Model-Driven Interoperability of Dependencies Visualizations

Vincent Mahé[1], Hugo Brunelière[2], Frédéric Jouault[2], Jean Bézivin[2], and Jean-Pierre Talpin[1]

[1] Espresso research team
INRIA Bretagne - Atlantique
Campus de Beaulieu
35000 RENNES (France)
{Vincent.Mahe|Jean-Pierre.Talpin}@inria.fr
[2] AtlanMod research team
INRIA Bretagne - Atlantique
École des Mines de Nantes
44000 NANTES (France)
{Hugo.Bruneliere|Frederic.Jouault|Jean.Bezivin}@inria.fr

Abstract. Software tools and corresponding knowledge tend to be collected and packaged into platforms like Eclipse, MathLab or KDE. Their success and usefulness combined with their growing size and complexity rise issues about management of dependencies between their components and between the platform and other applications which rely on its plug-in system and/or provided functionalities. Such problems imply need for dependencies management tools in which visualization is a core feature. As dependencies are also a concern in domains like Object-Oriented Programming or Operating System packaging, we may expect to reuse corresponding works in visualization. But each domain and its related dependencies problem have induced their own hard-coded viewing and browsing tools. In this article we present how we have reuse existing visualization tools for our platform cartography together with our own displays using a Model-Driven Interoperability approach to easily realize bindings btween visualization tools.

1 Introduction

[CfP keywords]: Visualisation in software engineering (e.g. UML diagrams), Integration of software visualization tools and IDEs, Visualization of software evolution

Platforms are a widely used concept we may encounter in domains as distinct as hardware industry, engineering environments or web services. They satisfy same needs: defining an interface to extra pieces, embedding selected services, etc. But their success together with their growing size and complexity rise management issues we try to deal with through a visualization approach. Despite their complexity, those platforms share main features and corresponding issues (e.g. dependencies management) we would like to reach once for all. We should rely on the numerous researchs done on these visualization issues to solve our platform problems. But each existing tool is built onto a specific domain in an independent way using its own data format or even a monolithic

application doing reverse engineering, computation and visualization in the same execution thread. We propose models paradigm and tools as a gateway between platforms specific needs and visualization tools. As an additional result, our generic dependency management gives easy binding to dependency tools for any domain which could need them out of platforms world. This paper presents our initial motivation in the section 2. The section 3 presents our visualization tool and its main properties. Our attempts to plug our platform model into existing tools is then explained with the experiments we have done in the section 4. The section 5 discusses about interests and needs of generic visualization tools. We conclude in section 6.

2 Motivation

In year 2000 the *Object Management Group* standardization organisation presented its Model Driven Architecture (MDA) initiative[11], which promotes a separation of software engineering process artifacts between Platform Independent Models (PIM) and Platform Specific Models (PSM) enriched from the previous ones by (automated) inclusion of platform dedicated stuff. They understood platforms as middleware but did not explore the underlying world; platforms were an issue they tried to escape from by modeling applications at a higher (platform-free) level.

Since this initiative, the multiplication of platforms has spread and blurred the concept: hardware platforms (as processors: x86, ARM; as all-in-one computers: smartphones, internet appliances), software platforms (as OSes: Windows, MacOSX; as frameworks: Java, Eclipse, OSGi), as sets of tools (Eclipse Modeling, Topcased, MathLab). The large use of platform concept emphasizes its usefulness and gathers needs like handling of components. As actual platforms are larger and larger, platform builders and users need a way to manage them in-the-large. The cartography of platforms and the management it enables are needed for many different platforms (Eclipse, Linux, web services) which do not fit well together. So our works rose some issues we managed on the fly. One of our main needs was a visualization tool for dependencies management.

