A Model Engineering Approach to Tool Interoperability

Yu Sun[1], Zekai Demirezen[1], Frédéric Jouault[2], Robert Tairas[1], Jeff Gray[1]

[1] Department of Computer and Information Sciences, University of Alabama at Birmingham
{yusun, zekzek, tairasr, gray}@cis.uab.edu

[2] AtlanMod (INRIA & EMN), Nantes, France
frederic.jouault@inria.fr

Abstract. The integration of various tools is a common requirement throughout the software development process. It is often desirable to consult several tools that perform similar functionalities in the same domain to obtain different perspectives and results to assist design and maintenance decisions. In many cases, tool interoperability requires the generalization of tool-specific data, which necessitates homogenizing the data such that intellectual assets can be shared through a common framework (e.g., the integration of results from various clone detection tools). This tool demonstration summary presents a software language engineering solution technique that uses Model-Driven Engineering to address tool interoperability. A specific focus of the paper is a demonstration of model transformation applied to the task of homogenizing different data formats among similar tools. The challenges of tool integration are discussed in the paper, along with a detailed case study that highlights the benefits of applying a model transformation solution to tool interoperability.

Keywords: Model Engineering, Tool Interoperability, Model Transformation, Domain-Specific Languages, AMMA

1 Introduction

The ability to model the key characteristics of new tools, and to integrate them with a set of previously defined tools, can be very useful. Often, however, researchers independently develop similar tools to perform the same functionality within a particular domain. Each isolated effort defines a different semantic model and uses uncommon storage representations. Unfortunately, this poses a problem when it comes to the important issue of integration – the result is an inability to provide a seamless exchange between tools. For example, our own past work observed numerous tools within the domain of avionics fault analysis that had very much in common, but those tools could not share models. A solution was an integrated model and associated tool adapters that allowed data exchange among the tools [7]. The area of Enterprise Application Integration (EAI) has provided several additional technologies to assist in the tool integration problem across many domains, such as healthcare [19]. This tool demonstration paper summarizes our investigation into using concepts from software language engineering, in particular model transformation, to address the tool integration problem.

In tool interoperability, different standards and formats make software interoperability a challenge [2]. There are several existing approaches that can be adopted to overcome the limitations of tool interoperability. For example, a general approach that is based on traditional parsing and interpreting activities can be implemented with general-purpose programming languages. As a second example, XML-based interoperability has emerged as a popular choice for a generic exchange format between software tools. However, although this works relatively well when all considered tools use some form of XML, this kind of solution is not as convenient in other contexts (e.g., when context-free parsing of the storage format is necessary).

A third approach to interoperability is based on model transformation and is the focus of this paper: the different formats are captured as abstract definitions of data structures (i.e., metamodels), and transformation rules map from one representation to another. The AtlanMod Model Management Architecture (AMMA) [8] is a model engineering framework that may be used to build bridges between tools or Domain-Specific Languages (DSLs). Each tool or DSL is captured and represented as a coordinated set of models. The work described in this tool demonstration summary uses three of the main capabilities of AMMA: metamodeling with the Kernel MetaMetaModel (KM3) [4], model transformation with the AtlanMod Transformation Language (ATL) [6], and projections to (i.e., extraction) and from (i.e., injection) other technologies (e.g., grammars, XML). Projections are especially useful in the context of model-driven tool interoperability because each tool typically uses a specific file format. Notably, AMMA provides the Textual Concrete Syntax (TCS) [5] tool to deal with context-free syntax.

Although our specific tool demonstration is focused on using AMMA as a solution strategy, we believe that the general concept can be used with most modeling and language engineering tools. The next section presents a case study that demonstrates how software language engineering (specifically model engineering and model transformation) can be used to assist in the sharing of results and data across tools from the same domain. A concluding section summarizes the paper by presenting lessons learned and pointing toward future work.

