Extending the Common Variability Language (CVL) Engine: A practical tool

Jose-Miguel Horcas
Universidad de Málaga
Andalucía Tech, Málaga, Spain
horcas@lcc.uma.es

Mónica Pinto
Universidad de Málaga
Andalucía Tech, Málaga, Spain
pinto@lcc.uma.es

Lidia Fuentes
Universidad de Málaga
Andalucía Tech, Málaga, Spain
lff@lcc.uma.es

ABSTRACT

The Common Variability Language (CVL) has become a reference in the specification and resolution of variability in the last few years. Despite the multiple advantages of CVL (orthogonal variability, architecture variability resolution, MOF-compliant, standard proposed,. . .), several approaches require extending and/or modifying the CVL approach in different ways in order to fulfill the industrial needs for variability modeling in Software Product Lines. However, the community lacks a tool that would enable proposed extensions and the integration of novel approaches to be put into practice. Existing tools that provide support for CVL are incomplete or are mainly focused on the variability model's editor, instead of executing the resolution of the variability over the base models. Moreover, there is no API that allows direct interaction with the CVL engine to extend or use it in an independent application. In this paper, we identify the extension points of the CVL approach with the goal of making the CVL engine more flexible, and to help software architects in the task of resolving the variability of their products. The practical tool presented here is a working implementation of the CVL engine, that can be extended through a proposed API.

KEYWORDS

CVL, Software Product Line, Variability

1 INTRODUCTION

The Common Variability Language (CVL) [6] has become one of the most used languages for specifying and resolving architecture variability in the last few years. CVL has displaced traditional feature modeling for managing the variability in Software Product Lines (SPLs) approaches [10], especially for architecture-centric approaches. This is partly due to several advantages that CVL has over other variability modeling languages (e.g., orthogonal variability, architecture variability resolution, MOF compliant, several levels of abstraction, units of modularization, complete definition of clonable and variable features).

Despite the great flexibility of the CVL language, several approaches [4, 7, 9, 11] have had to extend and/or modify CVL in different ways to fulfill the industrial needs of variability modeling in SPLs. For instance, CVL has been upgraded to the BVR (Base, Variability, Resolution models) language [7] by modifying the CVL metamodel, where some constructs have been removed for the sake of simplicity and other new constructs have been added for better and more suitable expressiveness. Other approaches have changed the materialization of the variation points to model the variability directly over code (e.g., Java programs) [4]; or have defined new semantics for the variation points to apply more complex model transformations to the products [9]. Moreover, some approaches may require having to modify the CVL execution engine to include additional steps before the materialization process, for example, to include an identification process of the points in the base models where the variation points can be safety applied [11].

However, the CVL community lacks tools that first, fully support the CVL standard[1], and second, allow the different extensions of the language to be integrated. Although new CVL tools have recently appeared [2, 13], they do not provide the flexibility required by the different approaches that use CVL. For example, the new BVR Tool [13] implements and provides support for the BVR language (an evolution of CVL), enabling feature modeling, resolution, realization and derivation of products, among other features. However, the tool only allows the materialization of the variability through substitutions of parts of the base models (i.e., fragments substitutions), and custom transformations defined by the user are not supported. Moreover, there is no API for CVL that allows direct interaction with the CVL engine to use or extend it in an independent application.

In this paper, we identify the points in the CVL approach where it can be extended without losing the main principles of the CVL standard for specifying and resolving variability. This is done with the goal of increasing the language's flexibility and helping its users in the task of resolving the variability in their products. To illustrate this, we provide a practical tool as a real implementation of the CVL engine, that can be easily extended with different execution engines

[1] CVL was approved by the OMG for standardization, but the process is currently frozen by legal issues.

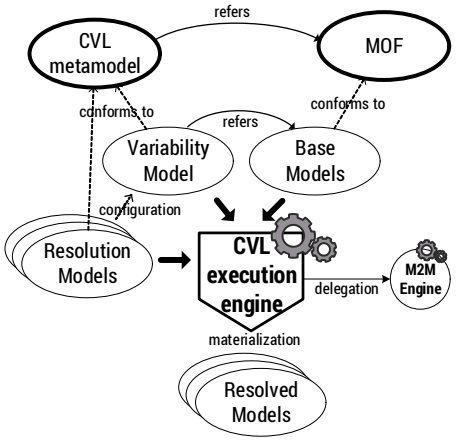

Figure 1: CVL approach.

