Domain Networks in the Software Development Process

Ulf Bergmann[1] and Julio Cesar Sampaio do Prado Leite[2]

[1] Departamento de Engenharia de Sistemas, Instituto Militar de Engenharia,
Praça General Tibúrcio 80, Rio de Janeiro, RJ, 22290-270, Brasil
ulf@ime.eb.br

[2] Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 225 Rio de Janeiro, 22453-900, Brasil
www.inf.puc-rio.br/~julio

Abstract. Domain Network (DN) is a set of domains interconnected by transformations. A previous implementation of this concept has been done by using the Draco-Puc machine, a software system that provides a very powerful transformation engine. In order to obtain the integration of this DN with approaches that use XML related technologies, we present a new DN implementation that allows an XML application to become a Draco-Puc domain. We believe that in this way we can use the power of Draco-Puc and gain a broadly use of DN by enabling the use of XML applications.

1 Introduction

Domain Network (DN) is a set of domains interconnected by transformations. Each domain is defined by its Domain Specific language (DSL) and by transformations to other domains. In a previous work[1] the concept has been defined and used in the generation of reverse engineering tools.

Using DN in software generation has two main advantages. First, while a usual software generator uses a single refinement step, a DN based generator provides more effective reuse by using several refinement steps. Second, we don't need to implement a new DSL from scratch because we create the transformations to other domains that are represented as DSLs.

An impediment to a broader use of DN is that their actual implementation uses an academic restrict software system - the Draco-Puc machine. The Draco-Puc has a very powerful transformation engine and it is a good choice to implement DN but we need to connect our DN implementation to an industrial standard.

The Extensible Markup Language (XML) and related technologies are broadly used and there are several applications in use today which have been developed using

C. Gacek (Ed.): ICSR-7, LNCS 2319, pp. 194–209, 2002.

this technology. In particular, the XML Metadata Interchange Format[1] (XMI) is used by many tools (like Rational Rose) to express the content of OO models in a format that allows the interchange with other tools.

In this work we will propose a more powerful implementation of DN by combining the Draco-Puc with the XML related technologies, allowing that an XML application can be a Draco-Puc domain. We believe that in this way we can use the power of Draco-Puc and at the same time, make possible that XML based applications be treated by a powerful transformation system.

In Session 2, three implementations of the DN concept will be presented: the Draco-Puc approach (Session 2.1); an XSL-Based approach (Session 2.3); and the combined approach (Session 2.3). In Session 3, examples of the DN use in the software development process will be shown. Finally, Session 4 brings the conclusions where this work's contributions and future work are pointed out.

2 Domain Network

A Domain Network[1][2] is a set of domains interconnected by transformations. Each domain encapsulates the knowledge of a specific area and is described by a domain specific language (DSL). We divide the domains in the network as: application domains, which express the semantics of real world domains or problem classes; modeling domains, which are domains built to help the implementation of application domains; and executable domains, which are domains to which a generally accepted translator exists, as it is the case of well-known programming languages.

Programs written in an application domain language need a refinement process to, across modeling domains, reach an executable language. This refinement process is made by applying transformations from a source to a target domain through a DN.

The use of DN in software generation has two main advantages. First, while a usual software generator uses a single refinement step, a DN based generator provides a more effective reuse by using several refinement steps. Second, we don't need to implement a new DSL from scratch because we create the transformations to other domains that are represented as DSLs.

In the next Sub-Sessions three ways which can be used to implement the Domain Network concept will be presented: the Draco-Puc approach; an XSL-Based approach; and a combination of both.

2.1 The Draco View of Domain Network

The Draco-Puc machine is a software system based on the construction of domains. A Draco-Puc domain must contain syntax, described by their specific grammar, and semantics, expressed by inter domains (vertical) transformations. Draco-Puc is

[1] XMI specifies an open interchange model defined by the Object Management Group (OMG) that is intended to give developers working with object technology the ability to exchange programming data over the Internet in a standardized way.

composed of: a powerful general parser generator, a prettyprinter generator and a transformation engine. The executable domains (like C) do not need semantics reduction since there is already an available compiler/interpreter.

