Article:
Garmendia, Antonio, Guerra, Esther, de Lara, Juan et al. (2 more authors) (2019) Scaling-up domain-specific modelling languages through modularity services. Information and Software Technology. pp. 97-118. ISSN 0950-5849

https://doi.org/10.1016/j.infsof.2019.05.010

Scaling-up Domain-Specific Modelling Languages through Modularity Services

Antonio Garmendia, Esther Guerra, Juan de Lara

Universidad Autónoma de Madrid (Spain)

Antonio García-Domínguez

Aston University (UK)

Dimitris Kolovos

The University of York (UK)

Abstract

Context. Model-driven engineering (MDE) promotes the active use of models in all phases of software development. Even though models are at a high level of abstraction, large or complex systems still require building monolithic models that prove to be too big for their processing by existing tools, and too difficult to comprehend by users. While modularization techniques are well-known in programming languages, they are not the norm in MDE.
Objective. Our goal is to ease the modularization of models to allow their efficient processing by tools and facilitate their management by users.
Method. We propose five patterns that can be used to extend a modelling language with services related to modularization and scalability. Specifically, the patterns allow defining model fragmentation strategies, scoping and visibility rules, model indexing services, and scoped constraints. Once the patterns have been applied to the meta-model of a modelling language, we synthesize a customized modelling environment enriched with the defined services, which become applicable to both existing monolithic legacy models and new models.
Results. Our proposal is supported by a tool called EMF-Splitter, combined with the Hawk model indexer. Our experiments show that this tool improves the validation performance of large models. Moreover, the analysis of 224 meta-models from OMG standards, and a public repository with more than 300 meta-models, demonstrates the applicability of our patterns in practice.
Conclusions. Modularity mechanisms typically employed in programming IDEs can be successfully transferred to MDE, leading to more scalable and structured domain-specific modelling languages and environments.

Keywords: Model-Driven Engineering, Meta-Modelling, Scalability, Domain-Specific Modelling Languages

1. Introduction

Model-driven engineering (MDE) promotes models as the central assets in software development [1, 2]. Models are described either using general-purpose modelling languages, like the UML, or with domain-specific modelling languages (DSMLs) tailored to an area of interest [3]. In both cases, models abstract away accidental details of the system being built, focusing on its essential features. However, models of large complex systems may become too big and hence difficult to understand by humans and to process by tools [4].

For decades, the programming languages community has developed modularity mechanisms to cope with large programs [5]. These include the possibility to fragment programs into language units (e.g., classes or interfaces) which are stored in different files and organized in modules (e.g., packages in Java). Many programming languages also support scoping and visibility rules controlling which elements are visible and can be accessed from other modules (e.g., through access level modifiers). Other techniques that contribute to the efficient management of large programs focus on ways to avoid recompiling a whole program upon localized changes, e.g., by incremental type-checking, compilation and linking mechanisms [6, 7].

In contrast, DSMLs often lack modularity mechanisms, and models expressed with them are frequently monolithic and reside in a single file. This is so as many language development frameworks do not provide intuitive means to enhance the definition of DSMLs with modularity facilities, like fragmentation strategies for models, or incremental validation of constraints within a given scope. Although some environments provide ad-hoc modularization services for specific modelling languages (e.g., UML) [8, 9, 10, 11, 12, 13], these are not readily available to developers of new DSMLs. Instead, developers need to program the required services manually for the platform where the DSML environment is being built. Since this is a complex task that requires expert knowledge, most DSMLs end up lacking these features, which hinders the scalability of modelling in practice.

To improve the scalability of domain-specific modelling in MDE, we propose generic modularity services comparable to those existing for programming languages. This is in general challenging as programming languages come with explicitly defined semantics that drive the abstraction mechanisms, but metamodels often have an implicit semantics. Hence, our proposal covers a subset of modularity concepts that seem useful to many languages (information hiding and compilation unit) and do not necessarily rely on an explicit specification of the meta-model semantics.

Our approach is based on extending the meta-model of the DSML with information about the modular organization of models, scope of references, object visibility rules, attribute indexing, and validation scope of DSML constraints. Model modularization permits describing strategies to fragment models into packages and different kinds of units, which are stored in separate files. As a model is fragmented into smaller parts, reference scoping and object visibility rules permit customising how objects can be accessed and cross-linked across

units. To improve efficiency, constraints can define a scope and be validated incrementally, while attributes and references can be indexed for more efficient queries [14].

We have realized our proposal in a tool called EMF-Splitter [15]. This describes the different modularization services as patterns which are instantiated on top of the DSML meta-model, and from which a scalable modelling environment with the defined services is synthesized. To improve efficiency, the generated environment makes use of Hawk [14], a model indexer supporting efficient querying across fragmented resources.

We have evaluated the benefits of our proposal by means of three experiments. The first one demonstrates the applicability of our approach, based on the analysis of a public repository of more than 300 meta-models, and 224 meta-models from specifications of the Object Management Group (OMG). The second and third experiments focus on performance. In particular, the second experiment evaluates the efficiency gains of scoped constraint validation on a fragmented model, with respect to standard constraint validation on monolithic models. The third one is a case study where our tooling shows performance gains over existing tools in the industrial automation domain [16].

This paper continues our work on facilitating the construction of scalable modelling environments [15, 17] by presenting four new services: reference scoping, object visibility, indexing, and scoped constraints. Moreover, we show the integration of EMF-Splitter with indexing services [14], and evaluate the efficiency gains and applicability of our approach with three experiments.

The rest of the paper is organized as follows. Section 2 motivates the need for modularity services for DSMLs based on a running example. Section 3 describes our approach, which is based on patterns and code generation. Section 4 introduces our modularization patterns for fragmentation, reference scoping, visibility, indexing, and scoped constraints. Section 5 presents tool support, and Section 6 reports on our evaluation. Section 7 compares with related research, and finally, Section 8 ends with the conclusions and directions for future work.

2. Motivation and running example

In this section, we introduce a motivating running example from which we elicit a number of requirements for scalable modelling environments.

As an example, we will be using an architectural language inspired by the industrial modelling case on wind turbine control system development presented in [18]. Figure 1 shows the meta-model of the language. It allows describing two aspects of wind turbine control systems: (i) the constituent types of system components and how they can be connected (classes Component, Port, InPort, OutPort and Connector); and (ii) the admissible states and state changes of those component types (classes StateMachine, DocumentElt, Vertex, InitialState, SimpleState and Edge). In addition, the meta-model provides classes to organize components into hierarchies of subsystems (class Subsystem), as well as to group the state machines in each subsystem (class ControlSubsystem).

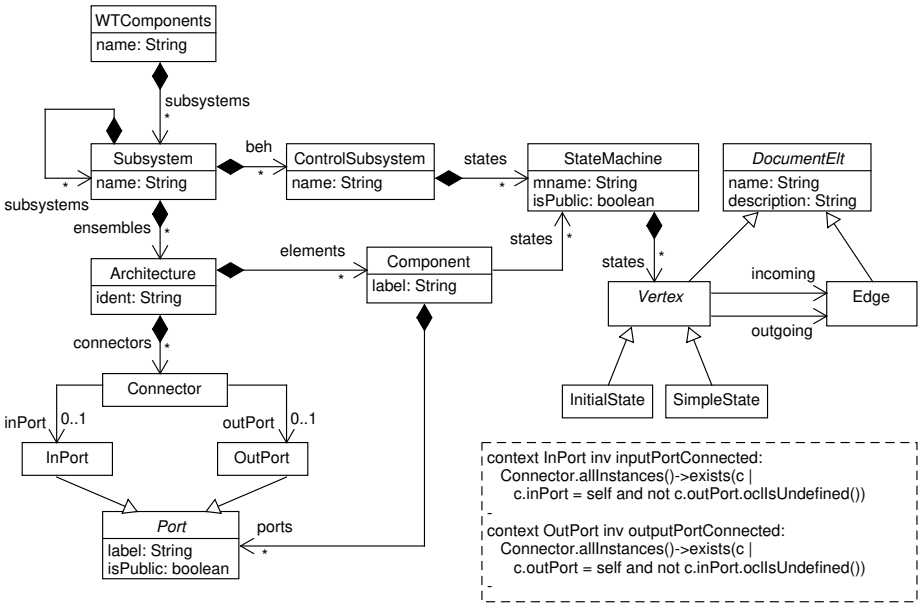


Figure 1: Meta-model of an architectural language for wind turbine control systems.

To be able to define models using the architectural language, we would like to have a customised environment with typical modelling facilities like model editing, conformance checking, model search, etc. Since we expect control systems to consist of many components, the environment should be optimized to deal with large models from the tool perspective (performance) and the user perspective (usability). Building this environment by hand is possible but costly. Instead, using a meta-modelling-based language development framework for its construction is faster.

Examples of graphical and textual language development frameworks include GMF [19], Sirius [20] and Xtext [21]. However, these frameworks typically yield environments for editing monolithic models, i.e., models with all their elements included in the same file/resource. As a consequence, these environments have performance problems when managing big models. Moreover, having monolithic models is not optimal in our example, as the language clearly identifies two different concerns (structure and behaviour) and so a mechanism that enables their separation is desirable [22]. Additionally, the language provides primitives (nested subsystems, control subsystems) that may be used to organize the model content in packages according to the subsystems structure. Even though modelling frameworks like EMF [23] permit cross-referencing elements across files, they lack a native way to define and enforce fragmentation policies, or to organize a model into packages (the latter is available for meta-models but not for models). While some language frameworks like Sirius [20] support the definition of diagram types, they do not provide mechanisms to combine several

model fragments into a unified model, or to map parts of the model structure to the file system (like packages in Java). Moreover, the fragmentation strategy should be intrinsic to the abstract syntax, not tied to specific concrete syntaxes.

Once a model is fragmented, it is desirable to control which elements can be cross-referenced from other model fragments. For example, we may wish to restrict a Component to reference only those StateMachines located in ControlSubsystems within the same Subsystem the Component belongs to. While it is possible to define a constraint that checks this using any constraint language, an advanced modelling environment would also filter out all StateMachine objects that are out of the scope. Frameworks like Xtext [21] support scopes, but these are tied to the concrete syntax, are normally defined in low-level programming languages, and require deep knowledge of the framework.

Meta-models may include integrity constraints, typically specified using the Object Constraint Language (OCL) [24], to be satisfied by models. As an example, Figure 1 shows in the bottom-right corner two constraints demanding that every input port is connected to some output port, and vice versa. Constraints are defined in the context of a class, and evaluated on every instance of the class that is contained in a model, which is time-consuming for big models. Instead, we may take advantage of the fragmentation of models to scope the evaluation of constraints to smaller fragment units. In the running example, this means that whenever a component is changed, only the ports in that component should get their constraints re-evaluated, but not the rest of the ports.

Finally, searches on big models can be slow. One way to tackle this problem is the use of model indexers and indices of relevant attributes to speed up the search [14]. However, since building an index incurs a time overhead, it should be possible to customize the subset of attributes to be indexed (only those used in recurrent searches).

Altogether, the modelling environment for the proposed architectural language has to fulfil the following requirements, which indeed are general requirements of scalable modelling environments:

R1 Ability to define fragmentation strategies for models, to enforce the separation of concerns.

R2 Ability to organize the model content hierarchically into packages, to improve usability.

R3 Ability to control the visibility of elements across fragments, and the scope of cross-references, to manage complexity through information hiding.

R4 Ability to customize the scope of integrity constraints, to improve their validation performance.

R5 Ability to define model indices, to improve the performance of recurrent searches.

The next section presents our approach to the construction of modelling environments for DSMLs, where the previous services can be customized.

3. A pattern-based approach to modularity in DSMLs

Figure 2 shows a scheme of our approach to the development of modelling environments with support for modularity services. The approach provides a catalogue of modularity services expressed as meta-model patterns [25], which can be applied to the domain meta-model for which the environment is being developed (label 1). The modelling language designer can apply as many patterns as required to the domain meta-model. Each pattern application produces a customization of the corresponding service for the domain meta-model. Section 4 will present our catalogue of modularity patterns, which covers services for model fragmentation, object visibility, reference scoping, attribute indexing and scoped validation.

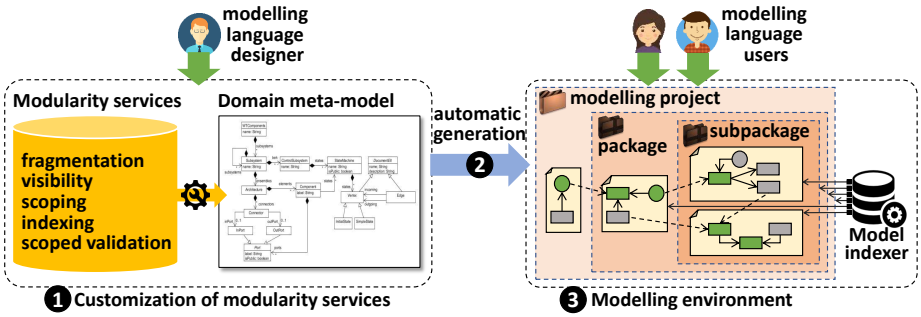


Figure 2: Overview of our approach to build modelling environments with modularity services.

