# Multi-Paradigm Architecture Constraint Specification and Configuration Based on Graphs and Feature Models

Sahar Kallel[1,2], Chouki Tibermacine[1], Ahmed Hadj Kacem[2], and Christophe Dony[1]

LIRMM, CNRS and University of Montpellier, France*
ReDCAD, University of Sfax, Tunisia**

**Abstract.** Currently, architecture constraints can be specified and checked in different paradigms of software development, the object-oriented, component-based and service-based one. But the current state of the art and practice do not consider their specification at a high level of abstraction, independently from any paradigm vocabulary. We propose in this paper a process combining graphs and feature modeling to specify multi-paradigm architecture constraints. These constraints are expressed with OCL on a particular meta-model of graphs. Then these constraints can be transformed to any chosen paradigm, after their configuration using a feature/variability model. This transformation allows later to handle these constraints in that (chosen) paradigm: to refine them, to generate source code from them, and to check them on models and on source code. A case study is presented in this paper; it concerns architecture constraint specification and configuration under software migration from the object-oriented to the component-based paradigm.

## 1 Introduction

Documenting software architectures provides a preliminary comprehensive view of the structure and the behavior of the software. This documentation includes the definition of architecture decisions which provide an important element: *Architecture constraints*.

Architecture constraints [11], which are meta-level specifications of invariants on the structure of the entities, constituting a user application (objects for instance), enable to "formalize" the topological/structural conditions imposed by design patterns, architectural styles or any design principle. They are involved throughout the software development life-cycle (from design to implementation stages and in maintenance). Currently, these artifacts can be specified and checked in different programming paradigms: the object-oriented, component-based and service-based one, among others. But constraint specifications in the different paradigms are defined completely separately from each

* {sahar.kallel,tibermacin,dony}@lirmm.fr
** ahmed.hadjkacem@fsegs.rnu.tn

other, while these share a major part of their specification. This part concerns the formalized structural conditions. The variable part between them is the set of architectural entities on which these conditions are checked (objects, object dependencies, components, ports, connectors, services, and so on). For example, in the Façade pattern, the façade entity is an object in an object-oriented application, and what it hides to client entities are the internal methods of the application. In a component-based application, the façade entity is a component which provides a unique port to client entities; it hides the provided services by the other components of the application. The structural conditions here are the same (presence of a unique entity – object or component – which serves client entities).

In our previous works [6,7], we have studied the use of OCL/UML [1] for architecture constraint specification and their checking at the design and implementation stages in different development paradigms. Our first work presented a process which enables to generate meta-programs that make possible constraint checking on object-oriented applications. The second work proposed another process which enables to generate reusable and executable components deployed in component-based applications, in addition to architecture constraints as services which are reusable, searchable, executable and checkable in service-based applications. We propose in this paper an approach (a language and a process) in which architecture constraints are specified in an abstract way, with a neutral structural constraint vocabulary. They are expressed in ocl and navigate in a meta-model of graphs. Then these constraints can be transformed towards a given paradigm (in our case, the object-oriented, component-based and service-based ones) by configuring a feature model. This feature model expresses the commonality and variability between development paradigms. Once constraints are transformed to a given paradigm, they can be checked on models defined in that paradigm, or be refined and transformed into meta-programs (particular classes or component/service descriptors) to be checked on the code of applications.

The remaining of this paper is organized as follows. In the following section, we present the graph meta-model and the feature model used in our approach. Section 3 explains the process of architecture constraint configuration and transformation. A case study is exposed in Section 4. Before concluding, we discuss the related work in Section 5.

## 2 Architecture Constraint Specification and Configuration

We define in the first subsection a meta-model of graphs on which an example of an architecture constraint is specified. In the second subsection, we present the feature model used for the configuration of constraints.

[1] OCL/UML means that the constraints are specified with OCL and navigate in the UML meta-model.