2 Visual Representation for Clone Detection

Code clones are blocks of statements that are duplicated in multiple locations of one or more programs. Programs containing code clones are prone to manifest problems in the maintenance phase of the software development process [13]. A number of tools have been developed to automatically detect code clones in programs, such as CloneDR [1], CCFinder [12], Simian [10], and SimScan [11]. These tools take the source program as input and generate mostly textual reports about the code clones that were detected. Figure 1 shows excerpts of the reports generated by three popular code clone detection tools. The reports record the location and length of each code clone. Depending on the size of the application that is analyzed, the corresponding clone report files could contain tens of thousands of lines of text.

Simian	Found 6 duplicate lines in the following files: Between lines 117 and 122 in E:\source\FieldRecordTextTest.java Found 6 duplicate lines in the following files: Between lines 611 and 617 in E:\source\Utility.java Between lines 602 and 608 in E:\source\Utility.java
SimScan	34751-E:\source\Utility.java:107-110, 34912-E:\source\Utility.java:119-122, 11421-E:\source\ExtendedVector.java:35-38, 34806-E:\source\Utility.java:111-114, 34967-E:\source\Utility.java:123-126, 48713-E:\source\WeightedStatistics.java:60-63
CloneDR	#1 5a7e550 +1234 rule_name 0.970 2 2 11 #2 5a7e550 3bcb780 19 0 29 0 E:/source/Utility.java #2 5a7e550 3bd6560 45 0 55 0 E:/source/KeyCounterTest.java

Fig. 1. Excerpts of three code clone detection reports.

2.1 Challenges of Comprehending Multiple Tool Results

Because many tools use different criteria and methods to detect clones, it is desirable for users to observe the reports of several tools and compare them so that a more comprehensive understanding of the code clones can be obtained. However, these tools apply varying formats and representations in their reports, making it very challenging to understand, compare, and integrate the results from these diverse sources. Therefore, a uniform format for code clone detection reports to which different outputs could be automatically transformed would greatly simplify the integration process of various detection results. Moreover, a graphical representation of clones could further aid clone comprehension.

2.2 Overview of Solution Approach using Model Engineering

This section introduces our work that transforms text-based code clone detection reports to SVG (Scalable Vector Graphics) [9], a language that describes two-dimensional graphics in XML. More specifically, certain shapes and graphical components will be drawn to represent the code clone information according to the textual results. The reason for choosing SVG as the final visual representation is that the language is declarative, which provides for a simpler metamodel. In addition, because SVG is based on XML, the built-in XML facilities of AMMA can be used.

Using our model transformation approach to tool interoperability, the reports in Figure 1 will be transformed to SVG code and displayed in a web browser. Performing source-to-source transformations can realize this, but most of the process would need to be hard-coded and therefore too complex and error-prone to adapt and extend when new clone detection tools emerge. As an alternative to source transformation, we use model engineering with a source-to-model-to-source approach, where the first source refers to context-free text, and the second corresponds to XML.

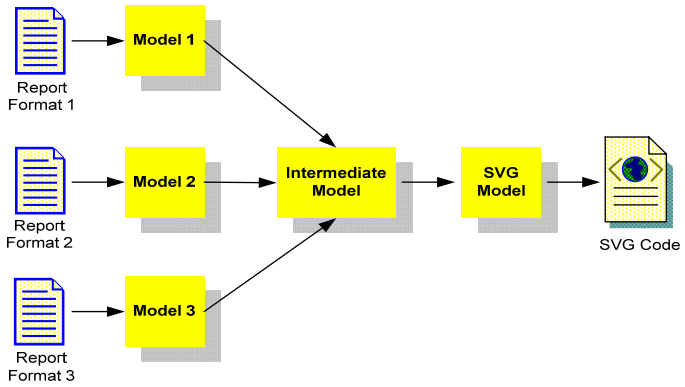

Fig. 2. Overview of code clone detection tools interoperability.

Because models have a more structural syntax to capture the main concepts and relationships (i.e., models are primarily focused on abstract syntax), implementation of tool data mappings can be simplified through model transformations. In our approach, we first project the textual report into its corresponding model through injection with TCS. This source model is then transformed into the target SVG model. Finally, we project SVG code from the SVG model through extraction. Figure 2 provides an overview of our approach. Our goal is to generalize the experience gained in this case study of one domain to address tool interoperability problems in other domains. Usage of an intermediate model makes it possible to decouple the transformations dealing with the various source formats from the transformation that produces the SVG model.