Once our need was identified we expected to find existing tools for visualization of dependencies because this question has been investigated in many works. As a matter of fact we found plural tools about dependencies with an embedded viewer:

- DepAn [2] is a tool for *Dep*endency *An*alysis of Java programs. It inspects a .jar file and offers many views on the generated data file.
- CAP [1] is a Code Analysis Plug-in for Eclipse. It analysis algorithm is based on Martin works [10].
- BARRIO [6] is a cluster analyser for object-oriented programs.
- GUESS [3] is a generic graphs visualizer which accepts GraphML XML format files as input data.
- SHriMP [8] stands for Simple Hierarchical Multi-Perspective. It is both a technique and a software application for exploring software.
- SIV [7] stands for Scenario Inter-dependency Visualization.
- SolidFX [12] is an integrated reverse-engineering environment for C and C++ code (not tested).

– STAN is an Eclipse-based STructure ANalysis tool for Java.
– X-Ray [9] is a software visualization plug-in for Eclipse.

But those tools suffer some lacks and rigidities that disqualify them for our purpose relying on platforms instead of OO source code: some visualization tools target a specific domain like Java packages or Object-oriented typing and inheritance. Another big issue is about their hard-coded relation to their subject (Java source code) which forbiddes a reuse in other domains. As our needs concern deployement features or underlying plug-ins relationships, those tools can not be used and we must reinvent the wheel once again.

These obstacles emphasize the need of a generic approach of visualization in order to capitalize the knowledge about dependencies viewing and offer its features and abilities to all domains which are facing dependencies issues. Such an independant tool will benefit of every effort and advance in the visualization domain with ability to apply them on all *customer* application domains.

3 Model-Driven Approach

As seen previously, dependencies management is a recurrent problem in software engineering and requires visualization to help. Many tools have been develop to visualize corresponding data relationships. Despite those tools are dedicated to a specific domain, we would like to reuse their main functionalities to display our platform-related dependencies. To do such a reuse, we need to interoperate our platform data and those tools in an easy way. The Model-Driven Engineering (MDE) gives us the way to do that.

MDE is a software engineering approach considering models as first-class entities, with a strong underlying conformity of those models to metamodels (models defining the syntax of dedicated languages). Those metamodels can be multi-purpose like the Unified Modeling Language (UML), or dedicated to a specific application field (they are called Domain Specific Modeling Languages, or DSMLs). Given this conformity, models can be processed with transformations or computed in order to verify some properties on them. MDE uses models as first class entities with metamodels as sets of type definitions.

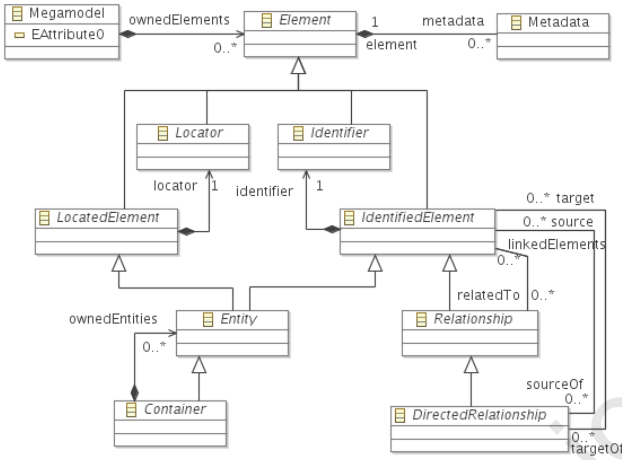
First we present our model-driven representation of platforms. Then we detail the generic dependencies visualization tool we build on it.

3.1 Megamodeling and Cartography

Megamodeling is a concept of Model-Driven Engineering (MDE) which goal is to provide over-modeling of MDE artefacts like models, metamodels, transformations that software engineers can produce when working on a given application domain or the whole company information system.

The AtlanMod MegaModeling Management (**AM3**) is a metamodel and dedicated tools to model and manage complex sets of (not only MDE) artifacts in a generic way. We use its metamodel (see figure 1) as a fondation for our cartography of Eclipse-like platforms.