### 2.1 A Meta-model of Graphs

As an underlying software representation we use graphs because they can capture the basic structure in a straightforward and generic way: nodes represent software entities and edges represent relationships between those entities. More precisely, we have used `typed`, `directed` and `labeled` graphs. We have used a `typed` graph to specify that nodes can be nested in other nodes. We have used `directed` graphs, implying that each edge has a source and a target node (directed dependencies between software entities) and `labeled` graphs to attach any number of domain-specific properties to the nodes and edges.

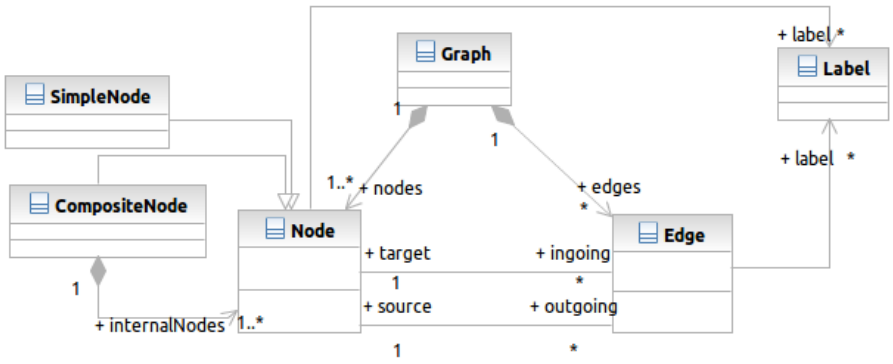


**Fig. 1.** Meta-model of graphs

Fig. 1 shows the meta-model of graphs used in our approach. A graph is composed of edges and nodes. A node has at least one outgoing and ingoing edge. Each edge has exactly one source node and one target node. A node can be composite or simple. The composite node can be composed of simple and also composite nodes. Each node and edge can be labeled in order to refine the graph. According to this meta-model, we can obtain a model (a graph) that contains edges going from inside a composite node to a simple one.

Listing 1.1 presents an architecture constraint characterizing the `Façade` pattern. This constraint is formalized with OCL navigating in the meta-model shown in Fig. 1. It consists of several sub-constraints. We suppose that there exist a set of nodes that represent `clients`, another set represents `systems` and a node represents a `facade`.

```
1  context Graph inv :
2  --Clients have only outgoing edges
3  clients->forAll(n:Node|n.ingoing->isEmpty())
4  and
5  --Systems have only ingoing edges
6  systems->forAll(n:Node|n.outgoing->isEmpty())
7  and
8  --No edges between clients and systems
9  clients->forAll(n:Node|n.outgoing->forAll(e:Edge|
10  systems->excludes(e.target)))
```

```
11 and
12 --All the edges whose sources are the clients should go to the facade
13 clients->forAll(n:Node|n.outgoing->forAll(e:Edge|e.target=facade))
14 and
15 --The facade should be linked to at least one system
16 facade.outgoing->exists(e:Edge|systems->includes(e.target))
```

**Listing 1.1.** Facade constraint specification in the graph meta-model

These node labeled *Client*, *System*, *Facade* may give a hint about constraint semantic but it is not clear that these nodes represent objects, components or classes. At this level, we can say that the constraint is formalized in an abstract way, *i.e.* independently from any paradigm. To translate the constraints into a specific paradigm, we have to configure a feature model which is presented in the following section.

### 2.2 Feature models

Feature models [8] are simple and hierarchical models that capture the commonality and variability of a set of products in a software product line. In our approach, a feature model is used to express the variability between software development paradigms.

A feature diagram is a representation of a feature model. We have used the notation of Czarnecki *et al.* [5] in the feature diagram developed for our approach because it is a practical way to integrate labels for the nodes and the edges. It is a useful way to configure the constraint which navigates, among others, in `Label`, `Node` and `Edge` meta-classes of the graph meta-model. Moreover, an architecture constraint is generally composed of sub-constraints assembled by the logic operator "and" (see Listing 1.1). In each sub-constraint, we find several (0-n) nodes and/or edges. Each node or edge can be translated to the appropriate element in the chosen paradigm (class, method, connector, port, object, etc). For doing so, we added a cardinality to the feature diagram in order to be able to do all the required transformation for each sub-constraint and configure each node and each edge.