— e.g., using general purpose model transformation languages such as the ATL Transformation Language.

The remainder of the paper is organized as follows. Section 2 introduces the CVL approach and motivates our proposed tool. In Section 3 we identify the extension points in the CVL approach and the existing approaches that use these points for their own purposes. Section 4 presents our proposed API of the CVL engine. Finally, Section 5 presents the conclusions and future work.

2 BACKGROUND AND MOTIVATION

In this section we present the CVL approach and motivate our proposal comparing our proposed tool with existing CVL tools.

2.1 The CVL approach

An overview of the CVL approach can be seen in Fig. 1. CVL specifies, in separate models, the variability that can be applied to a base model. The *base model* is a model in the domain language that can be defined using a MOF-based metamodel and does not contain any information about variability. The variability information is separately specified in a *variability model*, according to the CVL metamodel. How the variability model can be resolved to produce a configured new model from the base model is described in the *resolution models* (or "configuration models" in the SPL terminology). CVL provides an executable engine to automatically (*materialize*) a *resolved model*. The resolved model is a fully configured product model with the variability resolved.

In CVL, the variability model consists of two main parts: (1) an abstract level with *variability specifications* (*VSpecs tree*), and (2) a concrete level with *variation points* (*VPs*). VSpecs are tree structures representing choices ("features" in most SPL terminologies) and can include logical constraints defined in a subset of the Object Constraint Language (OCL). There are different types of VSpec: *Choices* (yes/no decision), *Variables* (attributes), *VClassifiers* (clonable features), *Composite VSpecs* (modularization units). Variation points define the points of the base model that are variable and can be modified during the materialization process. Variation points also specify how the elements of the base model are modified by defining specific modifications to be applied by means of model-to-model (M2M) transformations. The semantics of these transformations is specific to the kind of variation point. Some of the variation points supported by CVL are the existence of elements of the base model (*ObjectExistence*) or the links between them (*LinkExistence*), the assignment of an attribute's value (*ParametricSlotAssignment*), or the replacement of a set of elements with another set of elements (*FragmentSubstitution*). An important type of variation point is the *Opaque Variation Point* (*OVP*) that enables defining new custom model transformations that are not pre-defined in CVL.[2] During CVL's execution, the CVL engine delegates its control to an M2M engine in charge of executing the transformations defined by the variation points (as shown in Fig. 1).

Table 1: Comparative between the existing CVL tools.

Characteristic	MoSIS CVL	CVL 2	BVR	KCVL	Our tool
Graphical editors for the variability model	■	■	■	□	□
Support OCL cross-tree constraints	■	□	■	■	■
Materialization process	■	□	■	■	■
Support for custom transformations (OVPs)	□	□	□	□	■
CVL metamodel available	■	■	■	□	■
CVL Engine API	□	□	□	□	■

■ It supports the characteristic. □ It does not support the characteristic.

2.2 Motivation

Some approaches concentrate on the evolution of the CVL models to automatize the management of the models, or on improving the CVL language or extending it to their needs. For example, in [5] the semantics of the CVL's variation points is customized, using Kermeta, according to the semantics of the base model domain. In [4], the variation points of CVL are implemented as source code transformations to apply them directly to Java source code. In [8] and [9], the authors define custom model transformations to weave functionality related to quality attributes (e.g., security) into software architectures. Finally, Degueule et al. [2] present a variability solution, connected to the pattern technology and based on an extension of CVL, to support the derivation of model-based architectural variants. They have also developed KCVL, a custom-made implementation of CVL augmented with specific concepts needed for realizing patterns.