2.3 Implementation of Clone Detection Tool Interoperability

The basic idea of the model-driven solution is to define the abstract syntax (metamodel) and concrete syntax (grammar) for each of the different code clone detection reports and SVG, then execute associated transformations between the code clone detection reports domain and the SVG domain. These two steps can be performed with the facilities provided by AMMA. Figure 3 shows a lower level view of Figure 2 and offers the detailed structure of the implementation. Each step will be explained in the following sub-sections.

Definition of Sub-domains and Injection of Source Reports. Because the transformation process operates on models, the very first step involves injecting the textual code clone detection report generated by a certain tool into its corresponding model. To realize this, the metamodel must be defined for the tool's report using KM3 (Step 1 in Figure 3). Figure 4 shows the metamodel designed for the detection report

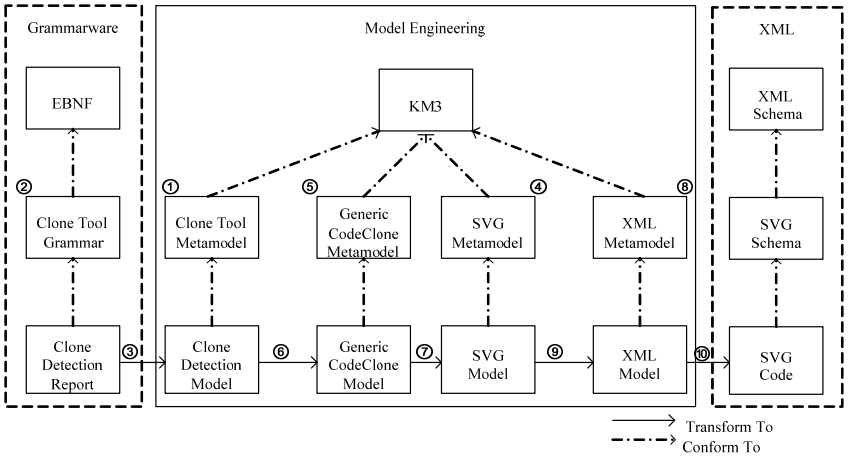

Fig. 3. Overview of the code clone tool interoperability implementation.

generated by Simian[1]. The report includes summary information about the analyzed files (e.g., the total number of lines checked, total number of blocks, and clone pairs). This metamodel represents the abstract syntax, because it defines the components and their relationships without the information about the concrete grammar.

Parsing is needed to translate the textual report and capture the necessary information. By defining the context-free grammar for a tool report using TCS (Step 2 in Figure 3), the textual syntax is mapped to its metamodel, so that it can parse the input file and inject it into the model, which conforms to the metamodel. However, not all reports are based on a context-free grammar. This necessitates some parts of the reports to be preprocessed for correct parsing.

For each kind of code clone detection tool or sub-domain, both a metamodel specification in KM3 and a concrete syntax definition in TCS are needed. These specifications work together to enable injection of the initial detection reports into models for the later transformation process (Step 3 in Figure 3).

Definition of Target Domain and Transformations. When the input model is ready, it can be transformed to the target model. In this experiment, SVG is the target domain, so the SVG metamodel must be defined first in KM3 (Step 4 in Figure 3). The TCS specification for the SVG grammar is not needed, because SVG is based on XML and AMMA provides an embedded extraction engine to generate XML. More details about this will be given in the next step. SVG is an XML-based language that enables the specification of a graph containing rich elements, so a complete metamodel for SVG can be very large. Because only a small number of elements (e.g., group, rectangle, and text) will be used in this interoperability phase, a subset of the SVG metamodel is used to satisfy these requirements, as shown in Figure 5.

[1] Please note that due to space considerations, the metamodels presented in this paper are UML class diagrams. The actual KM3 metamodels and all other artifacts of the project are available at [15].