Since this work is a continuation of our previous works (introduced in the previous section), we have chosen to translate ocl architecture constraints from a graph meta-model to UML [2] meta-model based on feature models. Therefore, the features (without considering the leaves) represent, among others, the meta-classes (Ex: Graph, Node, Edge) and the meta-roles (ex: source and target) of the meta-model of graphs, while the leaves of the feature diagram are elements of the UML meta-model (Fig. 2[3]).

The root feature of the diagram is the graph representing the architecture on which the constraint is formalized. The feature *Node* is a sub-feature of *EltGraph* (i.e.,*Node* is a child of *EltGraph* in the feature tree) and has an attribute for specifying its label, if any. Every feature is qualified by a feature cardinality. It

[2] UML `http://www.omg.org/spec/UML/2.4.1` is an OMG standard and covers both class/object and component modeling

[3] For space limitation, the constraints accompanied the feature diagram are not showed

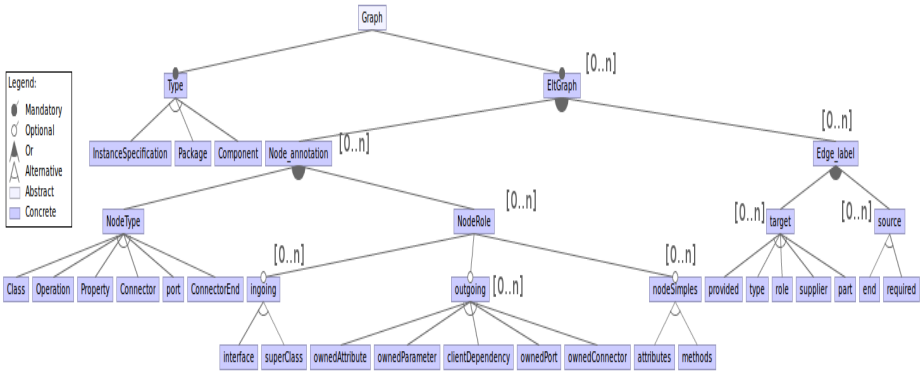


**Fig. 2.** An excerpt of the feature diagram for constraint transformation from graphs

specifies how often the entire sub-tree rooted in the solitary feature can be copied (with the roots of the replicated sub-trees becoming siblings). For example, the features *Node* and Edge have the feature cardinality [0..n]. This means that an *EltGraph* can be formed by 0 or n *Node*s and *Edge*s.

## 3 Multi-paradigm Architecture Constraints

In this section, we present the different steps of the constraint transformation. We use the `Façade` architecture constraint shown in Listing 1.1 as a running example.

### 3.1 Constraint Configuration

Constraint configuration consists in selecting, in the feature diagram, the suitable features to build a new constraint in the chosen paradigm. This step is started by configuring first the context of the constraint, then configuring the OCL definitions [4] and OCL let expressions, if any, and finally the sub-constraints by respecting their appearance order in the constraint. A step called *feature model specialization* [5] is performed before the configuration. It consists in choosing the precise values of cardinalities presented in the feature diagram. This facilitates the configuration of the constraint by reserving the exact number of features in the configuration interface.

Each sub-constraint, including the OCL let expression, is represented by *EltGraph*. In our constraint, we have 8 *EltGraph*s. We can configure all these *EltGraph*s thanks to the cardinality of this feature. We follow the order of the sub-constraints to configure them. Fig. 3 presents a possible configuration of the sub-constraint 5 (in Listing 1.1 without considering the let expressions) in the object-oriented development paradigm.

[4] OCL queries characterized by the keyword `def:`. They allow to declare and define attribute values (like let expression) and/or to return internal OCL operation values.