In [3], the authors present a usability evaluation of CVL applied to a modeling tool for firmware code. They conclude that model configuration in terms of model fragment substitutions is intuitive enough to materialize the variability. However, as discussed, there are several approaches for which the substitution of fragments alone is not enough to cover their needs [2, 5, 8, 9]. All these approaches have in common the need for a tool that fully supports the CVL standard and that allows customizing/extending the CVL approach to their needs, in order to make the approaches viable.

There have been several attempts to implement a complete CVL tool. Most of the existing tools belong to the SINTEF[3] research organization:

MoSIS CVL Tool [12]. It is a prototype implementation of the CVL specification based on Eclipse Modeling Framework (EMF) and supports any DSL defined through EMF. However, this tool is out-of-date and is no longer under active development.[4]

[2] The complete taxonomy of variation points of CVL is available in [1].
[3] http://www.sintef.no/
[4] http://www.omgwiki.org/variability/doku.php?id=cvl_tool_from_sintef

CVL 2 Tool. It is also a prototype tool for the CVL standard that is in its infancy, released as an early demo. The tool is an independent Java application, but only supports the specification of the variability and resolution models, and does not support the definition of variation points nor the materialization process.[5]

BVR Tool Bundle [13]. It is a prototype tool which implements and provides support for the BVR language. It consists of a set of Eclipse plug-ins that enable feature modeling, resolution, realization and derivation of products, their testing and analysis. Although this tool is up to date and despite the great support provided for both BVR and CVL, the tool still does not support the definition of custom transformations through the use of Opaque Variation Points (OVPs). In fact, the materialization of the variability is done only through the definition of Fragment Substitution variation points.[6]

KCVL [2]. KCVL is another prototype implementation of CVL, released as a set of Eclipse plugins. KCVL comes with a textual editor for expressing variability abstraction models, variability realization models and resolution models. Like the other tools, KCVL does not support all the taxonomy of variation points defined in the CVL standard, as for example the OVPs for custom transformations.[7]

A comparison of CVL existing tools is summarized in Table 1. The comparison focuses on those features that facilitate the use or extension of the CVL approach by software architects. As shown in Table 1, existing tools focus on the variability abstraction by providing graphical tools for the definition of the variability and resolution models. In our case, we do not provide a graphical editor for the models, but they can be created by using the CVL modeling tools in Eclipse like Sirius.[8] Regarding the materialization process, almost all tools (except the CVL 2 tool), implement it in order to resolve the variability. However, only our tool supports the customization of the variation points semantics, in addition to the definition of custom transformations using Opaque Variation Points. Another important motivation for implementing a tool able to extend CVL is making available the CVL metamodel, so that architects and developers can modify it according to their needs. Our tool provides a CVL metamodel as an ecore file specified in EMF. This metamodel allows creating instances of the variability and resolution models that include custom transformations for the variation points. Finally, we also provide a working implementation of the CVL Engine that can be used in third party applications (e.g., at runtime to implement Dynamic SPLs), independently from the CVL tools, and an API to extend our CVL engine.

3 EXTENSION POINTS OF CVL

In Figure 2 we identify the points in the CVL approach where CVL can be potentially extended, and include examples of these extensions. Basically, there are three ways in which the CVL approach can be naturally extended — i.e., extended without losing the main principles of the CVL standard for specifying and resolving variability. These are (1) extending the CVL metamodel, (2) extending the materialization process, and (3) extending the delegation engine.

[5]http://modelbased.net/tools/cvl-2-tool/
[6]http://modelbased.net/tools/bvr-tool/
[7]https://diverse-project.github.io/kcvl/
[8]https://eclipse.org/sirius/

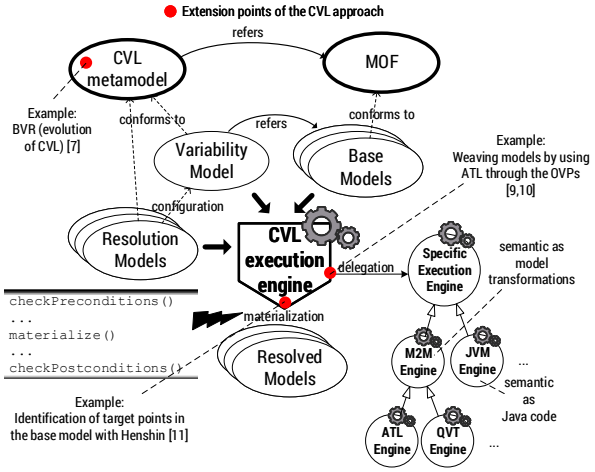

Figure 2: Extension points of CVL.