Fig. 1. AM3 Core metamodel (subset)

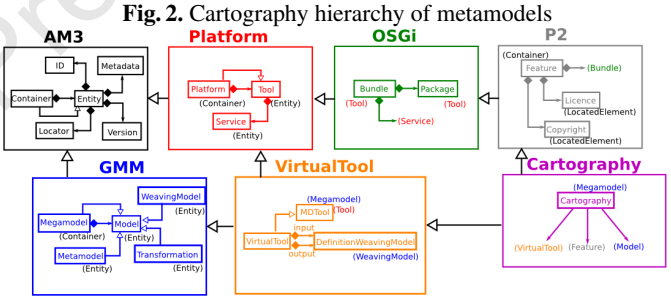


AM3 is a generic model-driven manager but our need to specifically manage platforms implies deeper concepts like versionning or tools. The corresponding answer is a more specific metamodel which inherits the **AM3** metamodel and specializes some of its concepts. It gives the **Platform** metamodel which types inherit from **AM3** types.

Eclipse platform offers its versatility services through a plug-in mecanism which relies on OSGi standard (on its Eclipse implementation which is named Equinox). As OSGi concepts are more precise and restrictive than equivalent concepts in **AM3** or **Platform** metamodels, we design a new wrapping **OSGi** metamodel which inherits from them.

The platforms under examination also include packaging (like the Eclipse **P2** features) and modeling environment (defined in the Global Modeling Managment **GMM** metamodel) so our complete **Cartography** metamodel is a hierarchy of metamodels which redefine the most generic ones (see figure 2).

Fig. 2. Cartography hierarchy of metamodels



The **OSGi** and **P2** metamodel extensions are specialized sub-types for easier representation of Eclipse platforms. Other kind of platforms which are not Eclipse-based could be mapped to the **Platform** metamodel (or even the **VirtualTool** metamodel if

they include modeling tools), taking full benefit of underlying concepts and corresponding types without Eclipse specificities.

3.2 Dependencies visualization

One of our main concerns facing software application platforms like Eclipse is the issues about dependencies between their components. One answer to this problem is a visualization displaying those dependencies in a graphical way.

Our hierarchy of metamodels from the most generic to the most specialized offers the capacity for specialized types inheriting from a generic type to be compatible with model-based tools built upon this generic type. We illustrate the benefits of this principle on the dependencies.

Relationship & DirectedRelationship concepts The **AM3** metamodel includes a notion which is close to dependency: the `DirectedRelationship` type (defined upon the more generic `Relationship` type). It relies one or multiple source `Entity`(ies) to one or multiple target `Entity`(ies) (see figure 1).

DirectedRelationship extraction An *AM3toGraphML* model-to-model transformation has been written in ATL language to take any **AM3** (compatible) model and extract all its `DirectedRelationship` and related `Entity` elements. It generates the corresponding graph edges and nodes in a model which conforms to the **GraphML** [4] metamodel. When a source or a target of the relationship lacks (no source or no target), it creates the corresponding node with an error tag and the lack information is added to the node label:

```
rule Relationship2Edge {
  from
    dr  : AM3!DirectedRelationship
  to
    edges : distinct GraphML!Edge foreach(pair in
     thisModule.STPairs(dr)) (
      id      <- dr.__id,
      source  <- if (pair.source = OclUndefined)
        then thisModule.createLackingNode(
          'source lacking for ' + dr.__id)
        else
          thisModule.NodefromEntity(pair.source)
        endif,
      target  <- if (pair.target = OclUndefined)
        then thisModule.createLackingNode(
          'target lacking for ' + dr.__id)
        else
          thisModule.NodefromEntity(pair.target)
        endif,
      graph   <- thisModule.graph
    )
}
unique lazy rule NodefromEntity {
  from
    s : AM3!Entity
      (s.sourceOf->size() > 0
       or s.targetOf->size() > 0)
  to
    node  : GraphML!Node (
```