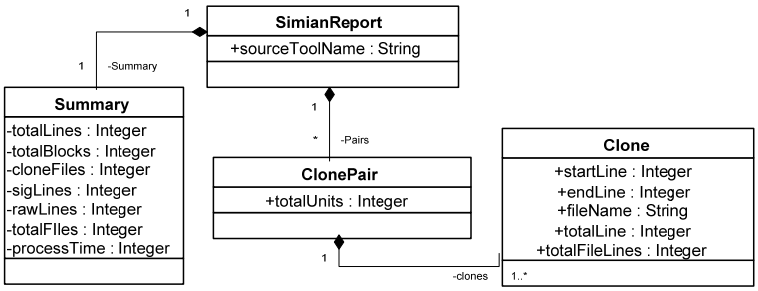

Fig. 4. Metamodel for Simian's code clone detection report.

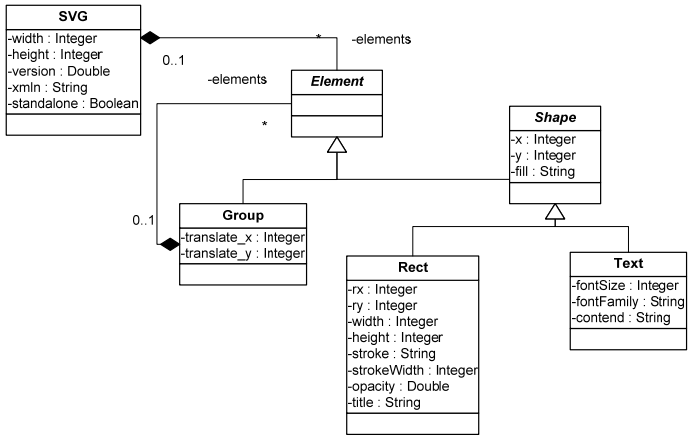

Fig. 5. Metamodel for a subset of SVG.

Our goal is to transform multiple tool results from the code clone domain to the SVG domain. However, defining the transformation process for every tool to SVG is not efficient and extensible, because these are two totally different domains with little direct connections and mappings; hence, the transformation rules would be comparatively complicated. If it is required to transform each code clone detection report to SVG directly, multiple complex transformations between these two domains would be required. In order to optimize the whole process, we defined an intermediate model called a *Generic Code Clone* (GCC) model (conforming metamodel in Figure 6), which is closely related to the code clone domain that contains the common concepts and features of code clone detection tool reports (Step 5 in Figure 3). This is actually the minimum set of the features shared by all the tools. Instead of transforming each Clone Detection model to the final SVG model, the Clone Detection model is transformed to the GCC model (Step 6 in Figure 3), which is then transformed to the SVG model (Step 7 in Figure 3). In this way, only one transformation from the code clone domain to the SVG graphical domain is needed. A similar idea also appears in compiler design, where an Intermediate Representation (IR) is used to optimize code generation [16].

The principles under the transformation from the GCC model to the SVG model are the following:

- A rectangular box is drawn for each clone group.
- Smaller rectangles are drawn inside this box to represent every file that contains one of the clone units in the clone group. The length of these rectangles is based on the length of the file they each represent.
- Colored lines with different width are used to represent the location and length of the code clones in the files.

We use ATL to write the transformation rules, specifying the mappings of the elements between the two domains.

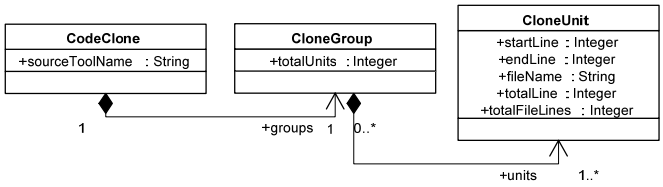

Fig. 6. Generic Code Clone (GCC) metamodel.

Generation of Target Code. An extraction process to generate the SVG code from an SVG model is needed. The AMMA platform supports automatic generation of XML from an XML model. Because SVG is based on XML, in the final step, the transformation is defined in ATL from the SVG model to the XML model (Step 9). However, the XML metamodel must be defined to enable the transformation (Step 8). When the final XML model is produced, the SVG code can be generated by a single extraction (Step 10).