3.1 Extending the metamodel

Extending the CVL metamodel means the CVL language can be modified directly. More specifically, this allows modifying the variability and the resolution models by defining new constructs to specify more complex variability relationships, and adding new kinds of variation points. An example of this kind of extension is BVR [7] where some constructs have been removed to improve the simplicity of the variability model and some new constructs have been added for better and more suitable expressiveness such as the concepts of targets, resolution literals, and staged variation points [7].

3.2 Extending the materialization process

Another point of extension in CVL is the process of transforming a base model into a configured product model. This implies having to modify the CVL engine to extend or adapt the materialization process to the SPL Engineer's needs. For example, as Figure 2 shows, it is possible to include additional steps before and/or after the materialization of the variability to check the validity of references to the base model, or to include a process that identifies points in base models where the variation points can be safety applied, as is done in [11] for weaving patterns using the Henshin language.

3.3 Extending the delegation engine

When CVL is executed, it delegates its control to an M2M engine in charge of executing the transformations defined by the variation points. By extending this delegation mechanism it is possible to modify the way variation points are realized over the base model. The most basic modification is to use a new M2M transformation engine that supports different transformation languages such as ATL [8, 10], QVT (Query/View/Transformation), or Epsilon Transformation Language (ETL). However, it is also possible to execute code instead of model transformations to resolve variability [4].

4 CUSTOMIZABLE CVL EXECUTION ENGINE

Using the extension points identified in Section 3, we have developed a customizable execution engine of CVL that fully supports the materialization process, including the delegation engine.

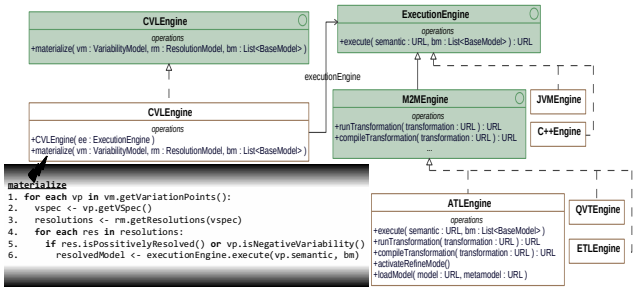

Figure 3: Class diagram of the proposed API of CVL.

Figure 3 represents the main classes and interfaces of the proposed API of our customizable CVL engine.[9] The `CVLEngine` class defines the skeleton of the materialization process in an operation (`materialize`), deferring some additional steps to subclasses following the *Template Method* design pattern. The `materialize` method completely resolves the variability as specified in the CVL standard, and lets subclasses define certain steps such as the checking of pre and post-conditions [11] without changing the algorithm's structure, as Figure 2 shows at the materialization point. The `CVLEngine` delegates its execution to an execution engine (`ExecutionEngine` interface) in charge of executing the semantics specified in the variation points. This `ExecutionEngine` can be an M2M transformation engine (`M2MEngine`) or any other implementation (e.g., `JVMEngine` for Java [4], `C++Engine` for C++, etc.) that performs the modifications defined in the variation points over the base model.

The custom semantics of the variation points is specified in independent files according to the specific M2M transformation language, and are introduced into the CVL Engine as part of the variability model by using references. This means that we can change the semantics of any variation point if necessary, in addition to defining custom semantics through the use of the OVP variation points.