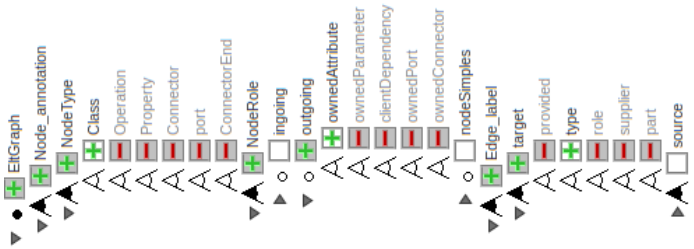


**Fig. 3.** A possible configuration of Facade constraint in OO paradigm

The constraint configuration is performed using the feature IDE plugin. It shows an interface to configure a feature diagram. We can see all the possible configurations and it produces exceptions if the configuration does not respect the requirements of the feature diagram.

### 3.2 Constraint Transformation

The implementation of the constraint transformation is performed using the editor of the feature model. The first step is a direct transformation of the constraint. It uses the configured feature model. The second step is based on the abstract syntax tree (AST) generated from the obtained constraint and the XMI document representing the UML meta-model. It has as a goal to make the constraint valid. The tool-set used for configuring the feature diagram provides a document that includes the inputs and the outputs of the configuration (names of features). Our process uses this document and automatically applies the mapping to the constraint. An abstract syntax tree is generated from the constraint (which is specified in the graph meta-model). The AST node names and their types (the meta-class names) are then modified by their corresponding features and are regenerated in order to obtain an architecture constraint written in the UML meta-model.

Listing 1.2 presents the `façade` constraint after the direct transformation. The meta-role *outgoing* in Line 9 in Listing 1.1 is replaced by *clientDependency* and in Line 16 by *ownedAttribute* as the configuration is defined (see Fig. 3).

```
1 context Package inv :
2 --
3 clients->forAll(n:Class|n.->isEmpty())  and
4 systems->forAll(n:Class|n.->isEmpty())  and
5 clients->forAll(n:Class|n.clientDependency->forAll(e:Edge|
6   systems->excludes(e.supplier))) and .... and
7 facade.ownedAttribute->exists(e:Edge|systems
8  ->includes(e.type))
```

**Listing 1.2.** An excerpt from a Facade constraint after a direct transformation

In Listing 1.2, the constraint is specified in UML meta-model, but this transformation does not necessarily produce a valid OCL constraint. OCL exceptions

are provided when compiling the constraint in an OCL compiler. For example *Edge* in Line 5 is undefined in UML meta-model. The two following sub-steps are implemented to solve these errors.

***1. Removing unnecessary sub-constraints*** : This is the case of the sub-constraints 1 and 2 in Listing 1.2. The user does not completely configure the sub-constraints. They do not have any equivalence in the target paradigm: the object-oriented paradigm. These sub-constraints are safely removed from the constraint.

***2. Adding OCL expressions*** : There are two cases where we should add OCL expressions. The process here examines the constraint in each case and try to add OCL expressions to make it valid and accurate.

The process in the first case consists first in replacing all the roles and meta-classes that are still written in the graph meta-model by their corresponding modeling elements in the UML meta-model. This transformation is complementary to the direct one. It is based on the AST generated from the constraint. The AST parser, taking into consideration the UML meta-model, indicates the AST nodes which whose types do not belong to the UML meta-model. We take the example presented in Line 5, in Listing 1.2 in which the meta-class `Edge` is not translated yet. According to the UML meta-model, *clientDependency* is a navigation that produces `Set(Dependency)`. So, `Edge` will be replaced by `Dependency`. The same processing is performed for the error located in Line 8 in the same Listing: `Edge` is replaced by `Property`.