Figure 1 shows an example of the domain construction in Draco-Puc. To create the lexical and syntactic specifications we use an internal Draco domain called *GRM* (grammar domain). This domain uses an extended BNF. Embedding commands from the Draco domain *PPD* (prettyprinter domain) in the syntactic specification provides the prettyprinter specification. The file *cobol.grm* in Figure 1 shows a partial implementation for the Cobol domain. The semantics for the domain are specified by transformations to other domains. These transformations are described using the Draco domain *TFM* (transformation domain). Basically, a transformation has a recognition pattern (the *lhs*) and a replacement pattern (the *rhs*): when the *lhs* matches any part of the original program it is replaced by the *rhs*. The file *cobol2c.tfm* shows an example of the Cobol to C transformation.

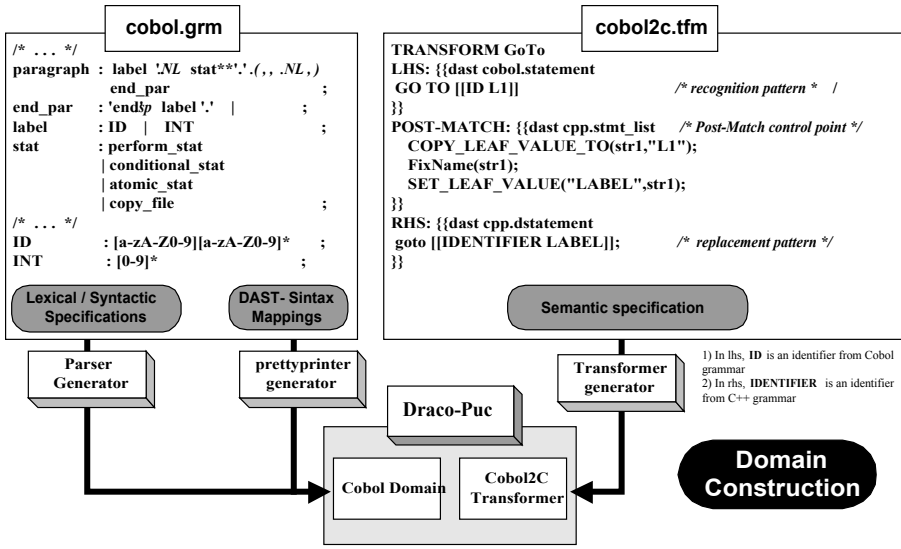

Fig. 1. Domain Construction in Draco-Puc

What makes Draco-Puc transformations more powerful is their very flexible structure. Figure 2 shows the entire transformation structure. Before and after the usual matching sides of a production rule we have 5 possible control points. *Pre-Match* is activated each time the *lhs* is tested against the program. *Match-Constraint* is activated when there is a bind between a variable in *lhs* and a part of the program. *Post-Match* is activated after a successful match between the program and the *lhs*. *Pre-Apply* is activated immediately before the matched program section is substituted by the *rhs*. *Post-Apply* is activated after the program has been modified.

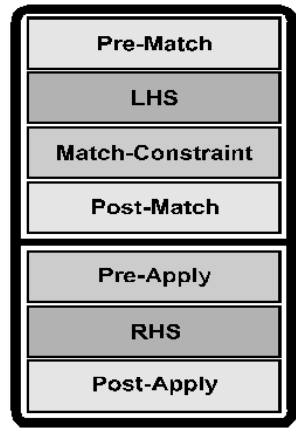

Fig. 2. Transformation Structure

Figure 3 shows an operational description of Draco-Puc main components. The parser for the original program is used by Draco-Puc to convert the program into the internal form used in Draco-Puc (DAST – Draco Abstract Syntax Tree). Transformations are applied to modify the DAST and, at any time, we can use the pretty-printer to unparse the DAST.