Technically, our modularity patterns have the form of a meta-model, and their elements (classes, references and attributes) are called *roles* [26]. Each role defines a cardinality interval which governs how many times the role can occur in a pattern application. If a role does not define a cardinality explicitly, then it is assumed to be of cardinality [1..1]. Class roles can be tagged with the stereotype abstract, in which case the class role cannot be instantiated but is a placeholder for attribute or reference roles that get inherited by children class roles. Since roles tagged as abstract cannot be instantiated, they do not have any cardinality.

Class roles may have two kinds of fields[1]: *field roles* and *configuration fields*. Field roles must be mapped to fields with a compatible type in the domain meta-model, while their name can be different. Just like class roles, field roles define a cardinality range ([1..1] by default). On the other hand, configuration fields are not mapped but need to receive a value when the pattern is applied. Inspired by deep characterization in multi-level modelling [27, 28], we tag the configuration fields with "@1" (potency 1) as they receive a value when the pattern is instantiated one meta-level below, while the field roles have potency

[1] We uniformly refer to attributes and references as fields.

2 as they receive a value two meta-levels below. We omit the inscription "@2" in the field roles for readability reasons.

As an example, the upper-left corner of Figure 3(a) shows a pattern definition. We use a synthetic example to better illustrate all features of our patterns and their application, and refer to Section 4 for the catalogue of proposed patterns. The pattern has two class roles, one reference role, one attribute role, and one configuration attribute (marked with "@1"). RoleA and roleatt do not specify a cardinality, hence they are assumed to have [1..1]; RoleB and roleref define cardinality [1..∗]. Note that field roles have two cardinalities: a role cardinality which governs the number of instances of the role ([1..∗] for roleref), and a field cardinality which must be compatible with the field mapped in the domain meta-model (∗ for roleref).

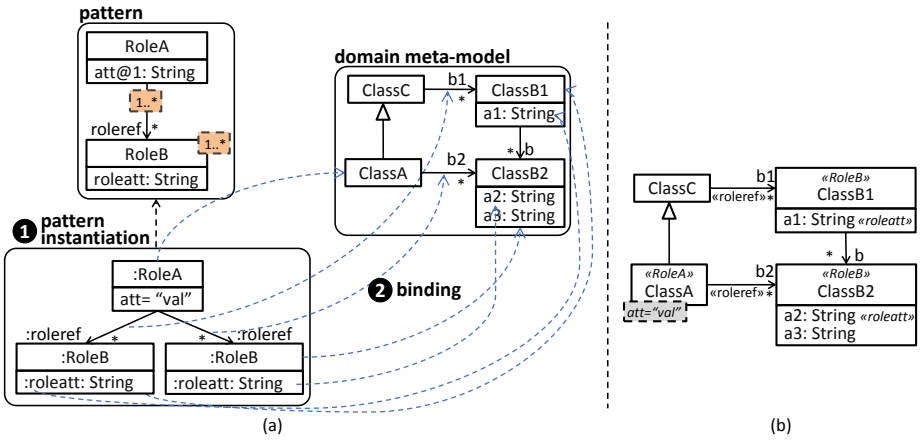


Figure 3: (a) Example of pattern application to domain meta-model. (b) Visualization of applied pattern.

The application of a pattern to a domain meta-model proceeds in two steps. First, the language designer instantiates the pattern as a regular meta-model, respecting its role cardinalities. Then, he/she needs to bind the elements in the pattern instance to elements in the domain meta-model (class roles to domain classes, attribute roles to domain attributes, and reference roles to domain references). This binding allows structural matching, i.e., if a class role r is mapped to a domain class c, then the field roles inside r must be bound to fields owned or inherited by c. In addition, some patterns may define extra conditions expressed in OCL or Java to restrict the bindings considered correct [25].

Figure 3(a) exemplifies the application of a pattern (in the upper-left corner) to a meta-model (in the upper-right corner). As a first step, the pattern is instantiated. In the example, the created pattern instance contains one instance of RoleA, two instances of RoleB, and the configuration attribute att receives the value "val". In a second step, the pattern instance elements are bound to elements of the meta-model. In Figure 3(a), the binding (depicted as dotted arrows) maps :RoleA to ClassA; one :RoleB to ClassB1 and the other to ClassB2;

one :roleatt to a1 and the other to a2; and the two :roleref to two references of ClassA, one owned and the other inherited. In general, one element in the domain meta-model is allowed to receive several roles. For example, if ClassA had a self-reference r and a String attribute a, then both :RoleA and :RoleB could be mapped to ClassA, with :roleref mapped to r, and :roleatt to a.

Figure 3(b) visualizes the domain meta-model with the applied pattern. Roles are depicted as stereotypes on the mapped class or field, and configuration fields (like att) are shown in a box attached to the stereotyped class.

The language designer can apply as many patterns as desired to the domain meta-model, following the described process. From this definition, a modelling environment that integrates modularity services configured in compliance with the instantiated patterns is automatically synthesized (labels 2 and 3 in Figure 2). This environment features model indexers [29] that improve the efficiency of searches across model fragments and manage inconsistent cross-references when fragments are moved to a different location [30]. Before detailing our tooling in Section 5, the next section describes the supported modularity patterns.

4. Catalogue of modularity patterns and services

In this section, we present our catalogue of modularity services and their associated patterns, which are illustrated using the running example.

4.1. Fragmentation

Programming languages offer techniques for dividing a program into smaller building blocks. This helps developers to tame the system complexity, and increments the efficiency of the tooling to perform tasks such as syntax and type checking, compilation and linking.

Similarly, to scale modelling to larger systems, we propose transferring the concept of fragmentation to DSMLs. This provides benefits in terms of model navigability and processing efficiency (see Section 6.2). Moreover, teams of engineers working collaboratively on a fragmented model will potentially have fewer conflicts in version control systems than when working on a monolithic model.

To support model fragmentation, we take inspiration from the modularity concepts of languages like Java and its JDT [31] Eclipse programming environment. Eclipse JDT allows creating Java *projects*, and the language permits breaking programs physically into compilation *units* (classes, interfaces and enumerations residing in different files) which are organised into hierarchies of *packages*. Projects, folders and files are located in the *workspace* organized into a tree structure with projects at the top, and folders and files underneath.

We have adapted these ideas to DSMLs by defining the *fragmentation* pattern shown in Figure 4(a). The pattern defines Project, Package and Unit as class roles. Designers can configure which classes of a DSML will play those roles. This way, the instances of the DSML will not be monolithic, but they will be structured into projects, packages and units according to the given strategy.

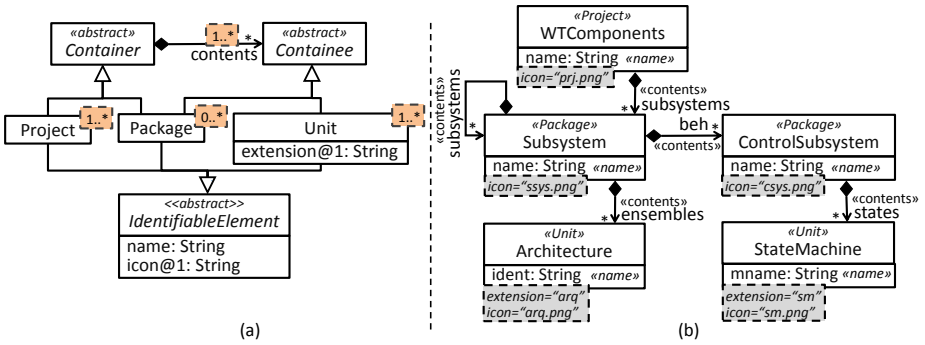


Figure 4: (a) Fragmentation pattern. (b) Applying the fragmentation pattern to the running example.

Typically, the root class that contains directly or indirectly all other classes of a DSML should be mapped to Project. The pattern declares a condition (omitted in the figure) checking that all classes bound to Project are root. As a result, each time the root class is instantiated in the generated environment, a new modelling project (i.e., a folder that will hold all fragments of a model) is produced. To account for meta-models that have several root classes, role Project has cardinality $[1..*]$. Similarly, when a class with role Package is instantiated, the environment creates a folder in the file system, together with a hidden file storing the value of the class attributes and non-containment references. Finally, instantiating a class c with Unit role results in the creation of a file that holds instances of the classes that can be directly or indirectly reached from c by means of containment relations.

In the pattern, all concrete class roles inherit the attribute role name to be used as the project, folder or file name, and the configuration attribute icon to specify the icon file to be used as a decorator in the generated environment. Units can also indicate a file extension using the configuration attribute extension. Containers (i.e., Projects and Packages) and Containees (i.e., Packages and Units) are related through the reference role contents. This must be mapped to a containment reference in the DSML meta-model, modelling the inclusion of files in folders, and these in projects. Extending our pattern to support fragmentation along non-containment relations is future work.

As previously mentioned, classes in the domain meta-model are allowed to play several roles. For example, a class C can be both Package and Unit. In such a case, upon creating a C object, the user decides whether a package or a file should be created. Similarly, a class can be both Project and Package (which is implicit for Project classes that have self-containment references); as well as both Project and Unit (to allow the creation of monolithic models, if desired).

Example. Figure 4(b) depicts the application of the fragmentation pattern to the meta-model of Figure 1. The role Project is assigned to the class WTComponents. Two types of packages are defined: Subsystem, which can recursively

contain other Subsystem packages, and ControlSubsystem. There are also two unit types: Architecture and StateMachine. These classes will be physically persisted as files with extensions "arq" and "sm" respectively, and may store objects of the classes contained in them. For instance, a unit of type Architecture can hold objects of types Architecture, Component, Connector, InPort and OutPort (see Figure 1). As an example, Figure 5(c) shows a model organised according to the defined fragmentation strategy.

4.2. Reference scoping

When a model is monolithic, its objects can refer to other objects within the same file, according to the reference types defined in the model's metamodel. However, when a model is fragmented, some references may need to cross fragment boundaries. In this context, there is the need to control the fragments a reference can reach, that is, its *scope*. This control mechanism is useful to manage the access modifiers for the classes of a given DSML, and to reduce the set of potential candidate objects for a given reference.

We allow customizing this information using the *scoping* pattern in Figure 5(a). This declares a single class role ScopedClass, which should be mapped to the domain class owning or inheriting the reference to be scoped. In its turn, this reference should be mapped to one of the five reference roles in the pattern, which represent five different scoping policies. The least restrictive policy is sameWS, which allows a reference to refer to objects in the same workspace, i.e., anywhere in any project. This is the default option for references with no scope. The sameProject policy permits referring to objects within the same project. The policies samePkg and sameRootPkg allow referencing objects within the same package, or that have the same root package, respectively. Finally, sameUnit restricts a reference to the objects residing in the same file. In any of the cases, it is possible to further constrain the scope of the reference by providing an OCL expression to filter out additional unwanted objects.

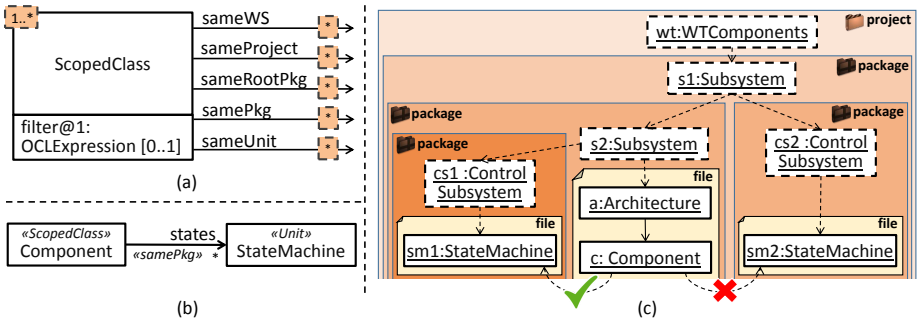


Figure 5: (a) Scoping pattern. (b) Applying *same package* scope to reference Component.states. (c) Effect of pattern application on a fragmented model.

It can be noted that our notion of pattern supports reference roles with no target class role, so that there is no need to map the latter to any domain class.

Moreover, the reference roles in the scoping pattern can be mapped multiple times to meta-model references, but they do not impose any cardinality to those references (i.e., they do not specify any reference cardinality).

Example. Figure 5(b) assigns the *same package* scope to the reference Component.states, and Figure 5(c) shows the effect of this scope on a fragmented model. The scope allows connecting the Component object c to any StateMachine located in the container package of c (i.e., in the Subsystem s2). Therefore, c can refer to sm1 as this is (indirectly) contained in s2, but not to sm2 because it is in a different package. Changing the reference scope to *same root package* would allow connecting c to sm2.

4.3. Visibility

Programming languages like Java allow classes to control whether other classes can use a particular field or method by means of access level modifiers. Similarly, we allow model fragments to define an interface to expose only a subset of its elements to other fragments, while hiding the rest.

By default, objects are accessible from any other fragment, but this can be constrained by using the pattern in Figure 6(a). This pattern allows defining whether the instances of a class are visible to other fragments in the same workspace, project, root package or package. An object is always visible within its file. The visibility is configured by means of an OCL expression which is evaluated on every object of the class, and only the objects satisfying the expression become visible to other fragments in the given scope.