The process in the second case consists in adding navigation patterns [5] in the constraint. Indeed, after the direct transformation, we can obtain in a sub-constraint an ocl inequality exception. Suppose that we take an example of a constraint that has, in its specification in the graph meta-model, a navigation towards the *Node* meta-class via *target*, to get the target node (one node [1..1]) (see Listing 1.3). The user configures *target* by *end* in the component-based development paradigm. *end* is a meta-role in the UML meta-model. It provides a set [0..*] of component connectors. So, we face an OCL exception `(Set(Connectors)= a component)`. Here, the process adds, among others, an appropriate quantifier that takes only one of the sets to complete the constraint transformation. More details are given in the following Listings.

In the first line of Listing 1.3, X and Y are nodes composing the graph of the model. The constraint imposes that the node X should have at least one outgoing edge towards the node Y. To transform this constraint in the component paradigm, the user configured *outgoing* by *ownedPort* and *target* by *end*. The process checks if the constraint has again errors of the first case. The second line represents the constraint specification under the transformation. We observe that the specification of this constraint is wrong. It is violated when evaluating it in the UML meta-model. To solve this problem, we integrate first some meta-roles

[5] A navigation pattern is a set of navigations. It includes more roles and ocl operations/quantifiers.

such as `ownedConnector` and `role` (an application of the first case) and then pattern navigations as presented in Listing 1.4. This Listing shows a possible result.

```
1 X.outgoing->exists(e:Edge|e.target=Y)
2 X.ownedPort->exists(e:Port|e.end = Y)
```

**Listing 1.3.** OCL AC before and after direst transformation

```
1 X.ownedPort->exists(e:Port|e.ownedConnector.end->
2     exists(ee:ConnectorEnd | ee.role -> includes(Y.role)) )
```

**Listing 1.4.** OCL AC specified in the UML meta-model

As we noticed above, the implementation of the process that consists in making the architecture constraint independent to any paradigm uses an Eclipse tool-set. This tool-set generates the abstract syntax tree (AST) and analyzes the UML meta-model. Each output (sub-constraint) provided by this process should be validated by the user.

## 4 Case Study

We have applied the proposed approach on a particular engineering activity: the automatic software migration from the object-oriented paradigm to the component-based one one. In this kind of activities, it is too difficult to directly specify the architecture constraint in the transformed application (component-based application) because many constraints imposed by the initial application (like, inheritance and instantiation) may generate other constraints (new architectural patterns are added under the migration, which are not known by the user, especially if the migration is automatic) and new architectural elements (connectors and ports) which can impose new architecture constraints.

We take the example of an object-oriented application which is designed with UML and implemented with java, and which represents an *information screen* [2]. This application simulates the behavior of an information screen, a software system which displays in a public transportation's embedded screen, the names of stations, the expected time at each station, etc. The *ContentProvider* class implements methods which send text messages (instances of the *Message* class), and time information obtained through *Clock* instances based on the data returned by *TimeZone* instances. The *DisplayManager* is responsible for viewing the provided information through a *Screen*. The design of this application imposed a set of architecture constraints that should be valid on the code. Some of these constraints are presented in the following list.

- *ContentProvider* class should be a singleton class.
- *Clock* and *Message* classes should be kept in relation with the *Content* abstract class (which is an inheritance relation in the OO application).
- The Observer pattern is instantiated in this application. We focus in this case study on a part of this pattern, in which *DisplayManager* class should be in association with *ContentProvider* to invoke methods returning the content.

When migrating an application, major changes of the architecture and then the source code are performed. Some elements are removed, others are added, *e.g.* dependencies between some elements are changed, etc. In fact, each paradigm imposes its own architecture design principles. For example, in the component-based paradigm, each component must hide its internal structure. It should provide its services without exposing the classes that implement them. These conditions should be taken into consideration. In addition, the works cited previously proposed an automatic migration of the applications, which generally produces additional intermediate classes, methods and components in addition to dependencies between them, which are seamless to developers. In this case, rewriting the constraints in the target paradigm is difficult because architectural elements constituting the target application can be unknown.