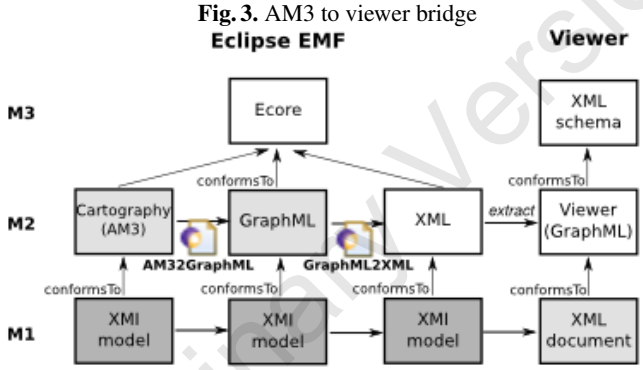
```
30        id <- s.identifier.value,
31        datas <- Sequence{
32          thisModule.createData('label',
33            s.identifier.value),
34          thisModule.createData('error', '')},
35        graph <- thisModule.graph
36      )
37  }
```

The resulting output is a GraphML model (conforming to the **GraphML** meta-model) which is transformed in the corresponding textual XML file (with the GraphML syntax) using the ATL XML extractor.

The ATL environment includes an extractor which generates an XML textual file corresponding to the XML model. *GraphML2XML* model- to-model transformation and XML extractor are generic tools made in a previous project.

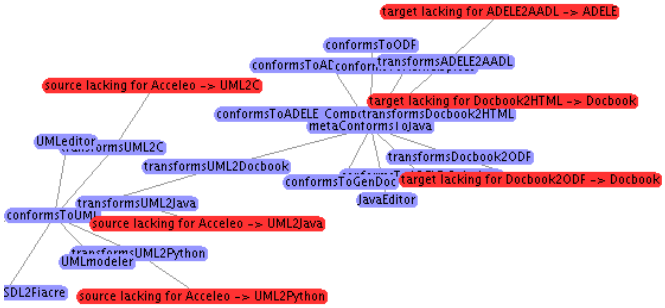
The scheme of the whole model-driven chain is presented in figure 3.

Fig. 3. AM3 to viewer bridge



DirectedRelationship visualization For visualization we have realized a simple tool using Prefuse to display the corresponding dependencies. Lacking nodes are in red. See figure 4 for a sample.

Fig. 4. A sample of our dependencies visualizer



In the next section we demonstrate the easyness of this model-driven approach to interoperate tools by displaying DepAn data in the SIV tool through our cartography tools and concepts.

4 Model-Driven Interoperability of visualizations

As we have seen in previous section, any kind of dependency which could be modeled as an **AM3** `DirectedRelationship` type can then be passed to our visualization tool. We will use this property to bridge our Cartography and some existing dependencies visualization tools. As such an approach uses models and their advanced properties in order to realize the corresponding gateways, it is called Model-Driven Interoperability (MDI).

4.1 What is Model-Driven Interoperability?

MDI is the third trend of concepts and solutions emerging from Model-Driven Engineering (MDE) as presented in section 3:

1. The first applications of MDE focused on code generation. The goal was to obtain final software artefact from design models.
2. A second generation of MDE techniques was concern by reverse engineering of existing systems. The purpose was to build models from software artefacts in order to use model-driven approaches on them.
3. A third trend is upcoming in MDE which uses model-driven approaches to build gateways between existing software applications. As exposed by Bézivin et alii [5], it relies on model transformations to transfer informations from a technical world into a distinct other world. MDE ability of tooling a given domain is used to write tools which:
   - extract models of the emitter data
   - transform them in receiver equivalent models
   - translate received models in the receiver kind of data

A strong impact of Model-Driven Interoperability is the fact that it builds tools which automate data transfers between the two technical distinct domain applications. The bridges can be complex but their use is clear and could be easily integrated into the graphical user interface of the interlinked applications.

We will use this approach in order to realize bridges between existing dependencies visualization tools with the **AM3** metamodel as a central root. A first gateway will enable export of our dependencies to the SIV tool. A second set of tools will be made in order to import DepAn visualizer data into our AM3 dependencies tool. The interest of using AM3 as a central root will then be demonstrated by easily realizing a complete chain between DepAn and SIV tools using previous bridges together.

