the compiler, that act as Adapters between the characterization benchmark and the Familiar language ecosystem and associated APIs. We describe in List. 6 how to perform such an adaptation, which relies on the implementation of two classes realizing the interfaces generated from the model description (List. 5), the first class (l.1-10) targets the

Listing 5 – Lightweight modelling of Familiar feature models

```
import stdlib.ace
model fm

trait NamedElement         { required name: String     }
concept FM <: NamedElement { required contents: String }

relation equivalence: FM x FM
operator union: FM x FM -> FM

declare FM::union as associative with equivalence
declare FM::union as commutative with equivalence
```

application of the equivalence relation, and the second class (l.11-21) targets the application of the merge union operator. In both cases, we leverage a Familiar shell to transform the serialized model into a Familiar one, and then rely on the language API to achieve the required task. The equivalence is classically delegated to a comparison framework (*e.g.*, List. 6, line 8). The internal merge operator defined by Familiar is configured according to our needs to build the merged model, which is then serialized into our `FM` concept (List. 6, line 23).

5.1.3 Generating Feature Models instances

The last effort required by the developer is to fill in the code skeleton that will be used to generate instances of feature models used to feed the characterization benchmark. It is necessary to create two different generators: the associativity property is defined on FM^3 and the commutativity one on FM^2. However, both rely on the same principles, so we here only describe the binary generator. As the *merge union* operator works to create a family of products, benchmarking it with totally random and disjoint models would have little if no interest.

It is out of the scope of this paper to describe in detail the structure of a feature model and how to generate one from scratch, and we only give here an intuition of the process. As a feature model is a tree-based structure, the generation is done recursively. Sub-trees are generated, and we randomly chose a way to assemble the sub-trees. This method allows one to obtain a base feature model b. Then, we randomly select several alterations to be applied to b to derive a similar feature model. The idea here is to generate almost equivalent trees, that will stress the merge algorithm.

5.2 Validation on Git merge properties

To highlight that our proposition is not tied-up to the Familiar use-case or any particular application domain, we applied it on the Git merge operation, a frequent operation that happens in distributed code versioning systems. When multiple developers worked on the same file concurrently, the version control system has to build the consolidated version of the file that integrates all the modifications using a *merge* operation. From our point-of-view this merge operation is a legacy composition operator that composes modifications together, checks for eventual conflicts, and applies these modifications on a codebase to yield an updated one.

Listing 6 – Adapting the Familiar ecosystem to the model

```java
public class FMEquivalenceRelation
    implements fm.FMEquivalenceRelation {

  public Boolean apply(FM left, FM right) {
    FMLShell _shell = FMLShell.instantiateStandalone(null);
    FeatureModelVariable vl = _shell.parse(left.getContents());
    FeatureModelVariable vr = _shell.parse(right.getContents());
    return vl.compare(vr).equals(Comparison.REFACTORING);
  }
}

public class FMMergeOperator implements fm.FMMergeOperator {

  public FM apply(FM left, FM right) {
    String rName = left.getName() + "_U_" + right.getName();
    FeatureModelVariable rContents =
      new FMLMergerBDD(Arrays.asList(left, right),
          FDOverApproximationStrategy.SYNCHRONIZED_FLA)
        .mergeFMs(Mode.StrictUnion);
    return FMFactory.createFM(rName, rContents.toString());
  }
}
```

The way the merge operation works is as follow: when one is merging two parallel branches of development, shes merges distant modifications (*i.e.*, made by an other developer) with her own modifications. She expects that *if* the other developer would have done the merge, the result would have been the same, *i.e.*, the two resulting codebase would be exactly the same. Assessing if the code-merge operation is commutative is then critical. However, the code of the *merge recursive* operator is made of 3892 lines of C code that make such an assessment really difficult[4]

Mapped to our proposition, the modeling step starts with the definition of the elements to be composed (here Code), and the signatures of the associated operators (git-*merge*) and property (*commutativity*). In List. 7, we use the DSL to describe the signatures of the equivalence relation and the *merge* operator. We then bind the commutative property to the operator. The framework abstractions will be mapped to Code (l.4), and stubs for equivalence (l.6) and merge (l.5) operation are automatically generated.