Additionally, the materialization process has been improved to allow executing variation points with negative variability (line 5 in the `materialize` method). This means that those variation points with negative variability will be executed when the variability specification bound to the variation point is resolved negatively — i.e., it is not selected in a configuration of the resolution model. This depends on the semantics of the variation point and its implementation. For instance, the `ObjectExistence` variation point defines that a specific element in the base model will exist or not. In the case that the core base model includes all possible variations, the element that is associated with the `ObjectExistence` variation point needs to be deleted from the model if it is not positively resolved in the resolution model. So, the `ObjectExistence` variation point needs to be executed in order to delete the associated element in the base model. Figure 4 shows an example of M2M transformation in ATL with the semantics of the `ObjectExistence` variation point. The 'target' element in the base model will be deleted when the variation point is executed. The M2M transformation is parameterizable by using a third *binding model* and allows providing external parameters to the transformation rules independently from the input model (line 9 in Figure 4).

[9]The complete code can be found in http://150.214.108.91/code/cvl-umatool

```
1  -- @atlcompiler atl2010
2  -- @nsURI UML=http://www.eclipse.org/uml2/5.0.0/UML
3  -- @path Params=/ParamsMetamodel/src/CVLParams.ecore
4
5  module ObjectExistenceUML;
6  create OUT : UML refining IN : UML, PARAMS : Params;
7
8  -- Name
9  helper def: ElementName : String = Params!MOFReference.allInstances()->any(e | e.key = 'target').value;
10
11 rule deleteElement {
12   from
13       e : UML!NamedElement (e.name = thisModule.ElementName)
14   to
15       drop
16 }
```

Figure 4: ATL semantics: `ObjectExistence` variation point.

5 CONCLUSIONS AND FUTURE WORK

We have proposed an implementation of the CVL execution engine, including an API to resolve the variability through custom model transformations. The API is an stand-alone library that can be used in any application that needs to resolve the variability of CVL models. In comparison with existing CVL tools, our tool can be extended to meet the needs of architects and developers through the extension points identified in the CVL approach.

As future work, we are completing the API of the CVL engine in order to make it fully compatible with existing CVL tools and take advantage of the graphical editor for the CVL models.

ACKNOWLEDGMENTS

This work is supported by the projects Magic P12-TIC1814 and HADAS TIN2015-64841-R (co-financed by FEDER funds).

REFERENCES

[1] CVL Submission Team. 2012. Common Variability Language (CVL), OMG revised submission. http://www.omgwiki.org/variability/. (2012).
[2] Thomas Degueule, Joao Bosco Ferreira Filho, Olivier Barais, Mathieu Acher, Jérôme Le Noir, Sébastien Madelénat, Grégory Gailliard, Godefroy Burlot, and Olivier Constant. 2015. Tooling Support for Variability and Architectural Patterns in Systems Engineering. In *International Conference on Software Product Line (SPLC)*. 361–364.
[3] Jorge Echeverria, Jaime Font, Oscar Pastor, and Carlos Cetina. 2015. Usability Evaluation of Variability Modeling by means of Common Variability Language. *Complex Systems Informatics and Modeling Quarterly* 5 (2015), 61–81.
[4] João Bosco Ferreira Filho, Simon Allier, Olivier Barais, Mathieu Acher, and Benoit Baudry. 2015. Assessing Product Line Derivation Operators Applied to Java Source Code: An Empirical Study. In *International Conference on Software Product Line (SPLC)*. 36–45.
[5] João Bosco Ferreira Filho, Olivier Barais, Jérôme Le Noir, and Jean-Marc Jézéquel. 2012. Customizing the Common Variability Language Semantics for Your Domain Models. In *VARiability for You Workshop (VARY)*. 3–8.
[6] Ø Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *International Software Product Line Conference (SPLC)*. 139–148.
[7] Øystein Haugen and Ommund Øgård. 2014. BVR — Better Variability Results. In *System Analysis and Modeling: Models and Reusability*. Springer, 1–15.
[8] J. M. Horcas, M. Pinto, and L. Fuentes. 2014. An Aspect-Oriented Model transformation to weave security using CVL. In *International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 138–150.
[9] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2014. Injecting Quality Attributes into Software Architectures with the Common Variability Language. In *International Symposium on Component Based Software Engineering*. 35–44.
[10] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2016. An automatic process for weaving functional quality attributes using a software product line approach. *Journal of Systems and Software* 112 (2016), 78–95.
[11] Jose-Miguel Horcas, Mónica Pinto, Lidia Fuentes, and Steffen Zschaler. 2016. Towards Contractual Interfaces for Reusable Functional Quality Attribute Operationalisations. In *International Conference on Modularity*. 201–205.
[12] Andreas Svendsen, Xiaorui Zhang, Franck Fleurey, Øystein Haugen, Gøran K Olsen, and Birger Møller-Pedersen. 2010. Modeling Variability in SPLs. In *Software Product Line Conference*. 299.
[13] Anatoly Vasilevskiy, Øystein Haugen, Franck Chauvel, Martin Fagereng Johansen, and Daisuke Shimbara. 2015. The BVR Tool Bundle to Support Product Line Engineering. In *International Conference on Software Product Line (SPLC)*. 380–384.