4.2 Display our data into an existing tool: Cartography to SIV

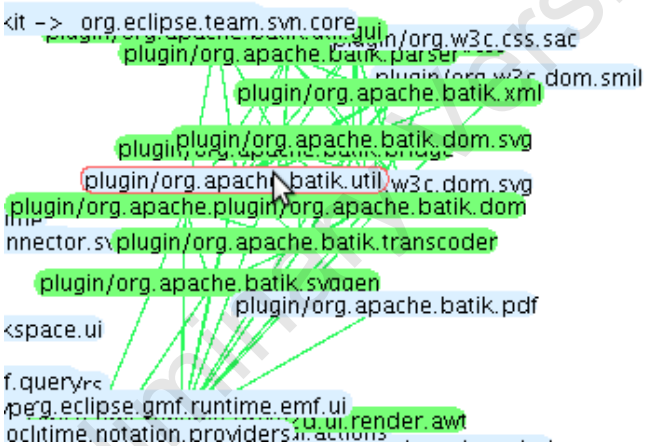
The goal is to export our Cartography dependencies to the SIV tool in order to take advantage of its existing visualization features [7].

As SIV tool use GraphML XML syntax as input, the chain of transformations between Cartography data and SIV tool is designed like the one which export **AM3** relationships to our generic visualization tool (see section 3.2). The only change is the first step which is now realized by the *AM3toSIV* transformation: it takes an AM3 (compatible) model as input and generates a SIV model which is a dedicated GraphML model with SIV specific tags (k0, k1, k2, k3).

The scheme of this Model-driven chain is the same as the one presented in figure 3

The resulting GraphML (with SIV data) is transformed in a XML model and then extracted in the corresponding XML textual file which is opened and displayed by SIV application. Figure 5 is a sample of our plug-ins dependencies displayed in SIV viewer.

Fig. 5. Cartography plug-ins dependencies in SIV



Using MDI we have connected our Cartography data to the SIV external tool.

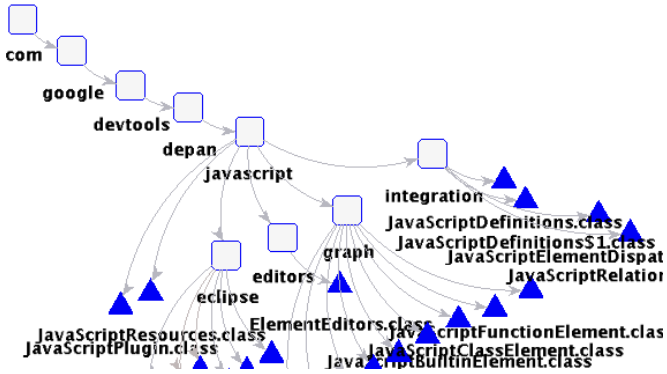
4.3 Import data from an existing tool: DepAn to AM3

The goal is to build a bridge from an existing dependencies tool to our generic AM3 metamodel and tools. The choosen DepAn tool could then be taken as input of our dependencies visualization tool.

The DepAn is a direct manipulation tool for visualization, analysis and refactoring of dependencies in the Eclipse environment (see figure 6 for a sample of its viewer). It includes a dependency discovery tool for Java applications which parse the `.java` files of a given directory (and all Java packages sub-directories) and generates the corresponding dependencies informations. The tool has its own XML file format with the `.dpang` extension.