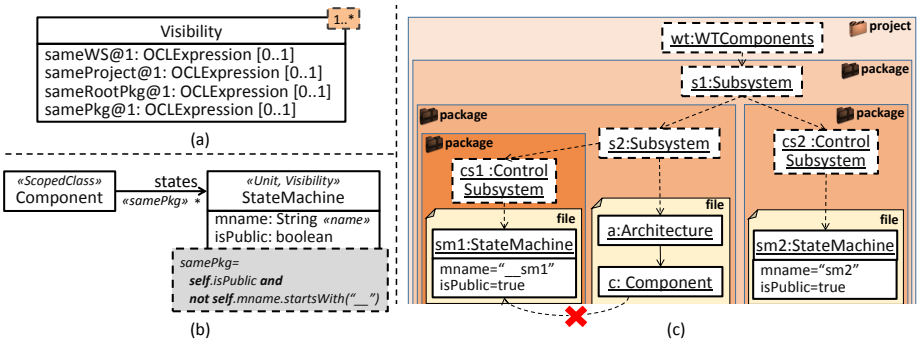


Figure 6: (a) Visibility pattern. (b) Applying *same package* visibility to class StateMachine. (c) Effect of pattern application on a fragmented model.

A class can define visibility conditions for different scopes, e.g., *same project* and *same package*. In such a case, the more general scope is selected. For example, if the visibility conditions for both scopes *same package* and *same project* evaluate to true, then the object is visible at the project level; otherwise, the object is only visible at the level of the expression that evaluates to true.

The visibility and scoping patterns are complementary. While scoping restricts the content of a reference, visibility restricts the usage context of an

object. If a certain class of objects should be visible to all fragments in its package or project, then specifying class visibility is equivalent to specifying the corresponding scope for every cross-reference pointing to the class; however, in this case, the first option is less costly as it is done once for the class instead of once per reference.

Example. Figure 6(b) applies the visibility pattern to class StateMachine. The OCL condition, which is defined for samePkg, appears in a grey box. In this way, StateMachine objects will only be visible from other fragments in the same package when they are public and their name does not start by a double underscore. Figure 6(c) shows the effect of this pattern on a fragmented model. In the figure, the StateMachine sm1 is not visible from other fragments in the same package because its name starts by a double underscore; therefore, it cannot be referenced from the Component c, even if sm1 is in the reference scope. On the other hand, sm2 is not visible to Component c because the visibility level is samePkg and sm2 is in a different package. Moreover, sm2 is not reachable from c because the reference states has scope samePkg.

4.4. Indexing

To speed up the computation of common queries over models, we support the creation of indices of objects by selected fields, backed by a model indexer [14]. The pattern in Figure 7(a) allows selecting the fields to index. It is made of the class role IndexedClass and the field role index, both with role cardinality [1..∗]. This way, for each domain class mapped to IndexedClass, it is possible to specify one or more fields (attributes or references) to be used as indices. The field role defines the data type Any and no field cardinality; this means that the indexed fields in the domain meta-model can have any type and cardinality.

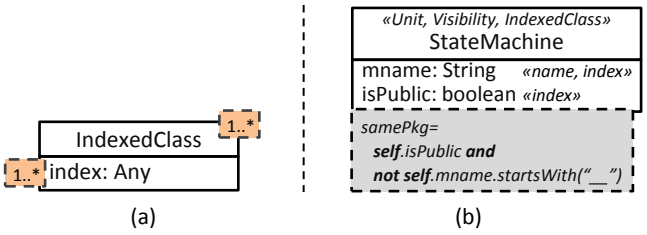


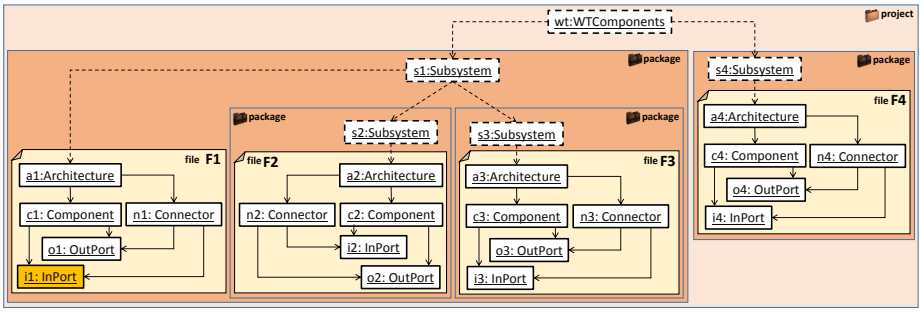
Figure 7: (a) Indexing pattern. (b) Applying pattern to class StateMachine.

Example. Figure 7(b) applies the indexing pattern to select the attributes mname and isPublic as indices for class StateMachine. The motivation to index these attributes is their use in the expression samePkg specified in a previous application of the visibility pattern. This index permits improving the retrieval of StateMachine objects [14].

4.5. Scoped validation

Meta-models may define integrity constraints that are evaluated on monolithic models, have access to all objects within the model, and need to be re-evaluated upon any model change. However, once a model is fragmented, it is natural to incorporate the notion of scope into the integrity constraints, so that each constraint only accesses the objects within the defined scope, and is re-evaluated just when there is a change within the scope. We call an integrity constraint that defines a validation scope a *scoped constraint*. As we will show in Section 6.2, scoped constraints can be evaluated more efficiently than standard constraints, as they consider a subset of model objects instead of the whole model, and the need of re-evaluation is less frequent. Moreover, scoped constraints decouple the constraint from the objects affected by it, while standard constraints need to explicitly select the affected objects by means of filters in the constraint expression.

As in the previous patterns, we consider five scopes for constraints: *same unit*, *same package*, *same root package*, *same project* and *same workspace*. To illustrate their implications, consider the fragmented model in Figure 8 and the constraint inputPortConnected below the model (also shown in Figure 1).




```
context InPort inv inputPortConnected:
  Connector.allInstances()→exists(c | c.inPort = self and not c.outPort.oclIsUndefined())
```

Figure 8: A fragmented model where a scoped constraint is to be evaluated on the InPort i1.

Depending on the validation scope assigned to the constraint, expression Connector.allInstances() returns a different set of objects when it is evaluated on object i1 of file F1:

- if the scope is *same unit*, the result is Set{n1} because n1 is the only Connector object within the same file as i1, even though other Connector objects exist in the whole model.
- if the scope is *same package*, the result is Set{n1, n2, n3} because the three objects are contained in the same package as i1 (s1) directly or indirectly. Instead, evaluating the expression on i2 returns Set{n2}, as this is the only Connector within i2's container package s2.

- if the scope is *same root package*, the result is Set{n1, n2, n3} as the root package of i1 is s1. Evaluating the expression on i2 yields the same result.
- if the scope is *same project*, the expression returns Set{n1, n2, n3, n4} independently of the object where it is evaluated. If this is the only project in the workspace, we obtain the same set of Connectors when we assign the scope *same workspace* to the constraint.

Listing 1 shows the algorithm to validate scoped constraints. It considers two scenarios: *(i)* the validation of all constraints within a project, which is useful, e.g., when a project is loaded in the workspace; *(ii)* the re-evaluation of constraints upon model changes, where we efficiently restrict the re-evaluation to the subset of objects where the constraint value may have changed. We refer to scenario *(i)* as *full validation*, and to scenario *(ii)* as *incremental validation*.

The entry point of the algorithm is the scopedValidation method in lines 1–23. This receives as parameters the resource r that has changed (a unit, a package or a project), and a boolean flag full (true to execute scenario *(i)*, and false for scenario *(ii)*). For each scoped constraint in the meta-model, the method first obtains its validation context, i.e., the instances of the constrained class where the constraint needs to be evaluated (lines 5–14). In case of scenario *(i)*, the constraint needs to be evaluated on all instances of the constrained class defined in the project (line 7); in case of scenario *(ii)*, the instances are retrieved from the scope specified by the constraint (lines 8–14), and this scope is computed by the methods in lines 25–56. As an example, the method package is invoked when the constraint defines the scope *same package*, and it returns the received resource if it is a package, or its container if it is a unit (lines 39–42). Before the obtained resource can be used, we make sure that all other resources that it contains directly or indirectly are loaded as well (line 32). We use a cache to avoid reloading previously loaded resources. The following steps in the algorithm are the same for both scenarios. Specifically, for each object in the validation context, the algorithm obtains its validation scope (line 18). This is calculated similarly to the validation context, but considering the resource that contains the object, instead of the modified resource. Then, the algorithm validates the constraint over each object of the validation context considering only the validation scope (line 22), and reports an error if the validation returns false (line 23).

Please note that given a constraint c with scope samePkg, the algorithm avoids validating c recursively on each container package until reaching the root package. Instead, it directly jumps to the root package (line 11), and then considers the validationScope of each context object when evaluating c (line 18).

For efficiency reasons, we do not maintain all model fragments in memory, but we load them on demand. Therefore, our algorithm needs to load and merge any model fragment that is necessary to perform the validation. Line 32 in Listing 1 takes care of this. In practice, we rely on EMF proxies and their lazy loading mechanism to achieve this behavior. This way, fragments can refer to elements in other fragments by means of proxies, and only when there is the

```
1  def scopedValidation (r : Resource, full : boolean)
2      for each scoped constraint c:
3
4          -- obtain objects over which the constraint will be evaluated (context)
5          var validationContext = nil
6          if full = true and type(r) = Project then
7              validationContext = objects of type c.constrainedClass within r
8          else if c.scope = samePkg then
9              -- jump to root pkg to avoid validating c in each intermediate container pkg
10             validationContext = objects of type c.constrainedClass
11                                 within resource4scope (r, sameRootPkg)
12         else
13             validationContext = objects of type c.constrainedClass
14                                 within resource4scope (r, c.scope)
15
16         for each object o in validationContext:
17             -- obtain model fragment where the OCL expression will be evaluated (scope)
18             var validationScope = resource4scope (o.resource, c.scope)
19
20             -- validate constraint and report detected errors
21             if validationScope <> nil then
22                 var result = o.validate (c.statement, validationScope)
23                 if result = false then o.report (c.error)
24
25 def resource4scope (r : Resource, s : Scope) : Resource
26     var r4s = nil
27     if s = sameUnit then r4s = unit(r)
28     else if s = samePkg then r4s = package(r)
29     else if s = sameRootPkg then r4s = rootPackage(r)
30     else if s = sameProject then r4s = project(r)
31     else if s = sameWS then r4s = workspace(r)
32     loadResource(r4s)
33     return r4s
34
35 def unit (r : Resource) : Resource
36     if type(r) = Unit then return r
37     return nil
38
39 def package (r : Resource) : Resource
40     if type(r) = Package then return r
41     if type(r.container) = Package then return r.container
42     return nil
43
44 def rootPackage (r : Resource) : Resource
45     var root = r
46     while type(root.container) = Package: root = root.container
47     if type(root) = Package then return root
48     return nil
49
50 def project (r : Resource) : Resource
51     var root = r
52     while type(root) <> Project and root <> nil: root = root.container
53     return root
54
55 def workspace (r : Resource) : Resource
56     return $WORKSPACE
```

Listing 1: Algorithm for the validation of scoped constraints.

need to access to these other fragments, we load them and resolve the proxies. This can be seen as a kind of model merge.

This algorithm performs an efficient re-evaluation of scoped constraints upon model changes, as constraints are evaluated only over the objects in the validation context, which are those objects that may have been affected by the change. This kind of incrementality is coarse-grained, in the sense that it only considers the resources that have changed to identify the constraints to re-evaluate, but not the actual model changes. Other approaches like [32, 33, 34] keep track of those changes as well. This permits reducing further the set of constraints to re-evaluate, at the cost of having to memoise the objects/fields accessed during the last evaluation of each constraint.

We support the definition of scoped constraints by means of the *scoped validation* pattern in Figure 9(a). A scoped constraint is defined by mapping a domain class (the context of the constraint) to the class role ConstrainedClass. Then, the configuration fields in the pattern allow defining the constraint name (name), the OCL expression (statement), the evaluation scope (scope), and the error message to be reported when the objects of the constrained class violate the constraint (error).

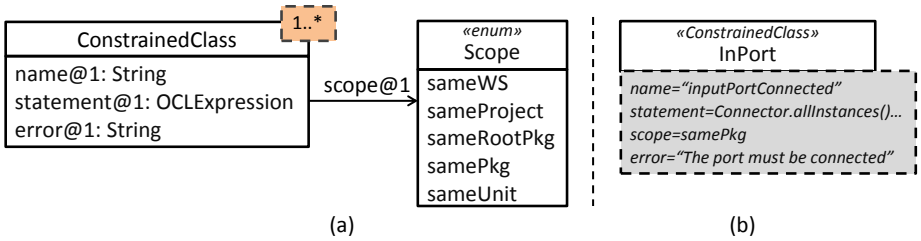


Figure 9: (a) Scoped validation pattern. (b) Defining a scoped constraint for class InPort.

Example. Figure 9(b) assigns the scope samePkg to the constraint in Figure 8. Now, assume file F1 in Figure 8 changes. Then, according to the algorithm in Listing 1, the validationContext is the package where F1 resides (i.e., s1). Hence, the constraint is evaluated in all InPort objects within s1 (i.e., {i1, i2, i3}) using the scope of the constraint (*same package*) as validationScope. For i1, the validation scope is the package s1 where i1 is located. Hence, Connector.allInstances() yields {n1, n2, n3}. In case of i2, the constraint is evaluated on the package that contains i2 (i.e., s2), where the only Connector is n2. In case of i3, it is evaluated on its container package s3, where the only Connector is n3. With this scoped validation strategy, we do not need to evaluate the constraint on i4 or consider Connector n4. If the change happened in file F4, then the constraint would be evaluated on i4 considering only the Connector c4.