Our intuition is that our approach can allow to simplify the migration of the architecture constraints of information screen object-oriented application in component-based paradigm. To apply our approach, we have used software migration works that are composed of two steps: architecture recovery then code transformation. These works generate automatically a graph describing the architecture of the target application. This graph contains labeled nodes that may represent the classes, the methods, the attributes and the components representing clusters of cohesive classes, in addition to edges that link between nodes (method invocations, connectors between required/provided interfaces, etc.). Besides, to make component interfaces operational, the graph is extended by other nodes and edges that represent new classes, interfaces and attributes that are generated to transform inheritance into the component-based paradigm [2]. Fig 4. shows an excerpt of this graph.

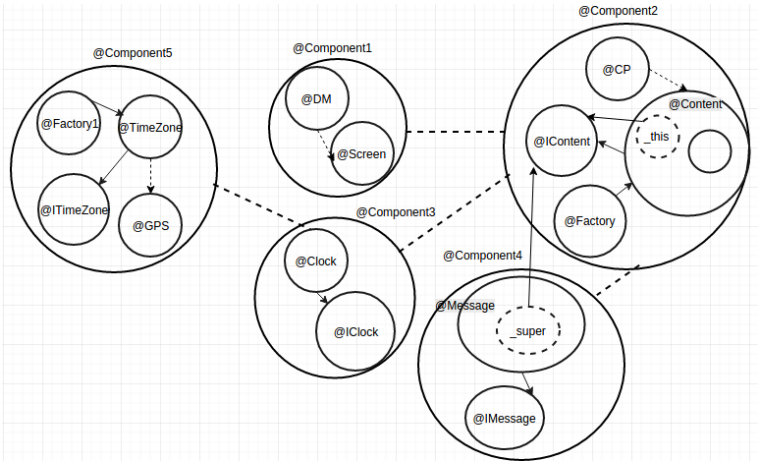


**Fig. 4.** An excerpt of a graph representing the architecture recovered from the Information Screen application

Based on this graph which contains architecture elements of the source application and also new elements added by the migration, we have rewritten the architecture constraints of the application. It is specified in the meta-model of graphs shown in Fig. 1. For reasons of space limitation, Listing 1.5 presents only an excerpt of this constraint.

```
1  context Graph inv :
2  let compo1:Set(CompositeNode)=self.nodes->select(n:Node|n.labels
3    ->exists(a:Label|a.name='Component1')) in
4  -- the same for compo2, compo3, compo4 and compo5
5  let content:Node= compo2.simpleNodes->select(n:Node|n.labels
6      ->exists(a:Label|a.name='Content'))->asOrderedSet()->first() in
7  -- other let expressions ...
8  in
9  compo1.outgoing->one(e:Edge|e.target=compo2)  and
10 compo2.ingoing->forAll(e1,e2|e1.source=compo3 and e2.source=compo4)
11 and ... and
12 content.outgoing->exists(e:Edge|e.target=iContent) and
13 factory.ingoing->one(e:Edge|e.source=content) and
14 content.simpleNodes->select(n|n.outgoing->exists(e|e.target=iContent))
15 and ... and
16 message.simpleNodes->select(n:Node|n.outgoing
17   ->exists(e:Edge|e.target=iContent))
```

**Listing 1.5.** An excerpt of AC specification in graph meta-model

In this constraint, the let expressions search for the elements composing the application, and which can be classes or components. `compo1` is an example of a variable which references the node named `Component1`. This component was identified in the architecture recovery step; it is considered in this constraint as a graph's node.

### 4.1 Configuring the constraint by the feature model

There are nodes that represent classes (annotated by CP, Factory, Content and TimeZone), attributes (dashed nodes in Fig 4), components (annotated by Componenti, i=[1..5]), etc. There are edges that represent connectors (thick dashed edges), others represent inheritance (between classes inside components). There are other nodes which are generated due to solutions kept to transform the instantiation and inheritance. Some of these nodes are annotated with *IContent*, *Factory*, *_this*, *_super*, *ITimeZone*. There are also edges which link them. These elements did not exist in the architecture of the source application (object-oriented information screen application). They imposed a new condition that consists in respecting the factory pattern (which is instantiated in the architecture when transforming an inheritance relation in the chosen migration solution in this case study). The constraint will be configured in our feature model starting by the first sub-constraint and so on as described in Section 3. We indicate for each element its equivalent in the new architecture.