We have to get data from DepAn files in order to visualize them. To reach this goal we built the Model-driven process below (similar to previous export chains but inverted):

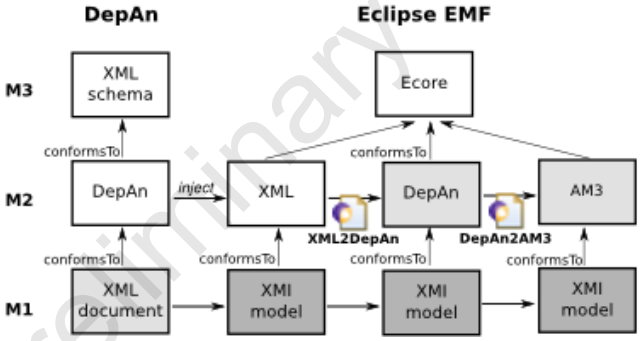
Fig. 6. A DepAn view of its javascript library



1. design the metamodel of DepAn tool domain
2. obtain a XML model from the DepAn XML file (.dpang) by XML injection
3. transform this XML model to the corresponding DepAn model (conforming to the **DepAn** metamodel we designed)
4. transform the DepAn model into an AM3 model
5. display the AM3 model in our visualization tool

The scheme of this Model-driven chain is presented in figure 7.

Fig. 7. DepAn to AM3 bridge



DepAn metamodel We design the DepAn metamodel corresponding to the DepAn tool file structure using Ecore tools of the Eclipse Modeling Platform[3].

XML extraction The ATL XML extractor produces a XML model (`.xmi`) of the original `.dpang` file.

From XML model to DepAn model Next is an excerpt of the XML2DepAn transformation written in ATL to translate XML model of an input `.dpang` file into the corresponding DepAn model:

[3] Cf. http://www.eclipse.org/modeling/

```
rule Root {
  from
    r : XML!Root (
      r.name = 'graph-model'
    )

  to
    root : DepAn!Root (
      elements <- Sequence{r.getChildrenByName(
                                    'java-type')}
        ->union(Sequence{r.getChildrenByName(
                                   'java-method')})
          [...]
    )
}

rule JavaType {
  from
    e : XML!Element (e.name = 'java-type')
  using {
      n : String = e.getStringAttrValue('name');
      id : String = 'java:' + n;
    }
  to
    jt : DepAn!JavaType (
      name <- n
    )
  do {
    thisModule.map <- thisModule.map->
                            including(id, e);
  }
}
```

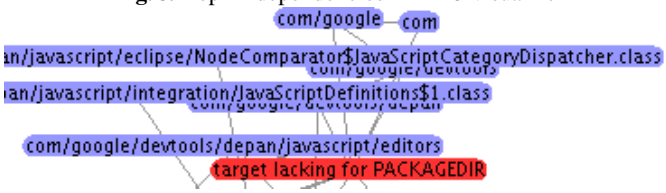

From strict DepAn model to AM3 model Then we need to rely DepAn model elements to **AM3** corresponding types.

As **AM3** metamodel is a generic one, its types are mainly abstract. In order to process the DepAn tool data as **AM3** elements, we defined a **DepAnAM3** metamodel which types inherits from **AM3** ones. Our goal is to display dependencies so we only use **AM3** identifiers as data into the types (here `JavaType` inherits from `Entity` and uses its `__id` and `identifier` attributes to store its name).

The transformation from DepAn effective model to the AM3 compatible model is a simple one-to-one translation.

Visualization The resulting AM3 compatible model can be process in our visualization tool chain. Figure 8 is a sample of the DepAn javascript library dependencies displayed in our viewer.

Fig. 8. DepAn dependencies in AM3 visualizer

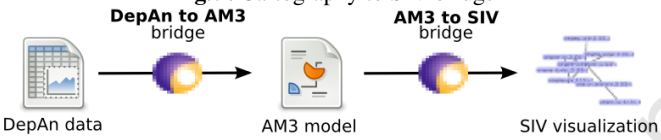


4.4 MDI chain between existing tools: DepAn to SIV

We have presented an export chain from AM3 data to SIV visualization tool and an extraction chain from DepAn data into AM3 model. The last step is to link together the second one to the first of these two chains to get a effective chain able to display depAn data into SIV. As all the transformations are written, the linking is only an additional call of the two previous chains.