5. Architecture and tool support

We have realized our approach as an Eclipse-based solution [35]. Figure 10 shows its architecture. It relies on the Eclipse Modelling Framework (EMF) [23] to represent the domain meta-models and their instances. It contains a modelling front-end called DSL-tao [25] which permits creating domain meta-models graphically and offers a repository of patterns with services to configure the functionality of the generated modelling environment. DSL-tao patterns describe the pattern structure by means of a model, and can contribute services for the modelling environment through an extension point called Pattern Implementation. The pattern repository includes all modularity patterns presented in Section 4, and EMF-Splitter is the tool implementing the extension points in charge of generating the modelling environment once the patterns have been applied. While EMF-Splitter complements DSL-tao, it can also be used stand-alone.

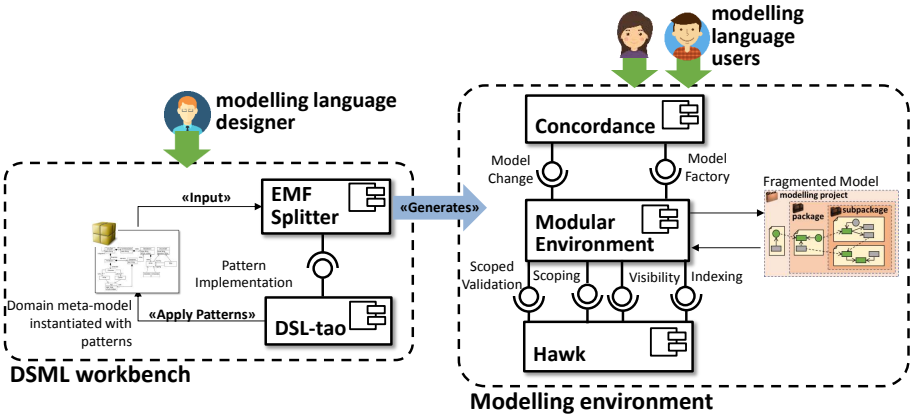


Figure 10: Architecture of EMF-Splitter.

This way, with our solution, the modelling language designer creates the domain meta-model and applies to it the modularity patterns using DSL-tao. The pattern applications are persisted as annotation models, which EMF-Splitter uses to synthesize a modelling environment. This environment provides the functionality defined by the patterns, like model creation using the defined fragmentation strategy and incremental validation of scoped constraints. The environment also provides support for working with legacy monolithic models, as it can automatically fragment an existing model according to the fragmentation strategy, and merge a partitioned model into a monolithic one.

The models created or fragmented by the generated environment are distributed in files and folders in the file system. In consequence, the integrity of a model can be affected if a file path changes (e.g., if a fragment is moved from a folder to another one). In order to maintain the model integrity, the generated environment integrates an indexer for cross-references called Concordance [30] (see Figure 10). By default, Concordance only indexes cross references. In addition, the generated environment implements its extension point ModelFactory

to index containment references as well, and its extension point ModelChange to receive notifications of model events in the workspace.

The generated environment can be extended through Eclipse extension points, for example, to integrate it with other tools. Based on this mechanism, our modularity patterns for scoped validation, reference scoping, visibility and indexing generate code that integrates Hawk [29], a scalable model indexer that improves the performance of queries over large models. This results in more scalable environments, as we will demonstrate in Section 6.2.

Figure 11(a) shows DSL-tao being used to define a modelling environment for the running example. The domain meta-model is built in the main canvas (label 1), and the patterns in the repository can be applied over it (Patterns View, with label 2). The repository includes patterns with services for filtering, graphical visualization and modularity, among others. Once a pattern is applied, the meta-model elements get tagged with the pattern role names. The environment provides a list of applied patterns, and when one is selected, the affected meta-model elements are highlighted in the canvas. The figure shows highlighted an application of the fragmentation pattern, while the upper-left corner of the canvas indicates that the Scope and Visibility patterns have been applied as well. Patterns can be instantiated using either a generic wizard or a specific wizard contributed by the pattern. The figure shows the wizard to instantiate the scoped validation pattern (label 3). This wizard permits selecting the context class, scope, and constraint statement using a dedicated editor (label 4). The editor features syntax validation of the constraint, which can be specified using the Epsilon Object Language (EOL) [36], a variant of OCL.

From the domain meta-model annotated with pattern instances, EMF-Splitter can synthesize a modelling environment. Figure 11(b) shows a snapshot of the environment for the running example. With the application of the fragmentation pattern, each model is represented as an Eclipse project (see Package Explorer view, with label 1), where packages are mapped to folders, and units to files. The model fragments in files can be edited using a tree editor (label 2), or a Sirius-based graphical editor which EMF-Splitter is also able to generate. The Problems view (label 3) shows the violations of scoped constraints.

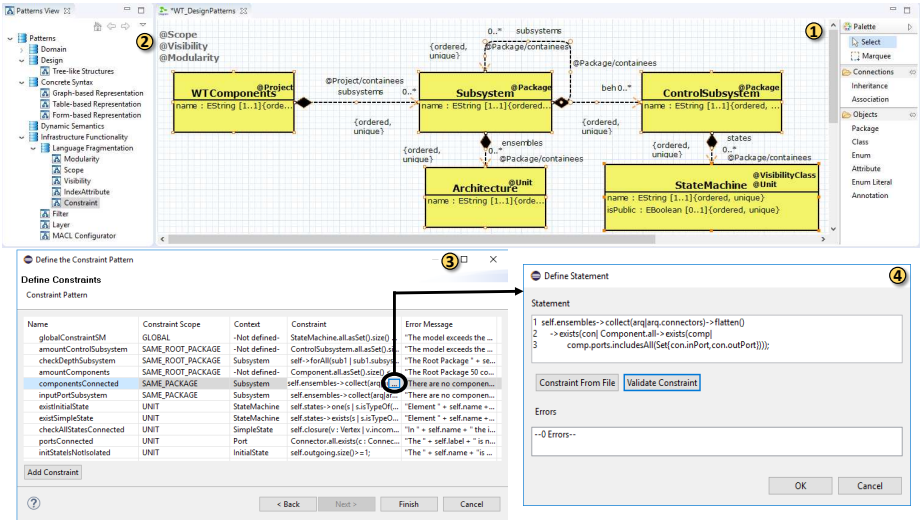
6. Evaluation

This section evaluates two relevant aspects of our approach, applicability and performance, which we reformulate as the following two research questions:

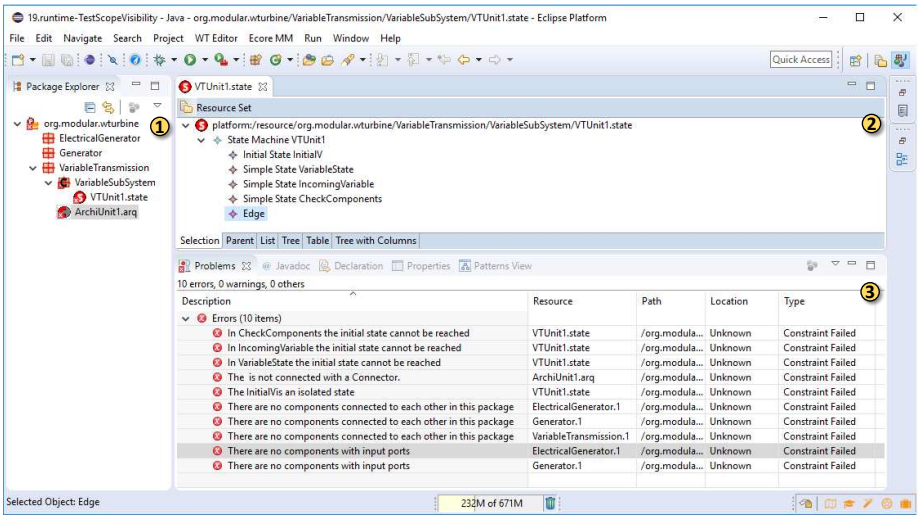
RQ1: Is the proposed model fragmentation approach into projects, packages and units applicable in practice?

RQ2: Is the evaluation of scoped constraints on fragmented models more efficient than the evaluation of standard constraints on monolithic models?

Next, Sections 6.1 and 6.2 describe the experiments we have performed to answer these questions and discuss the results. Then, Section 6.3 presents an



(a) Application of the scoped validation pattern using DSL-tao



(b) Generated modelling environment

Figure 11: Building a modelling environment for the running example.

industrial case study that further illustrates the usefulness of our proposal. In a previous publication [17], we reported on an evaluation of the scalability of our fragmentation strategies in combination with abstraction mechanisms to visualize large models, where we obtained a 97% reduction of memory consumption, and up to 55x speed-up of model visualization time.

The evaluation materials (e.g., meta-models, models) are available at `https://github.com/antoniogarmendia/emfsplitter-materials`.

6.1. Evaluating applicability of the fragmentation pattern

Our approach exploits the containment relations in meta-models to customize a fragmentation strategy. The rationale is that EMF meta-models make heavy use of containment relations, so that models have a tree structure where each object is contained under one parent, except the root object. Hence, a common idiom for EMF meta-models is to have a root class containing directly or indirectly all other classes of the meta-model. This root class plays the role of Project in our fragmentation pattern, while Package and Unit classes require subsequent containment relations.

To have an intuition of the practical applicability of our fragmentation approach, we have analysed the following two meta-model repositories to assess to which extent they make use of containment relations:

- The ATL meta-model zoo[2]. This is a repository hosted by the AtlanMod research team, consisting of 301 EMF meta-models created by developers with mixed experience. Hence, the repository includes meta-models used in academia and appearing in research papers, but also meta-models of large standards like BPMN or BPEL.
- Meta-models of OMG standards[3]. The Object Management Group (OMG) is a standardization body that produces meta-model-based standards for technologies like UML, OCL, QVT or BPMN, among others. These meta-models were created by professional engineers with high expertise. For our analysis, we have considered 224 meta-models of OMG standards, corresponding to those meta-models which are in a format we can parse, and considering that some standards provide several meta-models.

The repositories contain some very large meta-models. In the ATL zoo, the Industry Foundation Classes (IFC) meta-model contains 699 classes, and the OMG standard with most classes is the Robotic Interaction Service Specification (RoIS) with 657. On the other hand, both repositories have meta-models with a small number of classes. In the ATL zoo, 75% of meta-models have less than 38 classes, while in the OMG, 75% have less than 63 classes.

For each meta-model, we computed its *containment depth*. This is the length of the longest path of containment relations starting from the root class. We

[2] `http://web.emn.fr/x-info/atlanmod/index.php?title=Ecore`
[3] `https://www.omg.org/spec/`

detected the root class of each meta-model automatically as follows. For each class, we calculated the number of classes that it contained directly or indirectly. Then, the class containing more classes was selected as the root, and in case of tie, the class not contained by any other was selected. If several classes had those characteristics, then the first one in the list of possible roots was selected. The goal was to leave out as few classes as possible.

Within the analysed repositories, our algorithm did not detect a root in 7% of meta-models because they lacked containment relations, while 58% of meta-models had one root class. In the rest of cases, the algorithm found several roots in the same meta-model, which justifies our decision to support multiple roots in the fragmentation pattern.

Next, we heuristically assigned a fragmentation strategy to each meta-model. The heuristic annotated the root classes as Projects; the classes with recursive containment, or with containment depth greater than 1, as Packages; and the classes with containment depth equal to 0 or 1 as Units. By recursive containment we mean containment relations that can store instances of the class defining the relation, like Subsystem.subsystems in Figure 1.

Figure 12 shows the containment depth of the meta-models in both repositories (x axis) and the percentage of meta-models with that depth (y axis). We separate the ATL and OMG meta-models to understand whether there are differences between them, given the different background of their developers and the different scope of the meta-models.

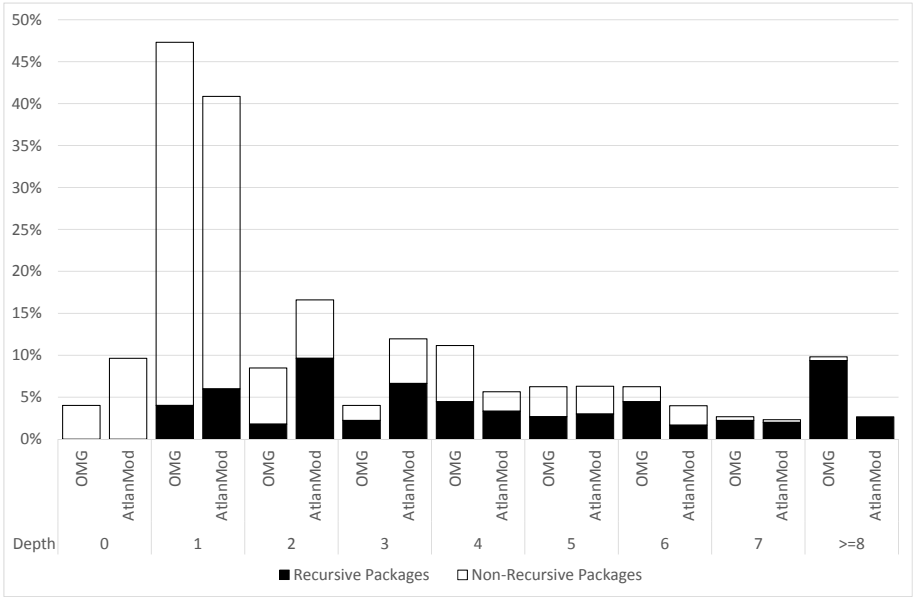


Figure 12: Containment depth across the repositories, and distribution of packages and recursive packages according to the depth.