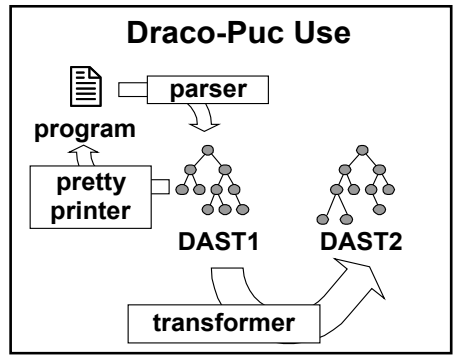

Fig. 3. Draco-Puc main components

Figure 4 shows the actual domain network available in Draco-Puc Machine. The main domains are briefly described in Table 1. This DN has been successfully used in previous work to generate reverse engineering tools[1].

2.2 The XSL Approach towards Domain Network

This approach uses several W3C technical recommendations[2] to implement a DN. Each domain is defined using a Document Type Definition (DTD) or a XML Schema to specify the syntax. The syntax domain doesn't define a real DSL, but a structure of XML documents for this domain.

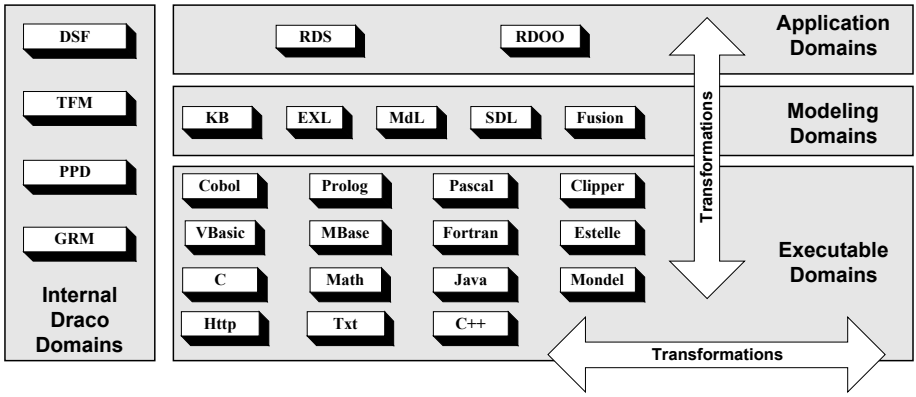

Fig. 4. Draco-Puc Domain Network

Table 1. Draco-Puc domains

Domain	Description
GRM	Describe the domain language syntax and its lexicon
TFM	Specify transformations
PPD	Specify the prettyprinter
RDOO	Recover Design from OO Systems
RDS	Recover Design from Structured Systems
KB	Manipulate a Knowledge Base
EXL	Specify source code extractions
MdL	Specify the visualization of information extracted from source code
SDL	Define a Graphics User Interface
C++, C, ..	Programming Languages

The semantic of the domain is expressed by a set of structural transformations defined in an XSL (Extensible Stylesheet Language). A stylesheet consists of a series of templates, together with instructions based on XPath expressions (the recognition pattern) which tell an XSL processor how to match the templates against nodes in a XML input document. When a node matches the recognition pattern, the instructions

[2] The W3C (World Wide Web Consortium) technical recommendations used here are: the XML - Extensible Markup Language; the DTD - Document Type Definition; the XML Schema; the XSL - Extensible Stylesheet Language; and the XPath - XML Path Language, all available at http://www.w3c.org/TR

in the template are executed in an output XML file. Figure 5 shows an example of the CRC[3] (Class Responsibilities and Collaborations) domain constructed by using this approach. The file *crc.dtd* defines the structure of a XML used to specify a CRC card and the *crc2java.xsl* stylesheet has a set of templates which generate Java code for the class in the card.