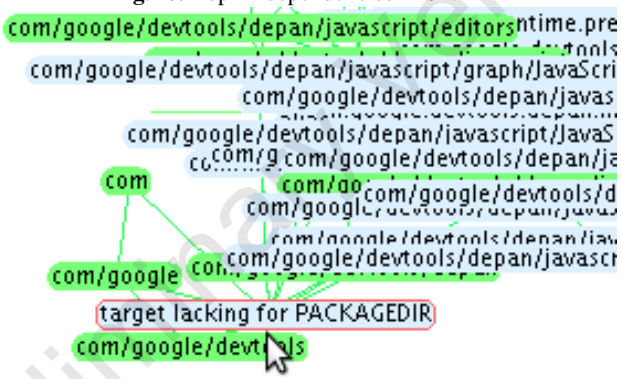
The scheme of this Model-driven chain is presented in figure 9

Fig. 9. Cartography to SIV bridge



Using the DepAn-to-AM3 chain and then the AM3-to-SIV chain as-is, we can display data from the DepAn analysis tool into the SIV tool. Figure 10 is a sample of the DepAn javascript library dependencies displayed in SIV viewer.

Fig. 10. DepAn dependencies in SIV



We have seen the ability of Model-Driven Engineering to deal with many problems and requirements like interoperability and genericity in a simpler way than previous paradigms. The new flexibility of the MDI approach answers some of the issues a technical domain like visualization may encounter. Given this, it opens interesting paths the research on dependencies visualization may now explore.

5 Discussion - separation of concerns

Visualization can help managing dependencies but existing viewers are dedicated to a specific domain (e.g. Java classes). Our approach proposes Model-Driven Interoperability (MDI) to transfer data from one tool to another one with further features. As the central metamodel is generic, this approach is domain-agnostic and gives an access to tools which have been design originally for different concepts.

Previously, tools concerned separate domains and could not share any data. Each approach was rewriting the visualization concepts others had explored because they

were not reusable. Using resources for that redoing work, the progresses were only marginal.

Now we have a way to pass dependencies data to a visualizer which can be domain-agnostic. Such an independent tool could make easier all visualization enhancements and research relative to dependency problems however are the concerned domains. Ways of presenting dependencies (both in hierarchical and flat forms) as well as user-friendly navigation between them remain open subjects which could be investigated for themselves without specific domains mind-focus.

6 Conclusion

We investigate in this paper a more flexible way for interconnecting visualization tools. For this purpose, we propose a binding tools chain relying on the recent Model-Driven Interoperability approach.

This paper brings a domain-agnostic solution which opens perspectives about a generic approach of dependencies problem. Using MDI approach could drive to enlarged experiments and sharing of knowledge about visualization specialized concepts.

Acknowledgements The present work is being supported by the CESAR European ARTEMISJU project and the French ANR IDM++ project, and relies on the AM3 tooling developed within the MODELPLEX European IST FP6 research project.

References

1. Code Analysis Plugin. http://cap.xore.de.
2. DEPendency vizualization and ANalysis tool. http://code.google.com/p/google-depan.
3. The Graph Exploration System. http://graphexploration.cond.org.
4. The GraphML File Format. http://graphml.graphdrawing.org.
5. J. Bézivin, H. Bruneliere, F. Jouault, and I. Kurtev. Model engineering support for tool interoperability. In *Proceedings of WiSME*.
6. J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of Java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 91–94. ACM, 2008.
7. D. Harel and I. Segall. Visualizing inter-dependencies between scenarios. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 145–153. ACM, 2008.
8. R. Lintern, J. Michaud, M. Storey, and X. Wu. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. In *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003.
9. J. Malnati. X-ray-an eclipse plug-in for software visualization. *Bachelor Project. Lugano University*, 2007.
10. R. Martin. OO design quality metrics. *An analysis of dependencies*, 1994.
11. R. Soley and OMG. Model Driven Architecture, 2000.
12. A. Telea and L. Voinea. An interactive reverse engineering environment for large-scale C++ code. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 67–76. ACM, 2008.