At the implementation level, the `merge` operator executes a system call to invoke the `git-merge` command. The equivalence operation is implemented as follows: two files are equivalent if there is no differences between them, *i.e.*, computing the Unix 'diff' on these yield an empty set.

Using this newly implemented classes that use our proposition, one can then assess if the merge operator is commutative, considering a reference corpus of merge scenarios. Where the Familiar case study referred to randmoly generated instances, in this case

[4] `https://github.com/git/git/blob/108b97dc372828f0e72e56bbb40cae8e1e83ece6/merge-recursive.c`

Listing 7 – Lightweight modeling of Git merge operation

```
import stdlib.ace
model code

concept Code  { required path: String }
operator merge : Code x Code -> Code
relation equivalence: Code x Code
declare Code::merge as commutative with equivalence
```

we bound our generators' skeleton to a database that contains thousands of merge scenarios. Based on this setup, the framework did not identify any cases where the commutativity property was violated, comforting us in the fact that the operator can be used in a distributed environment.

5.3 Discussions

By abstracting the domain of software composition and capturing it into a domain-specific language, we believe that the designed language and tool suite provides strong support for software developers who are not experts in software composition. We tried to keep the language as minimalist as possible by only specifying the structure and the signature of the composition artifacts. As a consequence, the developer can choose to rely on our metamodel to develop a composition operator or use the generated code skeletons to adapt an existing ecosystem to the characterization benchmark.

By providing a standard library that implements classical algebraic properties $\{P_1, \ldots, P_8\}$, we free the developer from this knowledge. It is not his/her responsibility anymore to find the right way to assess an algebraic property on a domain-specific operator, based on needs identified at the software developer level. It is out of the scope of this paper to make an automatic link between software developers needs N_i and the associated properties P_j. However, a goal modelling approach to map how each property contributes to a need can be considered to tackle this challenge, and it is one of the perspectives of this work.

The work described here suffers from two main limitations. First, the validation used only two case studies to check the feasibility of the approach and two ecosystems for validation purposes, but we did not validate with actual users the DSL and how we can improve it. We underlined that our proposition can capture operator as complex as the Familiar merge process, and as broadly used as the git-merge operator, in various application domain, still a qualitative validation may be of interest.

The second limitation is a side-effect of relying on PBT to implement the characterization benchmark: a lot of the complexity is pushed to the definition of the generators. It is a classical problem in software testing, as purely random instances might not be relevant for the property to be checked, and considering that the exploration space of all instances of a given concept is gigantic. The code required to generate the instances of feature models in this section was by far the one requiring more engineering efforts in the whole process. This contribution does not have the ambition to contribute to software testing and is purely reusing results from the state-of-t he-art at this level. So, if some breakthrough is made in this field, our proposition will benefit from it. Moreover, we believe that PBT is a good trade-off for software composition, considering that the two other alternatives are *(i)* being ignorant (or equipped with some handwritten unit tests) or *(ii)* having to (re-)implement the operator in a way that it can be statically analyzed and proved (which is far from being realistic in real-life case studies).

6 Related Work

Model Composition. Model composition was extensively studied by the community, to tame the design of complex models [PB03] or to support the definition of megamodels [SKS+20]. The need to characterize the properties of an operator was already emphasized in a manifest for model merging [BCE+06]. Our work complements this idealized version of a merge operator by providing an empirical way of assessing such properties when static analysis is not available.

Validating model transformations. Research efforts from the modelling community have to lead to test model transformations as a first-class citizen. Some research works aim at building frameworks to efficiently perform tests of model transformations [FSB04], acknowledging the need and difficulty to test such transformations properly. Tracts [GV11] are a way to specify how to test model transformations. A tract focuses on applying the transformation in a particular context of use and comes with a test suite that one can automatically execute [VGB+12]. One can also generate random input models via a dedicated DSL [GBR05]. In any case, these approaches require to implement the composition operator as a model transformation, which is not always possible or relevant.