The figure shows that less than 10% of the ATL meta-models and less than 5% of the OMG meta-models have no containment relations. In the ATL repository, 29 meta-models have no root, and 75% of them have less than 16 classes. In the OMG repository, 9 meta-models have no root, and 75% of them have less than 29 classes. Our fragmentation pattern cannot be applied to these meta-models as they lack a root class.

On the contrary, the depth is greater than zero in more than 90% of the meta-models, and our approach can be used on them. Most meta-models in both repositories have depth 1. The ratio of recursive containment relations (leading to recursive packages) vs. non-recursive packages increases as the containment depth increases. Interestingly, excluding the meta-models with depth zero, the containment depth follows a power law distribution, found in many natural and man-made phenomena [37].

Figure 13 depicts the correlation of containment depth (x axis) with meta-model size as number of classes (y axis). We can see that there is a tendency for longer containment depths in bigger meta-models in both repositories.

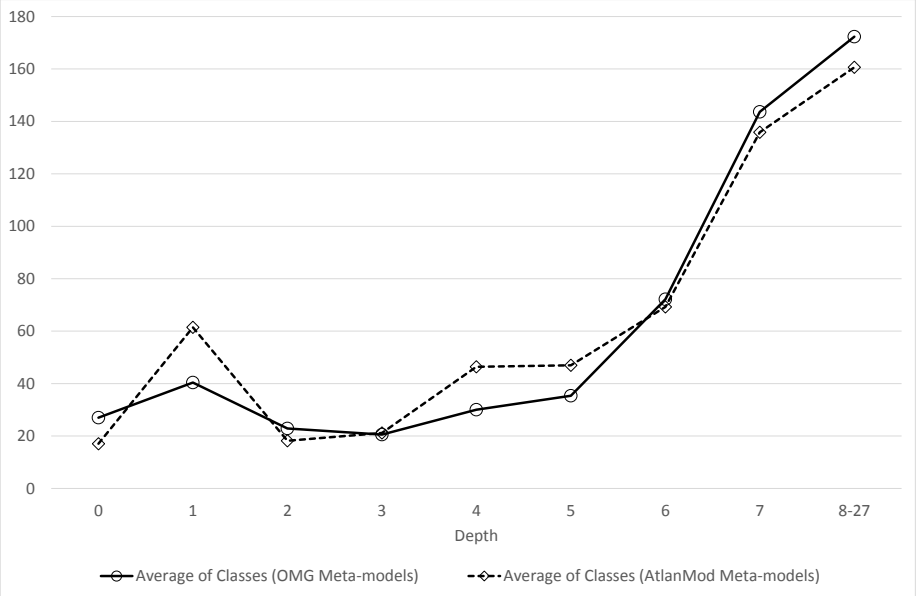


Figure 13: Meta-model size (in classes) vs containment depth.

From this experiment, we conclude that our fragmentation approach can be applied to most meta-models in both repositories (potentially 90% of ATL meta-models and 95% of OMG meta-models). With regards to the fragmentation strategies that we computed heuristically, we found that nearly 50% of the meta-models in both repositories contain some Package class. Packages permit grouping model fragments and provide a modular structure, which is one of the main objectives of our pattern. The analysis also shows that there are two

features that make a meta-model more amenable to fragmentation using our approach: first, deep containment trees permit creating different types of packages; second, recursive containment relations permit nested packages. Hence, our fragmentation is more useful on larger meta-models, as their containment depth and ratio of recursive containment relations are higher.

Altogether, we can answer the research question **RQ1** affirmatively: our fragmentation pattern is applicable in practice, being more beneficial for large meta-models.

6.1.1. Threats to validity

The analysis considers a large number of meta-models (more than 500) including standards, which ensures the robustness of the findings. Moreover, it takes into account two different repositories to foster diversity of meta-models. To strengthen our results, we plan to repeat the analysis on meta-models hosted in public code versioning systems like Github. On the other hand, our experiment applied fragmentation strategies automatically computed according to a set of heuristics, but these strategies may be different from the ones that a human may have manually defined. It is future work to perform another evaluation using fragmentation strategies manually defined by developers.

6.2. Evaluating performance of scoped constraints

To evaluate the performance gains of our approach, next, we report on four experiments analysing the effects of fragmentation and scope in the execution time of constraint validation:

1. First, we compare the validation time of standard constraints on monolithic models (the baseline) with respect to the full validation of equivalent scoped constraints in fragmented models.

2. Second, we investigate whether the number of fragments affects the validation performance.

3. Next, we compare the full validation of scoped constraints (i.e., on all units and packages of a fragmented model) vs. their incremental validation (i.e., only on the elements affected by a model change).

4. Finally, we analyse the efficiency gains when integrating the Hawk model indexer with scoped constraint validation.

Experiment setting. In all four experiments, we used the meta-model of the running example (see Figure 1) and the fragmentation strategy defined in Figure 4. We considered a suite of eleven EOL constraints in both standard and scoped versions. The constraints are available in Appendix A, and Table 1 summarizes their characteristics. We considered constraints with all kinds of scope (one with scope sameProject, three with scope sameRootPkg, two with samePkg, and five with sameUnit). As a measure of their complexity, the table includes the number of nodes in the abstract syntax tree of each constraint expression.

For example, the expression StateMachine.allInstances()→size() <= 10 corresponding to constraint numberStateMachines has three nodes. The average number of nodes in the constraints is 6.3, ranging from three to fifteen.

Table 1: Characteristics of scoped constraints used in the evaluation of performance.

Constraint	**Scope**	**Complexity (#nodes)**
numberStateMachines	sameProject	3
numberControlSubsystems	sameRootPkg	3
numberComponents	sameRootPkg	3
depthSubsystem	sameRootPkg	13
connectedComponents	samePkg	11
inputPortSubsystem	samePkg	8
oneInitialState	sameUnit	3
existsSimpleState	sameUnit	3
reachableState	sameUnit	5
connectedPorts	sameUnit	15
initStateIsNotIsolated	sameUnit	3

The experiments were executed four times in a computer with Windows 10 Education version, processor Intel(R) Core(TM) i7-3770, 3.40GHz, and Java SE 1.8 with 8GB as initial and maximum memory. The constraints were validated on synthetic models of increasing size (from around 20 000 objects to around 125 000 objects) created using the EMF random instantiator from the AtlanMod team[4].

6.2.1. Full constraint validation in monolithic and fragmented models

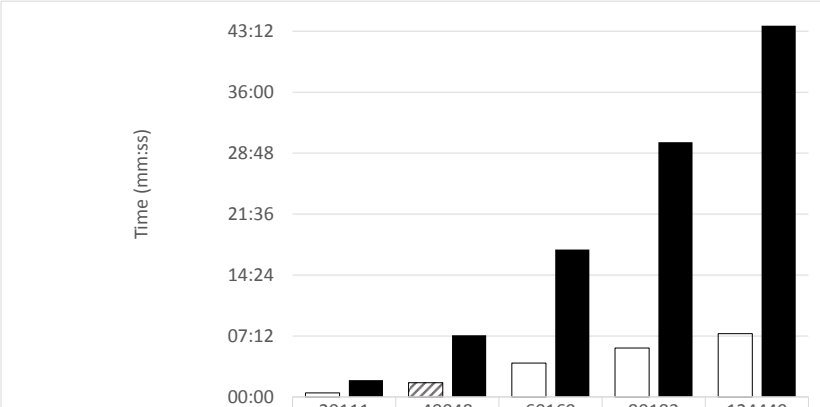
The first experiment compares the validation of standard constraints in a monolithic model, with the validation of equivalent scoped constraints in the fragmented version of the same model. We consider the full validation of scoped constraints, i.e., their validation on all units and packages of the fragmented model. The objective is to assess whether reducing the number of objects in the validation scope also reduces the validation time.

Figure 14 shows the experiment results. The vertical axis shows the validation time, and the horizontal one the size of the model in number of objects. The graphic shows that the validation of scoped constraints is faster even for the smallest model of 20 111 objects. For the largest model, scoped validation is six times faster than standard validation. As explained in Section 4.2, this happens because scoped constraints are validated within a limited scope, and hence, fewer objects need to be loaded/queried. Moreover, scoped validation is only performed on those packages/units that may contain instances of the context class on which the scoped constraint is defined.

6.2.2. Effect of number of fragments on scoped validation performance

The previous experiment shows that scoped validation pays off even for medium-sized models. However, fragmentation incurs an overhead, as each fragment requires an access to disk to load the fragment in memory. Therefore,

[4] `http://modeling-languages.com/a-pseudo-random-instance-generator-for-emf-models/`



	20111	40048	60169	80183	124449
Scoped validation in fragmented model (mm:ss)	00:29	01:41	04:00	05:47	07:29
Standard constraint validation in monolithic model (mm:ss)	01:59	07:17	17:25	30:06	43:51

Figure 14: Constraint validation times in monolithic and fragmented models.

this second experiment analyses the impact of the number of model fragments on the scoped validation performance. For this experiment, we created six synthetic models of 20 000 objects, each one of them fragmented in a different number of files ranging from 1 to 5 000. Then, we measured the validation time of the eleven constraints used in the first experiment.

Figure 15 shows the results. If the number of files is low (horizontal axis), the cost to iterate through the objects in the fragments increases (vertical axis). In the limit, if the model is in one file, the scoped validation time is similar to the time of evaluating the constraints in a monolithic model. Conversely, if the model is fragmented in many files, the overhead of loading them becomes apparent and the efficiency decreases. In this experiment, the best validation time was obtained when fragmenting the model of 20 000 objects in 200 files. This gives a ratio of 1 file for each 100 objects. However, this ratio cannot be taken as a general guideline, as the optimal ratio may depend on the structure and scope of the involved constraints. It is up to future work to investigate methods to obtain optimal fragmentation sizes given a set of constraints.

6.2.3. Comparison of full validation and incremental validation

Section 4.5 presented an algorithm for incremental scoped validation that is applicable when a localized model change occurs, optimizing the re-evaluation of constraints to the scope of the change. Hence, in this experiment, we emulate a model change, and then compare the incremental validation time (i.e., the re-evaluation of constraints on the affected model elements) and the full validation time (i.e., the re-evaluation of constraints on all model elements). The experiment considers changes on units that are located at different containment

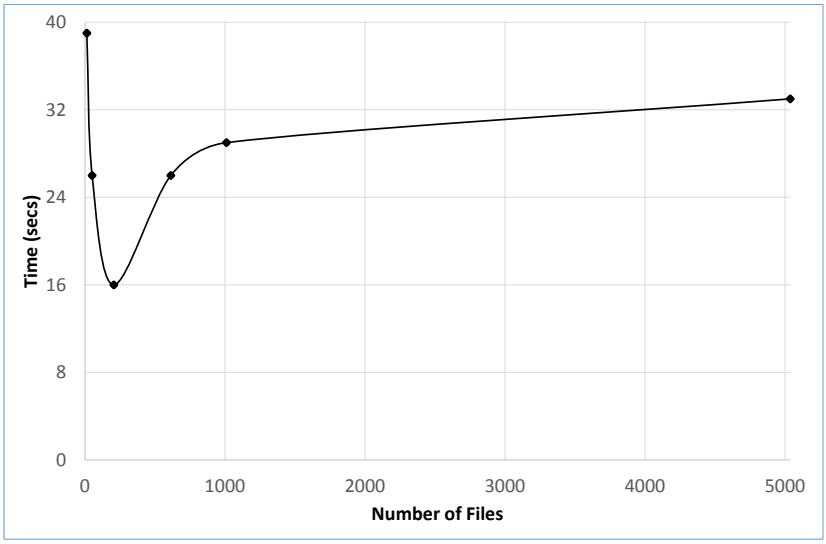


Figure 15: Effect of the number of files on the scoped validation performance.

depths, from 2 to 5. We distinguish the depth of the changed resource because changes in resources that are deeper in the containment tree need to re-evaluate more constraints, incurring longer validation times.