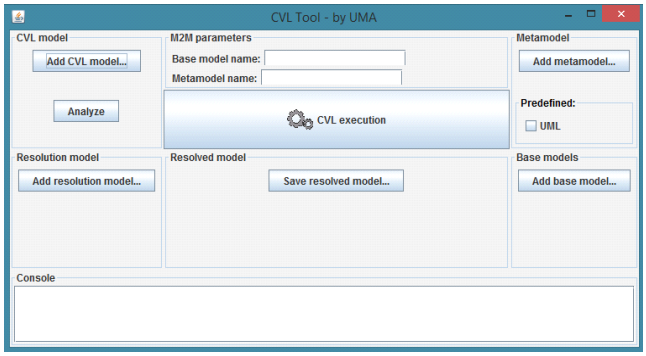

Figure 5: Our CVL execution engine tool.

A TOOL DEMONSTRATION

Figure 5 shows the GUI of the current version of our tool that implements the CVL engine using the API presented in Section 4.[10] It is released as an independent Java application, and allows integrating and executing the CVL approach both at design time in SPLs and at runtime for Dynamic SPLs. The CVL engine receives as input the variability model, the resolution model, the base models and their metamodels; and resolves the variability generating a resolved model. The variability and resolution models conform to the CVL metamodel.[11] The base models include (1) the core models over which the variability will be resolved, and (2) the models, if needed, that will be used as third-party libraries to be used during materialization (e.g., for the `FragmentSubstitution` variation point). By default, we provide the UML metamodel for base models conformed to UML, but the use of any other DSL is possible by providing its own metamodel. We provide an implementation of the M2M engine in ATL that allows performing any model transformation specified in ATL.

A.1 Demonstration roadmap

To illustrate the feasibility of our customizable CVL execution engine, we present the current available capabilities of our CVL engine prototype. The demonstration shows the materialization process of UML software architectures (i.e., the base models) based on the selection made by a software architect in a variability model — i.e., based on the resolution/configuration model. During the demonstration we present several examples that define different variability models, different variation points and different transformations. Examples of the items that we use in the demonstration are shown in Figure 6.

The example consists of a variability model that specifies the variability of two security concerns: `Encryption` and `Authentication`. Encryption allows choosing between the `RSA` and the `AES` algorithms. Authentication provides two different mechanisms: PIN-based and User + Password authentication (`PIN` and `UserPassword` features in the variability model). The base model is a library of security components that will be used in a specific application. To simplify the software architecture we only show the components related to the authentication concern. In order to customize the library to the application's needs, we link the variation points of the variability model with the software components of the architecture. Only one authentication mechanism is available in the application, so we use the Object Existence variation point, whose semantics is specified as a model transformation (implemented in ATL). In this case, we have implemented the Object Existence variation point as a negative variability variation point that means it will only be executed when the associated feature is not selected in the resolution model. The application developer selects the desired configuration for the authentication mechanism, providing a resolution model.

In our tool, when all the models have been successfully loaded and the CVL execution engine is executed, the execution is delegated to the ATL Transformation Engine that performs the transformations. In the example used to illustrate part of the demo, the feature `PIN` (child of `Authentication`) has not been selected and thus its associated variation point will be executed, removing the components from the base model. As a result, we obtained the `Resolved Model` with the `Authentication` component correctly configured with the `User + Password` mechanism.