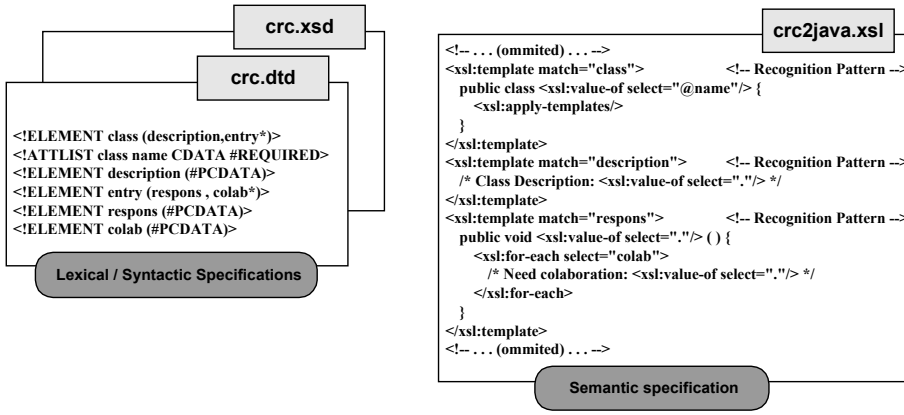

Fig. 5. An example domain in XSL-based approach

To put all these elements to work we need a DOM Parser[3] to read an XML file and construct the DOM Tree, and a XSL Engine to apply the stylesheets. Figure 6 shows an implementation of the XSL-Based approach. The file *sourceDef.dtd* defines the structure of the *source.xml*, which contains the program of the source domain. The DOMTree1 is build by applying the DOMParser on the source XML file. To obtain the correspondent tree in the target domain, the DOMTree2, a XSL Engine is used to apply the transformations in the stylesheet *transf.xsl*. Finally, the program in the target domain (*target.xml*) is obtained after applying another stylesheet (*view.xsl*) which formats the tree in the target domain.

The DN for this approach is showed in Figure 7 and enables the generation of Java code starting from an extended subset of the requirements baseline model proposed by Leite[6]. The domain Scenario is used to describe the situations in the user environment. Each scenario is a structure composed of goal, context, resources, actors and episodes for the scenario. The domain LEL (Language Extended Lexicon) is used in the elicitation of the user language in the macrosystem. The domain CRC (Class Responsibilities and Collaborations) shows the system classes and the distribution of functionality among these classes.

The program wrote in the domains Scenario and CRC can be transformed into the executable domain Java using the descriptions stored in the LEL domain. To make this transformation, we use the intermediate modeling domain XMI.

[3] A DOM Parser uses the Document Object Model (DOM) view of an XML document. The DOM specification can be found at http://www.w3c.org/TR/REC-DOM-Level-1. The parser used here is the Xerces from the Apache XML Project, available at http://xml.apache.org

2.3 Integrating Draco-Puc and XML Based Technologies to Implement a Domain Network

The preceding approaches have some issues that limit the full application of DN in the software development process. The Draco-Puc approach has a very flexible and powerful transformation mechanism but it is not broadly used. The XSL-Based approach uses a set of technologies already used by the industry but is not very flexible. In Table 2 we show a more complete analysis of the approaches.