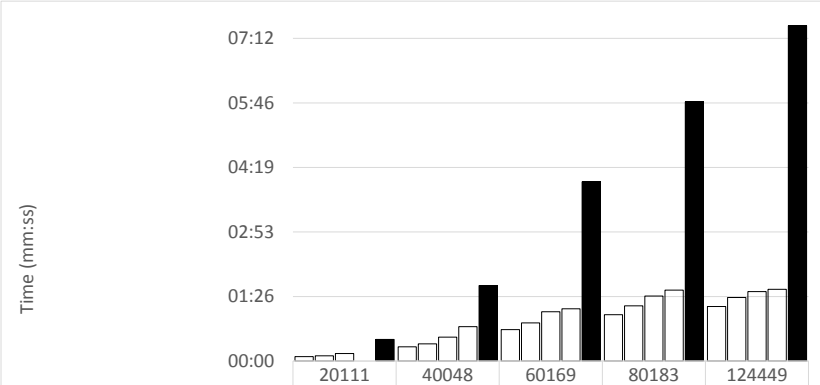
Figure 16 presents the results. The figure does not show data for the model with 20 111 objects at level 5, because this model has no units or packages at this level. We can observe that the incremental validation scales better than the full one. In average, the incremental validation time is just 1 second slower for models of 124 449 objects than for models of 80 183 objects, while the full validation is more than 90 seconds slower. For the biggest model, the incremental validation yields a speed-up of around 4.5x.

6.2.4. Effect of a model indexer on scoped validation performance

Next, we evaluate the use of the Hawk model indexer to execute scoped constraints. Although Hawk is integrated with several database technologies, we used Neo4j[5] for this experiment.

This case study drove a number of optimisations and additions to Hawk. The most notable change was the addition of two new query scoping modes for Hawk. Hawk already had the capability to limit the scope of a query so that `Type.all` would only return the instances within a certain subset of the indexed locations and/or files. Previously, Hawk only had one implementation for this: it would go to the type node, and then visit each instance node while checking if it belonged to a file node within the desired scope. For sufficiently common types,

[5] `https://neo4j.com/`



	20111	40048	60169	80183	124449
Incremental validation: level 2	00:06	00:19	00:42	01:02	01:13
Incremental validation: level 3	00:07	00:23	00:51	01:14	01:25
Incremental validation: level 4	00:10	00:32	01:06	01:27	01:33
Incremental validation: level 5		00:46	01:10	01:35	01:36
Full validation	00:29	01:41	04:00	05:47	07:29

Figure 16: Comparison of incremental and full scoped validation times.

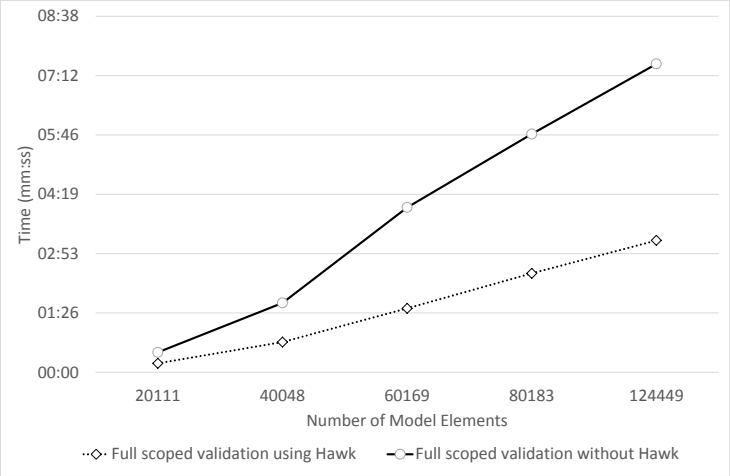
Hawk would end up visiting more instances than necessary. Two alternative implementations of Type.all were implemented to reduce the number of misses:

- The new *file-first* mode was used for validation rules spanning single files. It iterates over the contents of the file, filtering objects by type. This is useful when we have more instances of the type than objects in a typical fragment, which may be the case for sufficiently large fragmented models.
- The new *subtree scoping* mode was used for validation rules spanning projects or packages. The mode uses an advanced feature of Hawk called *derived edges*. Derived edges are references that are precomputed according to a *derivation logic* specified by the user. Derived edges are updated incrementally as new versions of the fragments are detected.

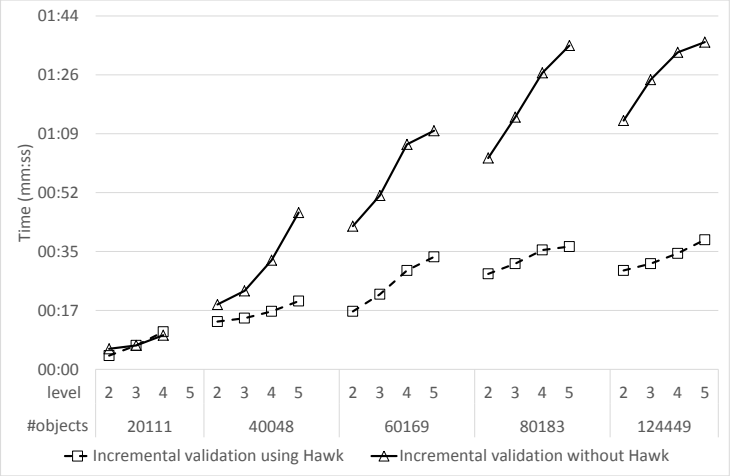
  Upon a request for Type.all, the subtree scoping mode will ensure that Hawk precomputes "allof_Type" derived edges from each instance to all its containers. Once this is done, answering Type.all scoped to the subtree rooted at the x project/package only requires following the "allof_Type" edges from x in reverse.

Figure 17(a) compares the full validation time with and without Hawk. For the biggest model, Hawk speeds up constraint evaluation more than 2x.

Figure 17(b) makes the same comparison but for incremental validation, and distinguishing the depth of the changed resource. As before, we generally obtained shorter validation times using Hawk, peaking a speed up of around 3x for the biggest model. However, the validation time for some of the smallest models (the deeper ones) was slightly faster without indexing.



(a) Full validation times



(b) Incremental validation times

Figure 17: Validation times with and without Hawk.

In view of the results obtained in the four experiments presented in this section, we can answer **RQ2** affirmatively: scoped constraint validation is more efficient than the evaluation of equivalent non-scoped constraints on monolithic models. Model fragmentation and scoped constraints permit the incremental validation of constraints, which leads to speed ups of up to 4.5x. Incremental validation can be enhanced with model indexers, obtaining an additional increase of efficiency up to 3x. The number of files in which a model is fragmented has an effect in performance as well; for this particular experiment, having 1 file for each 100 objects yielded optimal validation times, though this ratio may vary for other cases.

6.2.5. Threats to validity

This evaluation has been performed using a meta-model that comes from an industrial partner in a European project[6]. The aim of the project was improving scalability in MDE, and the meta-model was expected to have large models, though maybe not such large as in the experiment. Moreover, we slightly adapted the meta-model to better illustrate our approach, removing some of its complexity. Therefore, our results may have been different if the meta-model would have been designed differently. To mitigate this risk, the following section presents a case study built over a meta-model and a set of constraints from a third-party.

On the other hand, the evaluation uses synthetic models generated using an open-source random model instantiator. We configured the instantiator with a real seed model, and made sure that the generated models were balanced, in the sense that their fragmentation resulted in fragments with a reasonable size and structure (e.g., subsystem packages always contain at least one architecture file and zero or more subsystem packages, while architecture files contain several components connected by ports). This way, we generated more realistic models, and mitigated the impact that the use of synthetic models may have on the results of the evaluation.

6.3. Case study

In this section, we compare an existing meta-model-based modelling environment developed by a third-party, with another one created by us using our approach. We use as a case study the environment developed for Computer Aided Engineering Exchange (CAEX), which is described in [16]. CAEX is a neutral data format, used as a standard by the International Electrotechnical Commission (IEC) to represent the hierarchical structure of production systems. Other IEC standards use CAEX. For example, AutomationML uses CAEX to model plant components, including devices and communication structures.

To create an environment for CAEX, we first applied the fragmentation pattern to its meta-model, using as guiding principles the editor and models in the Github repository `https://github.com/amlModeling/`. Figure 18(a) shows an

[6] `http://mondo-project.org`

excerpt of the meta-model annotated with the instantiated fragmentation pattern. A CAEXFile (project) stores the engineering data in instanceHierarchies and contains several libraries of elements (RoleClassLib, InterfaceClassLib and SystemUnitClassLib, which are packages). Altogether, we identified one project class, four package classes, and seven unit classes in the meta-model.

Then, we converted the EVL [38] constraints available in the same Github repository into EOL scoped constraints in our meta-model. Appendix B contains the list of scoped constraints, and Table 2 shows their characteristics. There is one constraint with scope sameProject, one with scopes samePkg and sameUnit at the same time, and seven with scope sameUnit. The average number of nodes in the constraints is 10.3, ranging from two to seventeen.

Table 2: Characteristics of scoped constraints used in the case study (CAEX).

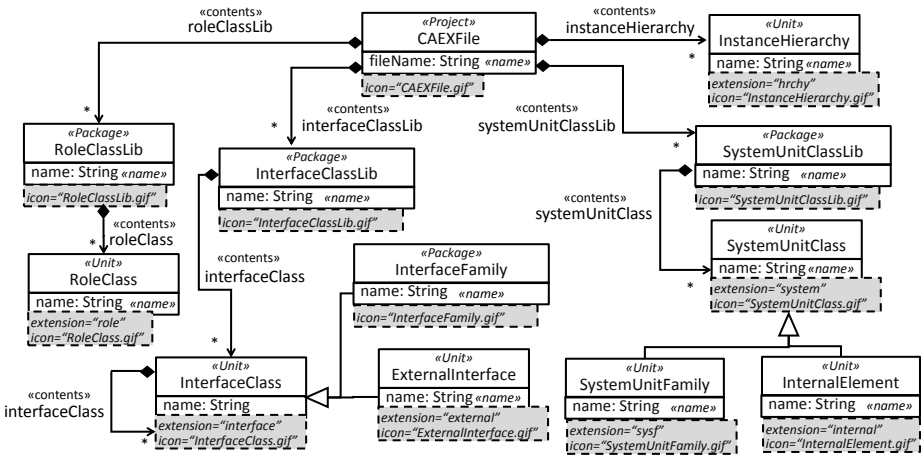
Constraint	**Scope**	**Complexity (#nodes)**
superiorStandardVersionIsMandatory	samePkg	3
CAEXObject	samePkg,sameUnit	2
inheritanceMustPointToSUC	sameUnit	5
strongConformanceSUC2IE	sameUnit	15
strongConformanceIE2SUC	sameUnit	15
noInheritanceForIEs	sameUnit	2
processContainsProcesses	sameUnit	17
resourceContainsResources	sameUnit	17
productContainsProducts	sameUnit	17

Finally, we used EMF-Splitter to generate an environment for CAEX from the meta-model and instantiated patterns. Figure 18(b) shows the environment. The Package Explorer view contains a project that represents a fragmented model (label 1). One model fragment is being edited (label 2). The Problems view lists the violated constraints (label 3). The Hawk view shows a running instance of Hawk for EMF-Splitter (label 4).

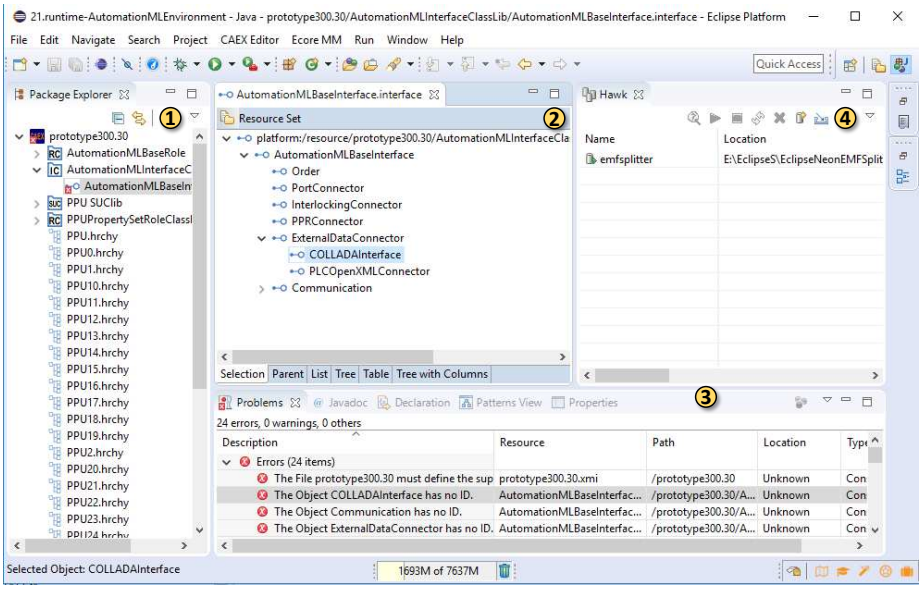
Next, we compared the constraint validation performance in our environment and the original one. For this purpose, we created models of size 210 162, 310 222 and 410 282 objects, and measured the time to validate the defined constraints on them. We created the models using a model generator available in the same Github repository, slightly modified to enable the generation or larger, more balanced models, compatible with CAEX version 3.0.

Figure 19 shows the time of validating the scoped constraints on the fragmented models, considering both full validation and incremental validation with Hawk upon an emulated model change. It also shows the validation time of the standard non-scoped constraints on the equivalent monolithic models. Using incremental validation with Hawk is the most efficient, as it improves in one third the validation time of standard constraints. The incremental validation on a model with 410 282 objects takes less than one second.

Overall, this case study shows that we could improve an existing industrial modelling environment with model fragmentation (strengthening our positive answer to **RQ1**), and improved performance of constraint evaluation (strengthening our positive results regarding **RQ2**).