Testing software composition. Much work has been done toward testing that the *result* of the composition is correct, at the domain-level, and not the composition operator itself, *e.g.*, [SPJF02, TBKC07]. These works addressed domain-specific questions or problems that involve a composition operator but does not address property assessment. Generating relevant sets of test cases is a difficult task. Several works have been done (often using machine learning techniques or meta-models) to generate meaningful test cases, *e.g.*, [Tan19, DAFP+19]. Meaningful and effective generation of test cases are complementary to our contribution.

Metamorphic testing. Metamorphic testing is an approach that alleviates the oracle problem by working on so-called metamorphic relations [CKL+18]. One has to define these relations that hold regardless of the input values (*e.g.*, $\pi - x1 = x2 \rightarrow sin(x_1) = sin(x_2)$). Once built, these relations can be automatically checked on top arbitrarily chosen test cases (*e.g.*, manually or randomly chosen). The majority of the tools used in metamorphic automation are initially meant for PBT. The core idea of PBT is to specify a property that should hold regardless of its input, and automatically find, if possible, the smallest counterexample possible to be shown to the user.

7 Conclusions & Perspectives

In this paper, we described why software composition operators need to be validated before being used, as the relationship between composition code and algebraic properties is not always evident but vital to assess. We described several needs encountered by software developers facing the challenge of working with software composition without being an expert in this field. We also showed according to various examples that this situation is encountered in many different contexts, from aspect weaving to J2E application servers, in both academic and industrial contexts. We proposed a

DSL-based approach to allow software developers to model the elements to be composed and the composition operators associated with these concepts. Based on this model, the compiler generates a framework for the developer, who only has to focus on writing code related to composition. The framework was designed to be lightweight, to support the creation of a brand new operator as well as the integration of an existing one. Then, the very same model is leveraged to generate a tailored benchmark based on algebraic properties defined in a standard library. Using a PBT-based approach, properties are probabilistically validated to provide feedback to the developers, using a reference implementation of PBT in the Java ecosystem.

We identify several interesting perspectives for this work. First of all, as we already stated, the generators are a limitation of the approach that derives from PBT itself. It is tough to identify how to generate relevant instances for a given operator. However, as the DSL was designed to support composition, it might be possible to capture at the language level meta-data that can describe the relevance of using a given operator in a given context. These meta-data can then be exploited to create generators that will use this domain of expert knowledge. Another essential dimension of software composition operators benchmarking from a software developer's point of view is related to performance testing. According to the performance of an operator, it might not be possible to use it in a given context, *e.g.*, operators that take seconds to compose cannot be used in a real-time context. Following the same idea of using meta-data, we will investigate how extra information can be provided at the model level to guide the definition of performance benchmarks. For example, how to simplify the definition of a benchmark that measures the execution time of a given operator according to the size of its input.

References

[ACLF13] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. FAMILIAR: A domain-specific language for large scale management of feature models. *Sci. Comput. Program.*, 78(6):657–681, 2013.

[BCE+06] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A Manifesto for Model Merging. In *International Workshop on Global Integrated Model Management*, page 5–12, New York, NY, USA, 2006. ACM.

[BMCR18] Benjamin Benni, Sébastien Mosser, Philippe Collet, and Michel Riveill. Supporting Micro-services Deployment in a Safer Way: a Static Analysis and Automated Rewriting Approach. In *Symposium on applied Computing*, Pau, France, April 2018.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *(ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[CKL+18] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):4, 2018.

[DAFP+19] Emanuele De Angelis, Fabio Fioravanti, Adrián Palacios, Alberto Pettorossi, and Maurizio Proietti. Property-based test case generators for free. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs*, pages 186–206, Cham, 2019. Springer International Publishing.

[FHMC18] François Fouquet, Thomas Hartmann, Sébastien Mosser, and Maxime Cordy. Enabling lock-free concurrent workers over temporal graphs composed of multiple time-series. In *International Symposium on Applied Computing*, SAC, April 2018.

[FSB04] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in model-driven engineering: testing model transformations. In *International Workshop on Model, Design and Validation, 2004.*, pages 29–40, November 2004.