[10] The tool is available in http://150.214.108.91/code/cvl-umatool
[11] CVL metamodel is available in http://150.214.108.91/code/cvl-metamodel

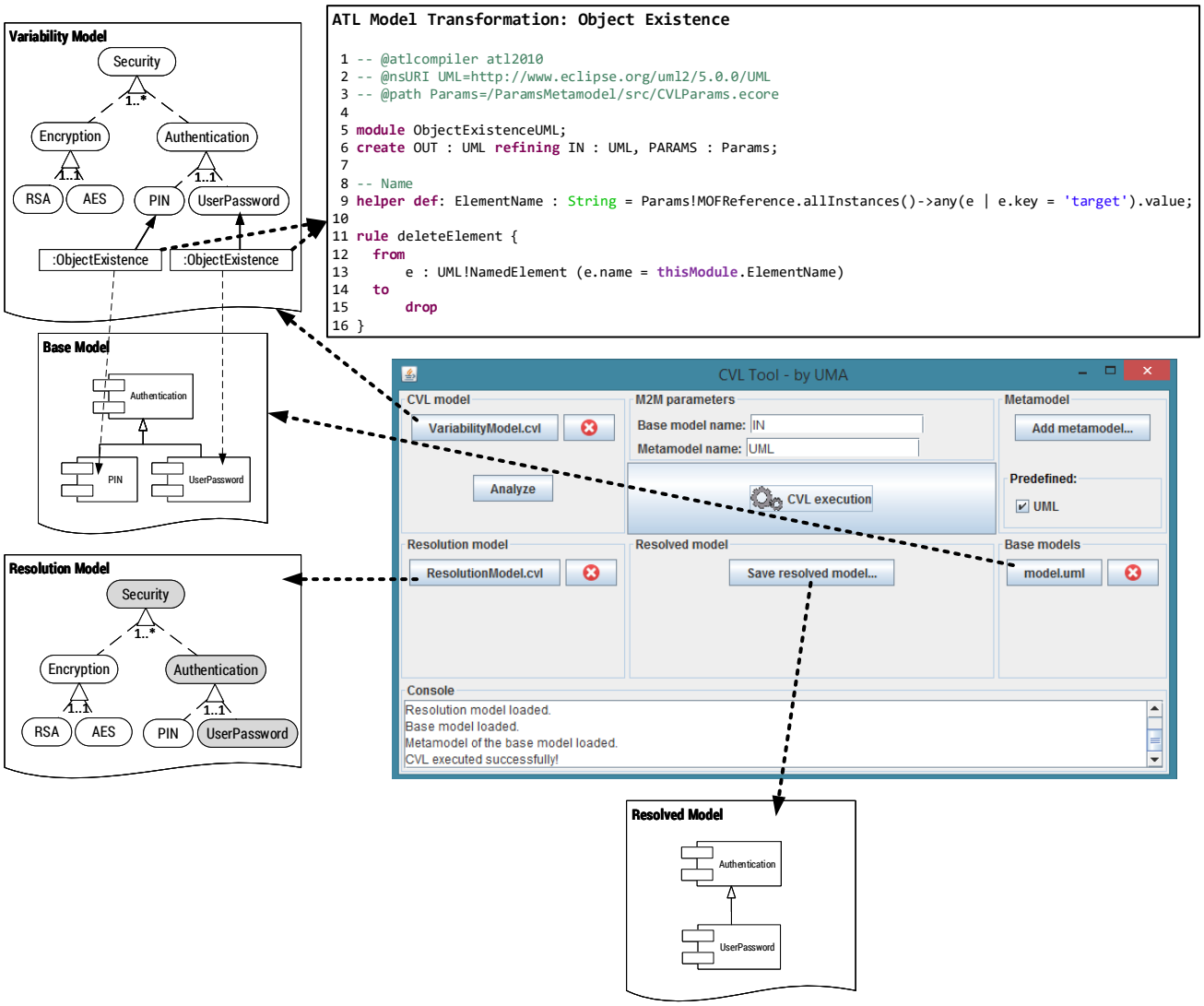

Figure 6: Demonstration scenario.