(a) Excerpt of the CAEX meta-model annotated with the fragmentation strategy



(b) Generated modelling environment

Figure 18: Building a modelling environment for the case study (CAEX).

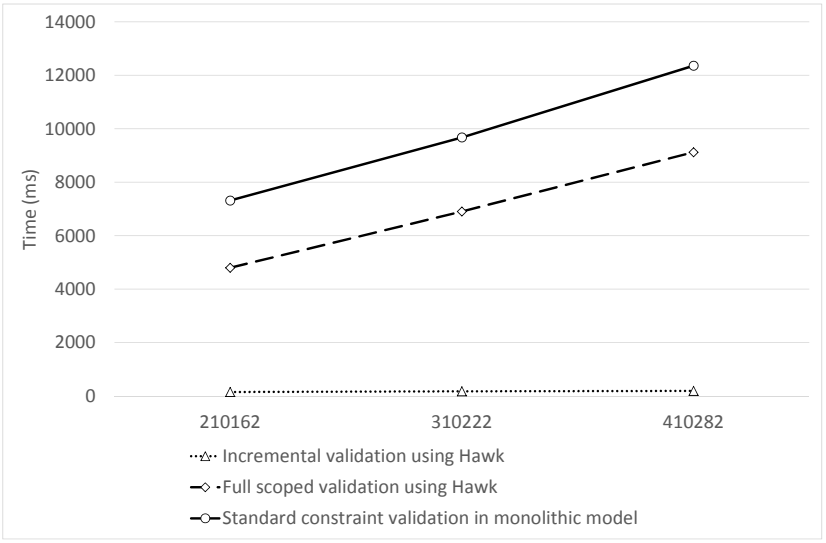


Figure 19: Comparison of full validation, incremental validation and baseline for CAEX.

7. Related work

Modularization (i.e., breaking a large system into smaller parts) is an essential mechanism for tackling the complexity of systems, and is common in many modelling notations [39]. However, support for modularity is often developed ad-hoc for each modelling language [40], which can be laborious to implement manually. Therefore, we contribute a generic pattern-based approach to customize fragmentation and other modularity services for arbitrary DSML meta-models.

The reuse of DSMLs is a major concern in the software language engineering (SLE) community. The aim is developing modularity concepts at the language level to permit creating new DSMLs out of existing ones, e.g., by composition [41, 42, 43]. Instead, our focus is on offering modularization services within single DSMLs to enable scalable modelling. Nonetheless, our services are reusable for different DSMLs. Related to this, the SLE community has urged researchers to explore the notion of *language interface* as a mechanism to reuse language services [44]. Our patterns are an example of interfaces that allow reusing these services.

There are some proposals to achieve modularity in modelling. For example, aspect-oriented modelling [45, 46] adapts ideas from aspect-oriented programming to modelling. Cross-cutting concerns can be modelled separately and then woven and composed into base models. While our fragmentation services could be the basis for aspect-oriented modelling, we do not provide weaving support.

Multi-view modelling permits describing a complex system using several model views. This promotes separation of concerns and helps tackling the complexity of the system [47, 48]. While each view is understandable on its own, it

needs to be consistent with the other views to achieve a meaningful system description. Frequently, views need to store redundant information (e.g., different aspects of the same element), and techniques to derive the whole system from the views are provided. Our approach could be used as a basis to build a multi-view framework, but for this purpose, it should be extended with mechanisms to handle redundant information.

Concepts from component-based software engineering [49] have been transferred to MDE to improve modularity. For example, in [50, 51, 52], composite models are created using components which comprise a model and a set of import and export interfaces for their interconnection. Our visibility pattern also provides an interface for a fragment to the rest of the model, customised with a visibility level (package, root package, project, or workspace). Moreover, our fragmentation pattern enables a hierarchical organization of models into packages, which we exploit to optimize constraint validation.

FRAGMENTA is the theory of model fragmentation [53] that describes our fragmentation strategy. The main result of the theory is the observation that the satisfaction of some local fragment constraints is enough to ensure that some relation types (inheritance, composition) remain well-formed (acyclic) globally when fragments are composed. While FRAGMENTA is theoretical, here we provide a concrete realisation along with further modularity services.

To improve the processing of large models, some authors have proposed splitting them for solving different tasks. For instance, models are sliced in [54] to improve the efficiency of test input data generation. Another example is EMF Fragments [55], a persistence framework which allows automatic and transparent fragmentation to add, edit and update EMF models. This process is executed at runtime with considerable performance gains. The framework has been applied to the model-based analysis of large code repositories [56]. In our case, we provide further services that profit from the fragmentation (e.g., scoped constraints), generate a modelling environment that integrates these services, and use a model indexer to improve efficiency.

Sometimes, the motivation for decomposing models into chunks is enhancing their comprehensibility. For example, in [57], the authors tackle the problem of synthesizing a lattice with all possible ways to partition a model into submodels, such that each submodel conforms to the meta-model. The meta-model can define OCL invariants restricted to so-called forward-constraints, which for example forbids the use of allInstances. In contrast, our scoped constraints have no restrictions, and their scope can be used to improve the efficiency of their evaluation and reduce unnecessary re-evaluations.

Also to improve model comprehension, Strüber and collaborators [58] use Information Retrieval (IR) algorithms to split a model based on the relevance of its elements. Hence, splitting models that belong to the same meta-model can produce different structures. We plan to include this fragmentation strategy into our framework in future work.

Some works propose techniques to improve the performance of querying large models [34, 59, 60]. For models persisted in NoSQL (a common choice for large models), in [59], the authors extend Morsa – a model persistence backend based

on NoSQL databases – with the query language MorsaQL, an internal language embedded in Java. Mogwaï [60] is a model query framework which translates OCL into Gremlin, a query language supported by several NoSQL engines. Not specifically for NoSQL, IncQuery-D [34] uses incremental graph search techniques along with a distributed cloud infrastructure to enhance query efficiency in large models. This tool relies on VIATRA2 to specify the queries [61]. Model/-Analyzer is a UML consistency checker, which supports incremental constraint evaluation by detecting the scope of changes and dynamically selecting the constraints to re-evaluate [33]. The technique in [32] is similar, but for UML/OCL schemas. With respect to these approaches, our contribution is to exploit an explicit definition of fragmentation strategies and constraint scopes. Compared to incremental approaches, relying on fragmentation avoids the need to calculate the scope of changes at run-time [32, 33], or incur in high memory costs for memoisation [34]. We believe the concepts of fragmentation and scoping may be transferred to those approaches to improve their efficiency.

Textual DSLs often need mechanisms to support reference scoping and visibility rules [62]. These are commonly supported by workbenches for textual DSLs like Xtext [21] and Spoofax [63]. While namespaces and scope rules are defined with a DSL in Spoofax, Xtext offers a Java API (based on the IScopeProvider interface) for this purpose. In our case, reference scoping and visibility rules are defined on the abstract syntax level, i.e., not tied to a particular concrete syntax, and using high-level patterns.

Regarding graphical language workbenches, Sirius [20] is a framework that permits defining visual concrete syntaxes for meta-models using a model-based approach. It supports viewpoints, layers and drill-down exploration. However, these facilities are at the concrete syntax level, but the model itself remains monolithic. Instead, our modularity services work at the abstract syntax level, producing a truly fragmented model. We have integrated our fragmentation pattern with high-level wizards to produce scalable Sirius editors [25], but a full integration with the other modularity services is future work.

Altogether, even though techniques for model modularization have been proposed to improve scalability in MDE, they typically focus on one specific aspect – like fragmentation, scoping, persistence or query – and sometimes are tied to a concrete syntax style (textual or graphical). Instead, ours is a comprehensive proposal providing five services (fragmentation, reference scoping, visibility, indexing and constraint scoping) that are highly reusable and customizable via pattern instantiation.

8. Conclusions and future work

In this paper, we have discussed the need for modularization services to improve the scalability of DSMLs. For this purpose, we have proposed five modularity services (fragmentation, reference scoping, object visibility, field indexing, and scoped validation) which can be configured by means of patterns. We have presented tool support (EMF-Splitter), evaluated its applicability (by analysing meta-model repositories built by third parties), assessed the efficiency gains on

a synthetic running example inspired by an industrial scenario, and with respect to an existing modelling tool in the industrial automation domain.

We are currently developing an API to facilitate the programmatic use of EMF-Splitter, and improving the fragmentation service with the possibility to fragment models across non-containment references, e.g., based on strategies like [58]. Another line of future research is developing heuristics to recommend a scope and attribute indices for a given constraint, based on the static analysis of the constraint. In particular, we plan to recommend a scope for an OCL constraint based on the scope and visibility of the objects and references appearing in it. Our idea is traversing the constraint expression and, for each reference access, look at the applied patterns to extract the scope assigned to the reference, and the visibility assigned to the reference type, annotating the reference with the narrowest of both. When the scope of all reference accesses has been computed in this way, we plan to heuristically recommend the widest one as the scope of the constraint.

We are also investigating methods to recommend optimal fragmentation granularities given a set of constraints. Finally, fragmented models are more amenable to collaborative use – for example via version control systems – than monolithic models, as they tend to produce fewer conflicts. Hence, we plan to integrate our approach with version and access control systems for collaborative modelling [64].

Acknowledgements. Work funded by the R&D programme of the Madrid Region (P2018/TCS-4314).

References

[1] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, Morgan & Claypool Publishers, 2012.

[2] D. C. Schmidt, Guest editor's introduction: Model-driven engineering, Computer 39 (2) (2006) 25–31. doi:10.1109/MC.2006.58.

[3] S. Kelly, J. Tolvanen, Domain-Specific Modeling - Enabling Full Code Generation, Wiley, 2008.

[4] D. S. Kolovos, L. M. Rose, N. D. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, M. Tisi, J. Cabot, A research roadmap towards achieving scalability in model driven engineering, in: BigMDE@STAF, ACM, 2013, pp. 2:1–2:10.

[5] F. DeRemer, H. H. Kron, Programming-in-the-large versus programming-in-the-small, IEEE Trans. Software Eng. 2 (2) (1976) 80–86.

[6] R. W. Quong, M. A. Linton, Linking programs incrementally, ACM Trans. Program. Lang. Syst. 13 (1) (1991) 1–20. doi:10.1145/114005.102804.

[7] S. P. Reiss, An approach to incremental compilation, in: SIGPLAN Symposium on Compiler Construction, 1984, pp. 144–156.

[8] ArgoUML, http://argouml.tigris.org/ (last accessed in 2018).

[9] Enterprise Architect, http://sparxsystems.com/products/ea/ (last accessed in 2018).

[10] IBM Rational, https://www-01.ibm.com/software/rational/uml/products/ (last accessed in 2018).

[11] MagicDraw, https://www.nomagic.com/products/magicdraw (last accessed in 2018).

[12] Modelio, https://www.modelio.org/ (last accessed in 2018).

[13] Papyrus, https://www.eclipse.org/papyrus/ (last accessed in 2018).

[14] A. García-Domínguez, K. Barmpis, D. S. Kolovos, R. Wei, R. F. Paige, Stress-testing remote model querying APIs for relational and graph-based stores, Software and System Modeling (2017) 1–29.

[15] A. Garmendia, E. Guerra, D. S. Kolovos, J. de Lara, EMF Splitter: A structured approach to EMF modularity, in: XM@MoDELS, Vol. 1239 of CEUR Workshop Proceedings, CEUR-WS.org, 2014, pp. 22–31.

[16] T. Mayerhofer, M. Wimmer, L. Berardinelli, R. Drath, A model-driven engineering workbench for CAEX supporting language customization and evolution, IEEE Trans. Industrial Informatics 14 (6) (2018) 2770–2779.

[17] A. Jiménez-Pastor, A. Garmendia, J. de Lara, Scalable model exploration for model-driven engineering, Journal of Systems and Software 132 (2017) 204–225.

[18] A. Gómez, X. Mendialdua, G. Bergmann, J. Cabot, C. Debreceni, A. Garmendia, D. S. Kolovos, J. de Lara, S. Trujillo, On the opportunities of scalable modeling technologies: An experience report on wind turbines control applications development, in: ECMFA, Vol. 10376 of Lecture Notes in Computer Science, Springer, 2017, pp. 300–315.

[19] GMF, http://www.eclipse.org/modeling/gmp/ (last accessed in 2018).

[20] Sirius, https://eclipse.org/sirius/ (last accessed in 2018).

[21] Xtext, https://www.eclipse.org/Xtext/ (last accessed in 2018).

[22] E. Dijkstra, On the role of scientific thought, EWD447 (1974).

[23] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2^{nd} Edition, Addison-Wesley Professional, 2008, see also http://www.eclipse.org/modeling/emf/.

[24] OCL, http://www.omg.org/spec/OCL/ (2014).

[25] A. Pescador, A. Garmendia, E. Guerra, J. S. Cuadrado, J. de Lara, Pattern-based development of domain-specific modelling languages, in: MODELS, IEEE Computer Society, 2015, pp. 166–175.

[26] T. Kühn, S. Böhme, S. Götz, U. Aßmann, A combined formal model for relational context-dependent roles, in: SLE, ACM, 2015, pp. 113–124.