[GBR05] Martin Gogolla, Jørn Bohling, and Mark Richters. Validating UML and OCL models in use by automatic snapshot generation. *Software & Systems Modeling*, 4(4):386–398, November 2005.

[GV11] Martin Gogolla and Antonio Vallecillo. Tractable Model Transformation Testing. In *Modelling Foundations and Applications*, pages 221–235, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, September 2001.

[MB13] Sébastien Mosser and Mireille Blay-Fornarino. ADORE, a logical meta-model supporting business process evolution. *Sci. Comput. Program.*, 78(8):1035–1054, 2013.

[PB03] Rachel Pottinger and Philip A. Bernstein. Merging models based on given correspondences. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 826–873. Morgan Kaufmann, 2003. URL: `http://www.vldb.org/conf/2003/papers/S26P01.pdf`, `doi:10.1016/B978-012722442-8/50081-1`.

[PLHM08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. *SIGOPS Oper. Syst. Rev.*, 42(4):247–260, April 2008. `doi:10.1145/1357010.1352618`.

[PMP+15] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46:1155–1179, 2015. `doi:10.1002/spe.2346`.

[SKS+20] Rick Salay, Sahar Kokaly, Alessio Di Sandro, Nick L. S. Fung, and Marsha Chechik. Heterogeneous megamodel management using collection operators. *Software and Systems Modeling*, 19(1):231–260, 2020. URL: `https://doi.org/10.1007/s10270-019-00738-9`, `doi:10.1007/s10270-019-00738-9`.

[SPJF02] Andreas Speck, Elke Pulvermuller, Michael Jerger, and Bogdan Franczyk. Component composition validation. *International Journal of Applied Mathematics and Computer Science*, 12:581–589, 2002.

[Tan19] C. Tan. A model-based approach to generate dynamic synthetic test data. In *2019 12th IEEE Conf. on Software Testing, Validation and Verification*, pages 495–497, April 2019. `doi:10.1109/ICST.2019.00063`.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering - GPCE07*, GPCE '07, pages 95–104, New York, NY, USA, 2007. ACM Press. doi:10.1145/1289971.1289989.

[TOJ02] Peri L. Tarr, Harold Ossher, and Stanley M. Sutton Jr. Hyper/j: multi-dimensional separation of concerns for java. In Will Tracz, Michal Young, and Jeff Magee, editors, *ICSE 2002, Orlando, Florida, USA*, pages 689–690. ACM, 2002. doi:10.1145/581339.581447.

[VGB+12] Antonio Vallecillo, Martin Gogolla, Loli Burgueño, Manuel Wimmer, and Lars Hamann. *Formal Specification and Testing of Model Transformations*, pages 399–437. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-30982-3_11.

[ZDD06] Alanna Zito, Zinovy Diskin, and Juergen Dingel. Package merge in UML 2: Practice vs. theory? In *International Conference on Model Driven Engineering Languages and Systems*, pages 185–199. Springer, 2006.

About the authors

Benjamin Benni is a PhD student at *Université Côte d'Azur*, working on software composition mechanisms. In his PhD thesis, he applied software composition to various software ecosystems, such as the Docker container system, or the Spoon code rewriting engine. Contact him at benni@i3s.unice.fr, or visit https://www.i3s.unice.fr/~benni/

Sébastien Mosser is *Professeur* of Software Engineering at *Université du Québec à Montréal*, where he leads a research group working on scalable software composition. Since 2007, he applied software composition to various domain such as business processes, databases and IoT. Contact him at mosser.sebastien@uqam.ca, or visit https://ace-design.github.io

Mathieu Acher is Associate Professor at University of Rennes 1 and a member of the DiverSE team (Inria/IRISA). His research interets are related to any form of variability (in software, in videos, in 3D printing, in data, etc.). He is also the lead architect and maintainer of the Familiar language. Contact him at mathieu.acher@irisa.fr, or visit https://www.mathieuacher.com.

Mathieu Paillart is a graduate student (M.Sc.) in Software Architecture at *Université Côte d'Azur*. He developed the initial prototype and associated tooling of the DSL described in this paper during a research internship at UQAM.
Contact him at mathieu.paillart@etu.univ-cotedazur.fr.