### 4.2 Transforming the constraint

Following the process explained in Section 3 by using the configured feature model of our constraint and after making the constraint well specified in the UML meta-model, we obtain as an excerpt of a result the following Listing.

```
1  context Component inv :
2  let internalCompo : Set(Component) ... in
3  let compo1: Component=internalCompo->select(n:Component|
4   c.name='Component1')->asOrderedSet()->first()   in
5  -- the other let expressions ... in
6  compo1.ownedPort->one(e:Port|e.ownedConnector.end
7    ->forAll(ee:ConnectorEnd|ee.role->includes(compo2.role))) and
8  compo2.ownedPort.ownedConnector->forAll(e1,e2|
9    compo3.role->includes(e1.end.role) and compo4.role
10      ->includes(e2.end.role))    and
11 content.interface->exists(e:Interface|e.name=iContent) and
12 factory.ownedAttribute->one(e:Property|e.type=content)
```

**Listing 1.6.** An excerpt of AC specification in UML meta-model (Component modeling)

This constraint declares first the internal component which composes the target application. This implies modifications in the let expressions like in Line 3. According to the configured feature diagram, the sub-constraints 1 and 2 (Lines 6 to 10) handle the relations between the generated components, and the remaining of the constraint deals with classes. Indeed, the migration solution used in this case study produces a component-based application in which components are clusters of classes (a hybrid object/component target model). This is the reason why the end of the architecture constraint in the Listing still treats classes. This makes this example a multi-paradigm architecture constraint.

***Discussion:*** The migration of the object-oriented information screen application has produced new architecture elements and new architecture relations. This is observable (in Fig. 4) by the production of 5 components, 6 classes and several attributes. Therefore, a direct transformation of the application's constraints is obviously very complex because they do not treat the newly created architectural elements. After specifying the constraints of the target application in the graph meta-model, based on the generated graph from the architecture recovery step, which should be done only once, the user can transform the constraints after a simple configuration of the feature model. To migrate the application to another paradigm, such as the service-oriented one, with the proposed approach the developer can just configure again the feature model to transform her/his constraints.

In addition, the usage of the graph meta-modeling and the feature model facilitate constraint specification at an abstract level. In the long term, we imagine the development of a catalog of architecture constraints written in the graph meta-model. This catalog can be used in different scenarios. Suppose that we use another software migration solution, like [1], which transforms inheritance and instantiation from object-oriented to component-based paradigm by using the *Adapter* and *Facade* patterns, in contrast to the one used in this case study that is based on the *Factory* pattern. The architecture constraints formalizing these two patterns (*Adapter* and *Facade*) can be checked out from the catalog, then configured (by adding the necessary labels) and at last integrated in the architecture constraint specification of the application.

## 5 Related Work

Vranic *et al.* proposed a method of multi-paradigm software development called multi-paradigm design with feature modeling (MPDFM) [13]. Feature modeling is used to model both an application and the solution domain. Solution domain concepts (paradigms) are represented as features. These later (called paradigms) are being selected in the feature model in order to obtain code skeleton. This method is evaluated on the AspectJ paradigm as a solution domain. Like our approach, this method uses feature modeling to express variabilities between paradigm instances. But the term paradigm denotes a solution domain concept, which corresponds to a programming language mechanism/extension. In our approach, we used the common definition of a paradigm – a way of development. This covers a larger spectrum.