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

[28] J. de Lara, E. Guerra, J. S. Cuadrado, When and how to use multilevel modelling, ACM Trans. Softw. Eng. Methodol. 24 (2) (2014) 12:1–12:46.

[29] K. Barmpis, D. S. Kolovos, Towards scalable querying of large-scale models, in: ECMFA, Vol. 8569 of Lecture Notes in Computer Science, Springer, 2014, pp. 35–50.

[30] L. M. Rose, D. S. Kolovos, N. Drivalos, J. R. Williams, R. F. Paige, F. A. C. Polack, K. J. Fernandes, Concordance: A framework for managing model integrity, in: ECMFA, Vol. 6138 of Lecture Notes in Computer Science, Springer, 2010, pp. 245–260.

[31] JDT, Java Development Tools, `http://www.eclipse.org/jdt/`.

[32] J. Cabot, E. Teniente, Incremental integrity checking of UML/OCL conceptual schemas, Journal of Systems and Software 82 (9) (2009) 1459–1478.

[33] A. Egyed, K. Zeman, P. Hehenberger, A. Demuth, Maintaining consistency across engineering artifacts, IEEE Computer 51 (2) (2018) 28–35.

[34] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, D. Varró, Emf-incquery: An integrated development environment for live model queries, Sci. Comput. Program. 98 (2015) 80–99.

[35] E. Clayberg, D. Rubel, Eclipse plugins, 3^{rd} Edition, Addison-Wesley Professional, 2008, see also `http://www.eclipse.org/`.

[36] D. S. Kolovos, R. F. Paige, F. A. C. Polack, The Epsilon Object Language (EOL), in: A. Rensink, J. Warmer (Eds.), ECMDA-FA, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 128–142.

[37] M. Newman, Power laws, Pareto distributions and Zipf's law, Contemporary Physics 46 (5) (2005) 323–351.

[38] D. S. Kolovos, R. F. Paige, F. A. Polack, Eclipse development tools for Epsilon, in: Eclipse Summit Europe, Eclipse Modeling Symposium, Vol. 20062, 2006, p. 200.

[39] D. L. Moody, The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering, IEEE Trans. Software Eng. 35 (6) (2009) 756–779.

[40] J. L. Lawall, H. Duchesne, G. Muller, A.-F. L. Meur, Bossa Nova: Introducing modularity into the Bossa domain-specific language, in: GPCE, Vol. 3676 of Lecture Notes in Computer Science, Springer, 2005, pp. 78–93.

[41] H. Krahn, B. Rumpe, S. Völkel, Monticore: a framework for compositional development of domain specific languages, STTT 12 (5) (2010) 353–372.

[42] T. Degueule, B. Combemale, A. Blouin, O. Barais, J. Jézéquel, Melange: a meta-language for modular and reusable development of dsls, in: Proc. SLE, ACM, 2015, pp. 25–36.

[43] A. Sutîi, M. van den Brand, T. Verhoeff, Exploration of modularity and reusability of domain-specific languages: an expression DSL in metamod, Computer Languages, Systems & Structures 51 (2018) 48–70.

[44] T. Degueule, B. Combemale, J. Jézéquel, On language interfaces, in: Present and Ulterior Software Engineering., Springer, 2017, pp. 65–75.

[45] M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, E. Kapsammer, A survey on uml-based aspect-oriented design modeling, ACM Comput. Surv. 43 (4) (2011) 28:1–28:33.

[46] F. Heidenreich, J. Henriksson, J. Johannes, S. Zschaler, On language-independent model modularisation, T. Aspect-Oriented Software Development VI 6 (2009) 39–82.

[47] C. Atkinson, C. Tunjic, T. Moller, Fundamental realization strategies for multi-view specification environments, in: EDOC, IEEE Computer Society, 2015, pp. 40–49.

[48] H. Bruneliere, E. Burger, J. Cabot, M. Wimmer, A feature-based survey of model view approaches, Software and System Modeling (in press) (2017) 1–22.

[49] B. J. Cox, A. J. Novobilski, Object-oriented programming - an evolutionary approach (2. ed.), Addison-Wesley, 1991.

[50] S. Jurack, G. Taentzer, A component concept for typed graphs with inheritance and containment structures, in: ICGT, Vol. 6372 of Lecture Notes in Computer Science, Springer, 2010, pp. 187–202.

[51] D. Strüber, S. Jurack, T. Schäfer, S. Schulz, G. Taentzer, Managing model and meta-model components with export and import interfaces, in: BigMDE@STAF, Vol. 1652 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 31–36.

[52] D. Strüber, G. Taentzer, S. Jurack, T. Schäfer, Towards a distributed modeling process based on composite models, in: FASE, Vol. 7793 of Lecture Notes in Computer Science, Springer, 2013, pp. 6–20.

[53] N. Amálio, J. de Lara, E. Guerra, Fragmenta: A theory of fragmentation for MDE, in: MoDELS, IEEE, 2015, pp. 106–115.

[54] D. D. Nardo, F. Pastore, L. C. Briand, Augmenting field data for testing systems subject to incremental requirements changes, ACM Trans. Softw. Eng. Methodol. 26 (1) (2017) 1:1–1:40.

[55] M. Scheidgen, A. Zubow, J. Fischer, T. H. Kolbe, Automated and transparent model fragmentation for persisting large models, in: MODELS, Vol. 7590 of Lecture Notes in Computer Science, Springer, 2012, pp. 102–118.

[56] M. Scheidgen, M. Schmidt, J. Fischer, Creating and analyzing source code repository models - A model-based approach to mining software repositories, in: Proc. MODELSWARD, SciTePress, 2017, pp. 329–336.

[57] Q. Ma, P. Kelsen, C. Glodt, A generic model decomposition technique and its application to the Eclipse modeling framework, Software and System Modeling 14 (2) (2015) 921–952.

[58] D. Strüber, J. Rubin, G. Taentzer, M. Chechik, Splitting models using information retrieval and model crawling techniques, in: FASE, Vol. 8411 of Lecture Notes in Computer Science, Springer, 2014, pp. 47–62.

[59] J. Espinazo-Pagán, J. G. Molina, Querying large models efficiently, Information & Software Technology 56 (6) (2014) 586–622.

[60] G. Daniel, G. Sunyé, J. Cabot, Mogwaï: A framework to handle complex queries on large models, in: RCIS, IEEE, 2016, pp. 1–12.

[61] G. Bergmann, Z. Ujhelyi, I. Ráth, D. Varró, A graph query language for EMF models, in: Proc. ICMT, Vol. 6707 of Lecture Notes in Computer Science, Springer, 2011, pp. 167–182.

[62] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, G. Wachsmuth, DSL Engineering - Designing, Implementing and Using Domain-Specific Languages, dslbook.org, 2013.
URL `http://www.dslbook.org`

[63] G. Wachsmuth, G. D. P. Konat, E. Visser, Language design with the spoofax language workbench, IEEE Software 31 (5) (2014) 35–43.

[64] C. Debreceni, G. Bergmann, M. Búr, I. Ráth, D. Varró, The MONDO collaboration framework: secure collaborative modeling over existing version control systems, in: Proc. ESEC/FSE, ACM, 2017, pp. 984–988.

Appendix A. Scoped constraints for the evaluation of performance

This appendix contains the scoped constraints used in the experiment of Section 6.2. For completeness, we consider constraints with all kinds of scopes: one

with scope sameProject, three with scope sameRootPkg, two with scope samePkg, and five with scope sameUnit.

Listing 2 shows the constraint with scope sameProject, which controls the number of state machines in the whole model.

```
1 context StateMachine inv numberStateMachines:
2     StateMachine.allInstances()→size() <= 10
```

Listing 2: Scoped constraints with scope sameProject.

Listing 3 shows the invariants with scope sameRootPkg. The first two constraints control the maximum number of instances of ControlSubsystem and Component. The third one validates that there are no more than 5 nested Subsystems (i.e., nested packages of type Subsystem).

```
1  context ControlSubsystem inv numberControlSubsystems:
2      ControlSubsystem.allInstances()→size() <= 10
3
4  context Component inv numberComponents:
5      Component.allInstances()→size() <= 50
6
7  context Subsystem inv depthSubsystem:
8      self.subsystems→forAll(sub1 |
9          sub1.subsystems→forAll(sub2 |
10             sub2.subsystems→forAll(sub3 |
11                 sub3.subsystems→forAll(sub4 |
12                     sub4.subsystems→forAll(sub5 |
13                         sub5.subsystems→size() = 0)))))
```

Listing 3: Scoped constraints with scope sameRootPkg.

Listing 4 shows the invariants with scope samePkg. The first one checks that every subsystem contains a component connected with itself through references inPort and outPort. The last one validates that each Subsystem has at least one component with an input port.

```
1 context Subsystem inv connectedComponents:
2     self.ensembles→collect(connectors)→flatten()→exists(con |
3         Component.allInstances()→exists(comp |
4             comp.ports→includesAll(Set{con.inPort, con.outPort})))
5
6 context Subsystem inv inputPortSubsystem:
7     self.ensembles→collect(elements)→flatten()→exists(c |
8         c.ports→exists(p | p.oclIsTypeOf(InPort)))
```

Listing 4: Scoped constraints with scope samePkg.

Finally, Listing 5 shows the constraints with scope sameUnit. The first two constraints ensure that every StateMachine has exactly one InitialState and at least one SimpleState. The third constraint checks that every SimpleState is reachable from the InitialState. The fourth constraint checks that every Port is connected to another one. The last constraint ensures that each InitialState is connected to some state.

```
1 context StateMachine inv oneInitialState:
2     self.states→one(s | s.oclIsTypeOf(InitialState))
3
4 context StateMachine inv existsSimpleState:
5     self.states→exists(s | s.oclIsTypeOf(SimpleState))
```

```
6
7  context SimpleState inv reachableState:
8      self→closure(incoming.source)→exists(v | v.oclIsTypeOf(InitialState))
9
10 context Port inv connectedPorts:
11     Connector.allInstances()→exists(c |
12         (c.inPort = self and not c.outPort.oclIsUndefined()) or
13         (c.outPort = self and not c.inPort.oclIsUndefined()))
14
15 context InitialState inv initStateIsNotIsolated:
16     self.outgoing→size() >= 1
```

Listing 5: Scoped constraints with scope sameUnit.

Appendix B. Scoped constraints for the case study (CAEX)

This appendix contains the nine scoped constraints used in the case study of Section 6.3. One constraint has scope sameProject, another has scopes samePkg and sameUnit simultaneously, and seven constraints have scope sameUnit.

Listing 6 shows the constraint with scope sameProject, which validates the version of the AutomationML model.

```
1  context CAEXFile inv superiorStandardVersionIsMandatory:
2      self.superiorStandardVersion→exists(v | v = 'AutomationML 3.0')
```

Listing 6: Scoped constraints with scope sameProject.

Listing 7 shows the constraint with two scopes: samePkg and sameUnit. This happens because CAEXObject is a base class from which many other classes inherit, and therefore, its instances can be found in packages and units. The constraint ensures non-empty object identifiers.

```
1  context CAEXObject inv idIsMandatory:
2      self.iD <> null
```

Listing 7: Scoped constraints with scope samePkg and sameUnit.

Finally, Listing 8 shows the constraints with scope sameUnit. The first one checks that the base class of a SystemUnitClass is a SystemUnitClass as well. The second and third constraints validate that InternalElements with a base system unit define, for every attribute in the base system unit, another attribute with the same name and value, and vice versa. The fourth constraint ensures that InternalElements have no base class. The last tree constraints ensure that if an InternalElement contains a requirement with role class name *Process*, *Resource* or *Product*, then, all its internal elements must also define a requirement with an equally named role class.

```
1  context SystemUnitClass inv inheritanceMustPointToSUC:
2      self.baseClass <> null implies self.baseClass.oclIsTypeOf(SystemUnitClass);
3
4  context InternalElement inv strongConformanceSUC2IE:
5      self.baseSystemUnit <> null implies
6          self.baseSystemUnit.attribute→forAll(aC |
7              self.attribute→one(cI |
8                  aC.name = cI.name and aC.value = cI.value));
9
```

```
10 context InternalElement inv strongConformanceIE2SUC:
11     self.baseSystemUnit <> null implies
12         self.attribute→forAll(aI |
13             self.baseSystemUnit.attribute→one(aC |
14                 aC.name = aI.name and aC.value = aI.value));
15
16 context InternalElement inv noInheritanceForIEs:
17     self.baseClass = null;
18
19 context InternalElement inv processContainsProcesses:
20     self.roleRequirements.roleClass.name→exists(r | r = 'Process') implies
21         self.internalElement→forAll(ie |
22             ie.roleRequirements.roleClass.name→exists(r | r = 'Process'));
23
24 context InternalElement inv resourceContainsResources:
25     self.roleRequirements.roleClass.name→exists(r | r = 'Resource') implies
26         self.internalElement→forAll(ie |
27             ie.roleRequirements.roleClass.name→exists(r | r = 'Resource'));
28
29 context InternalElement inv productContainsProducts:
30     self.roleRequirements.roleClass.name→exists(r | r = 'Product') implies
31         self.internalElement→forAll(ie |
32             ie.roleRequirements.roleClass.name→exists(r | r = 'Product'));
```

Listing 8: Scoped constraints with scope sameUnit.