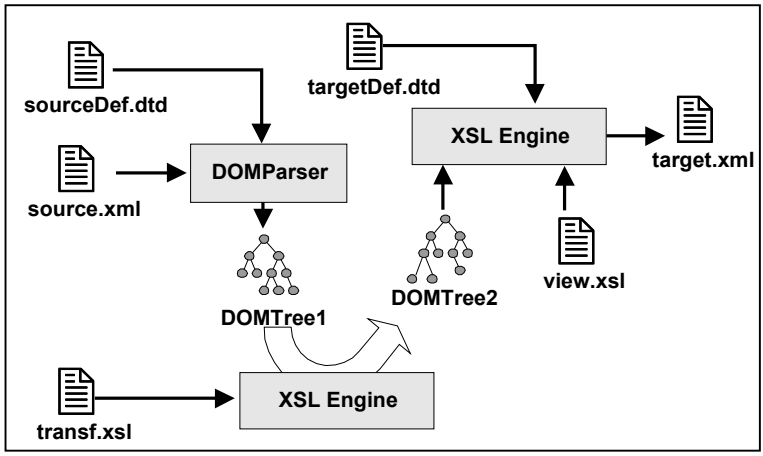

Fig. 6. Use of the XSL-Based approach

In order to avoid these problems we decide to design a DN which integrates these two approaches by providing a gateway between them. Figure 8 shows this gateway, implemented by a set of Draco-Puc transformations (see file *dtd2grm* in Fig. 8). This gateway permits every domain in the XSL-Based approach, the *DomainA*, to have an equivalent Draco-Puc domain, the *DomainA'*. The creation of the equivalent domain *DomainA'* is done by applying either the transformation *dtd2grm.tfm*, if *DomainA* syntax is defined by a document type descriptor (DTD), or the transformation *xsd2grm.tfm*, if the syntax is defined by an XML Schema file (XSD). The semantics of the *DomainA* (XSL transformations) is transformed into the equivalent Draco-Puc transformations (TFM) by applying the *xsl2tfm.tfm* transformations.

Figure 9 exemplifies the use of the gateway concept: the syntax for the domain CRC (Class Responsibilities and Collaborations) are originally described in the XSL-Based approach using the Document Type Description (file crc.dtd); and the correspondent syntax in Draco-Puc (file crc.grm) are obtained by applying the transformation *dtd2grm*. In the same way, the semantics specified as XSL are translated to Draco-Puc transformations. This combined approach to DN permits every XML application to be a Draco-Puc domain or, in others words, we can use the Draco-Puc Domain Network to manipulate XML applications as showed in figure 10 where a set of class described in CRC syntax are transformed to Java source code using either the XSL-Based approach or the Draco-Puc approach.

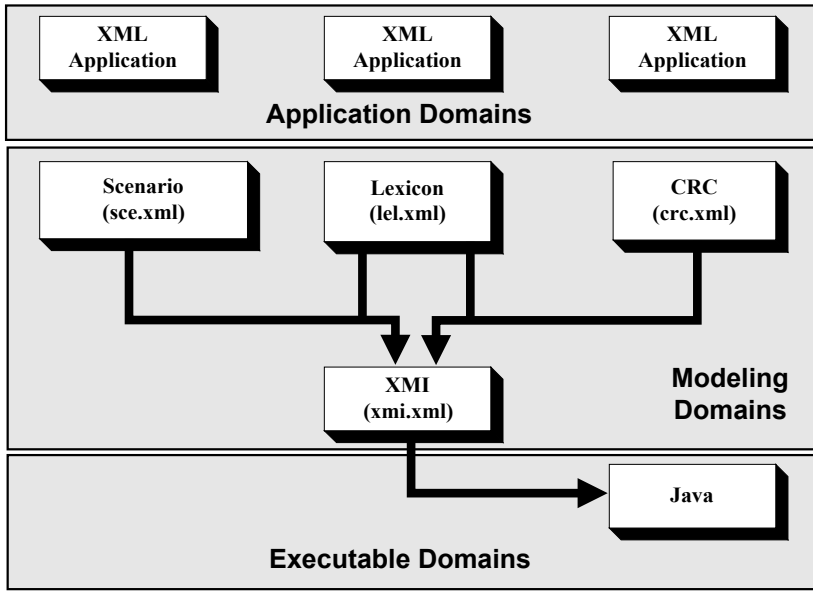

Fig. 7. DN for the XSL-Based approach

Table 2. Draco-Puc and XSL-Based approach comparison

	Draco-Puc	XSL-Based
Domain Language	- Uses the same words as the real world	- Uses XML which is not the user language, it is a markup language
Syntactic Specification	- Uses BNF-like syntax which is more flexible	- Can use DTD or XML Schema. Every document must be a tree
Semantic Specification	- Can use any other domain language within a transformation - Has many different control points to insert code: Pre-Match, Match-Constraint, Post-Match, Pre-Apply, the rhs rule and the Post-Apply	- Can use extensions to access Java code - Only one local to put code: de rhs rule
Use	- Restricted to a small set of researchers - Not used by the industry	- Large set of researchers - Used by many real applications