Balarin *et al.* proposed a formalism for constraint specification at higher levels of abstraction [3]. This formalism use mathematical theorems to remove any ambiguity in its interpretation, and yet it allows quite simple and natural specification of many typical constraints. In our work, we have proposed an abstract specification level of constraints based on graphs. With graphs, we can benefit from a visualization that simplifies the comprehensibility of any kind of constraints. Constraint specification with graphs allows later transformation, refinement and code generation which is very complex when using a pure mathematical formalism.

ACL [12] is a family of languages which allows the specification of constraints associated to architecture decisions, at any stage of the component-based software development process. Independently to any component-based model, architecture constraints can be specified with this language. The authors proposed a generic meta-model that includes the common concepts found in existing component models. This meta-model can be used to specify these constraints, which are independent from component models. Then, through XML transformations, constraints can be checked on a precise component model, like Corba. In contrast to our work, this work deals with the component-based software development paradigm only, and not the other paradigms. Their generic meta-model includes common concepts in component-models and not variable concepts. In our work, thanks to feature modeling, we specified common and variable concepts in development paradigms and used this in constraint transformation.

Many works [4,10,9] handle the specification of constraints with graphs. These works share the same context as our approach but their goal is different from ours. They focus on, among authors, formalizing semantics in UML models and transformations using ocl, verifying them on models. But no one considers ocl **architecture** constraint specification. To the best of our knowledge, there is no work that enables to make architecture constraints specified independently to the paradigm used in the application development.

## 6 Conclusion

We presented in this paper an approach that enables the specification of multi-paradigm architecture constraints. These constraints are written in an abstract way independently from any paradigm. The key idea is to combine the usage of OCL with a graph metamodel, and a feature model to implement our method. The meta-model of graphs is used to specify the constraints and the feature model is exploited to express paradigm variabilities. The constraints can be translated to any specific paradigm, simply through the configuration of the feature model.

As a future work, we plan to provide a way to express architecture constraints at (yet) a more abstract level, with a natural language syntax, and then combine it with this work and our previous approaches to provide a complete process. A transformation method should be developed to transform the architecture constraint specification from natural language into graph-based specification and then into UML-based one, until source code generation according to a specific paradigm. This will make the architecture constraint specification simpler, yet keep it operational (checkable on source code and at runtime).

## References


1. Allier, S., et al.: From object-oriented applications to component-oriented applications via component-oriented architecture. In: WICSA. pp. 214–223. IEEE (2011)
2. Alshara, Z., et al.: Migrating large object-oriented applications into component-based ones. In: ACM SIGPLAN Notices. pp. 55–64. No. 3, ACM (2015)
3. Balarin, F., et al.: Constraints specification at higher levels of abstraction. In: HLDVT Workshop. pp. 129–133. IEEE (2001)
4. BAUER, E.: Enhancing the dynamic meta modeling formalism and its eclipse-based tool support with attributes, bachelor thesis. University of Paderborn (2008)
5. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice 10(2), 143–169 (2005)
6. Kallel, S., et al.: Automatic translation of ocl meta-level constraints into java meta-programs. In: Proc. of SERA, Hammamet, Tunisia. Springer (May 2015)
7. Kallel, S., et al.: Generating reusable, searchable and executable "architecture constraints as services". Journal of Systems and Software 127, 91–108 (2017)
8. Pohl, K., et al.: Software product line engineering: foundations, principles and techniques. Springer Science & Business Media (2005)
9. Radke, H., et al.: Translating essential ocl invariants to nested graph constraints focusing on set operations: Long version. ICGT (2015)
10. Rutle, A., et al.: A formal approach to the specification and transformation of constraints in mde. The Journal of Logic and Algebraic Programming 81(4), 422–457 (2012)
11. Tibermacine, C.: Architecture constraints. Software Architecture 2 pp. 37–90 (2014)
12. Tibermacine, C., et al.: A family of languages for architecture constraint specification. Journal of Systems and Softwre 83(5), 815–831 (2010)
13. Vranić, V.: Multi-paradigm design with feature modeling. Computer Science and Information Systems 2(1), 79–102 (2005)