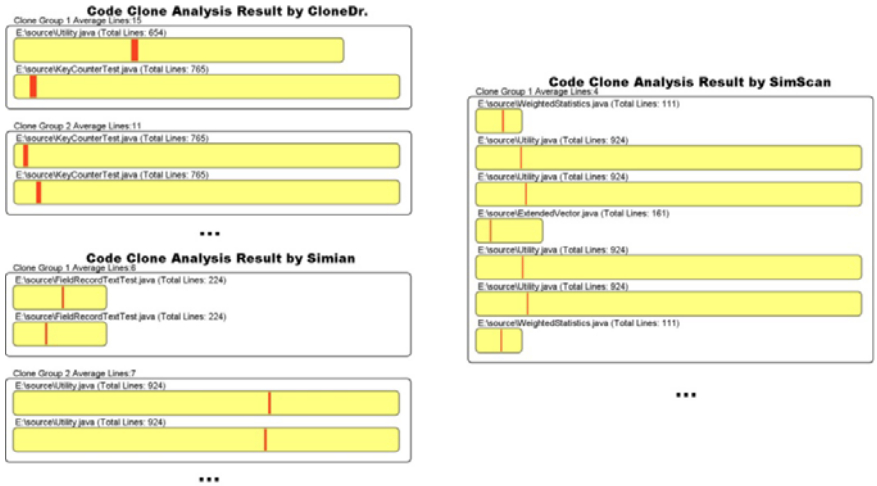

Fig. 7. Uniform visual representation for reports of the three tools (subset of reports).

Interoperability Results Reported by SVG. Figure 7 shows the result of one conversion experiment to SVG. The three small graphs correspond to the three textual reports in Figure 1. The files are represented by rectangles with different lengths. The location and length of each code clone block is highlighted in the rectangle. Also, other information such as the file name and average number of clone lines is shown. Support for other tools (e.g., CCFinder [12]) can be extended by: 1) defining the new tool's metamodel and textual grammar, and 2) writing the transformation rules from the new tool model to the GCC model. The complete experiment can be found in [15].

3 Conclusion and Future Work

The challenge of tool interoperability often occurs in the development of applications where a cadre of analysis and design tools is available. Each specialized tool contributes to a crucial step in the development process. It would be beneficial to capture the information in the context of one tool and use it in a different tool. We have used software language engineering to implement one form of interoperability which concerns the data exchange and sharing of information between tools. There are several lessons that were learned from applying model transformation to the tool interoperability problem:

- **Model transformation provides separation of concerns across the integration process**: Separation of concerns in our experiments is shown in four aspects: 1) The whole process is composed of three parts – defining source domains, defining target domain, and writing transformation rules; 2) When defining a domain, the metamodel (abstract syntax) and parser (concrete syntax) are specified separately. The AMMA platform connects the abstract syntax and concrete syntax when doing injection; 3) Metamodels have a clear and organized structure about the components and relationships they contain, which enables users to focus more on the concept and semantic mappings when writing transformation rules.
- **Adaptability and extensibility in defining new tools**: It is often necessary to define a new tool or DSL when implementing tool interoperability. We observe that a certain level of modularity may be achieved by introducing intermediate (or *pivot*) metamodels in the transformation chain between the source and target domains. For instance, a generic or pivot tool is often needed to optimize the exchange among different tools. In the context of model transformation, only a metamodel is needed for a new tool. Although some tools with complex functions may make the metamodel difficult to specify, we can define a subset of the tool to satisfy the specific need of a tool interoperability case. In some other cases, a parser (concrete syntax) is also needed to support a new tool. In this situation, the extensibility depends on the complexity of the grammar, because building a parser is not an easy task [17]. However, because most of the tools are domain-specific, their grammar and parser are generally much simpler than general-purpose programming languages. In addition, some preprocessing can simplify the complex detection report text in order to make the parser easier to build.

Admittedly, if the tool or data has a very complex syntax that makes building the parser extremely difficult, additional effort is needed.

- **Models should be the primary representation for tools**: Models are not the initial and final representations for most tools, so an extra initial effort is needed to inject and extract the tool data into a model. This involves multiple steps, and might make the whole approach less direct and efficient than source-to-source transformation. Fortunately, some powerful facilities are provided to simplify the exchange between sources and models. For example, in AMMA, XML source code can be automatically generated from the XML model by an embedded XML engine. In addition, the Ant build tool [18] is also supported in AMMA, which enables the generation of build files to execute all the steps automatically.
- **Maturity of modeling tools**: Although tools like AMMA provide many benefits of software language engineering, there are still limitations in the support and usage of such tools. As an example, the level of debugging support in AMMA is not as mature as in traditional programming languages. Furthermore, we believe that the general principles of using software language engineering to solve the tool interoperability problem is not unique to AMMA; other language environments should also be able to provide similar advantages.

As implied by the last lesson learned, there are several topics that represent future work in this area. One limitation of this approach is that the number of tools that can be integrated depends on the ability to parse the tool representation. The textual format of some tools may not be based on a context-free grammar, which complicates the mapping process. The only option is to eliminate the parts that do not conform to the context-free grammar. However, this will make the whole process less automated, and will also make it possible that some important information is lost.

The complete details about the project (e.g., full source for the various KM3, TCS, and ATL specifications) are available at the project website [15].

Acknowledgements. This work was supported in part by an NSF CAREER award (CCF-0643725), an NSF CPA award (0702764), the OpenEmbeDD project, and the EDONA project.

References

[1] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection using Abstract Syntax Trees. In: International Conference on Software Maintenance, pp. 368-377, Bethesda, Maryland (1998)

[2] Benguria, G., Larrucea, X.: Data Model Transformation for Supporting Interoperability. In: International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems, pp. 172-181, Alberta, Canada (2007)

[3] Budinsky, F., Steinberg, D., Ellersick, R., Merks, E., Steinberg, D., Grose, T.: *Eclipse Modeling Framework*. Addison-Wesley, Reading (2003)

[4] Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 171-185, Bologna, Italy (2006)

[5] Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: International Conference on Generative Programming and Component Engineering, pp. 249-254, Portland, Oregon (2006)
[6] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Model Transformations in Practice Workshop, International Conference on Model-Driven Engineering, Languages, and Systems, Montego Bay, Jamaica (2005)
[7] Karsai, G., Gray, J.: Component Generation Technology for Semantic Tool Integration. In: IEEE Aerospace Conference, pp. 491-499, Big Sky, Montana (2000)
[8] Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL Frameworks. In: International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 602-616, Portland, Oregon (2006)
[9] Scalable Vector Graphics (SVG). http://www.w3.org/Graphics/SVG/.
[10] Simian. http://www.redhillconsulting.com.au/products/simian/.
[11] SimScan. http://www.blue-edge.bg/simscan/simscan_help_r1.htm.
[12] Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multi-Linguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering 28, 654-670 (2002)
[13] Tairas, R., Gray, J.: Phoenix-Based Clone Detection Using Suffix Trees. In: ACM Southeast Conference, pp. 679-684, Melbourne, Florida (2006)
[14] Tratt, L.: Model Transformations and Tool Integration. Software and Systems Modeling 4, 112-122 (2005)
[15] Representation for Clone Tools (Eclipse ATL Use Case). http://www.eclipse.org/m2m/atl/usecases/VisualRepCodeClone.
[16] Aho, A., Lam, M., Sethi, R., Ullman, J.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston (2007)
[17] Lämmel, R., Verhoef, C.: Cracking the 500-Language Problem. IEEE Software 18, 78-88 (2001)
[18] Apache Ant. http://ant.apache.org
[19] Khoumbati, K., Themistocleous, M., Irani, Z.: Investigating Enterprise Application Integration Benefits and Barriers in Healthcare Organisations: An Exploratory Case Study. International Journal of Electronic Healthcare 2, 66-78 (2006)