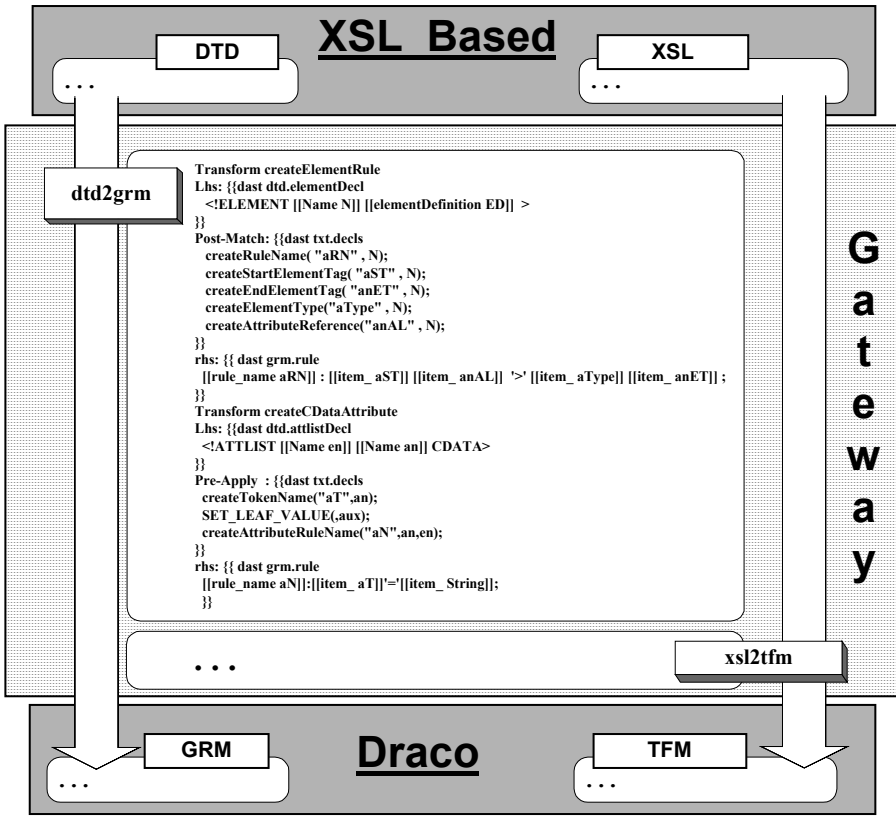

Fig. 8. Draco-Psuc and XSL-Based combined approach

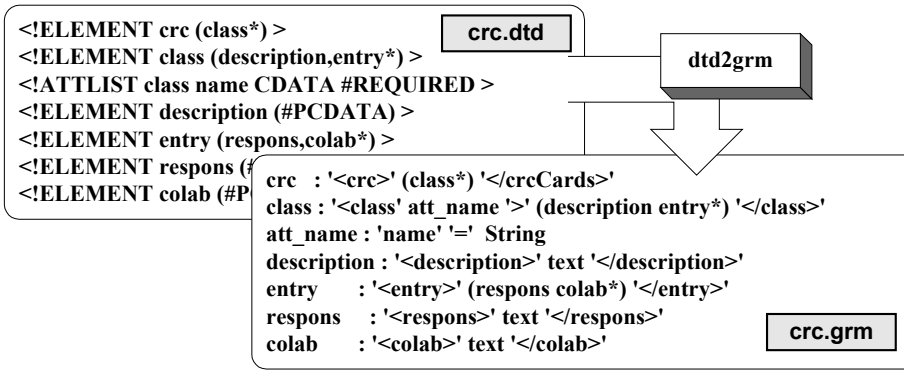

Fig. 9. Building the Draco-Puc domain using the gateway concept

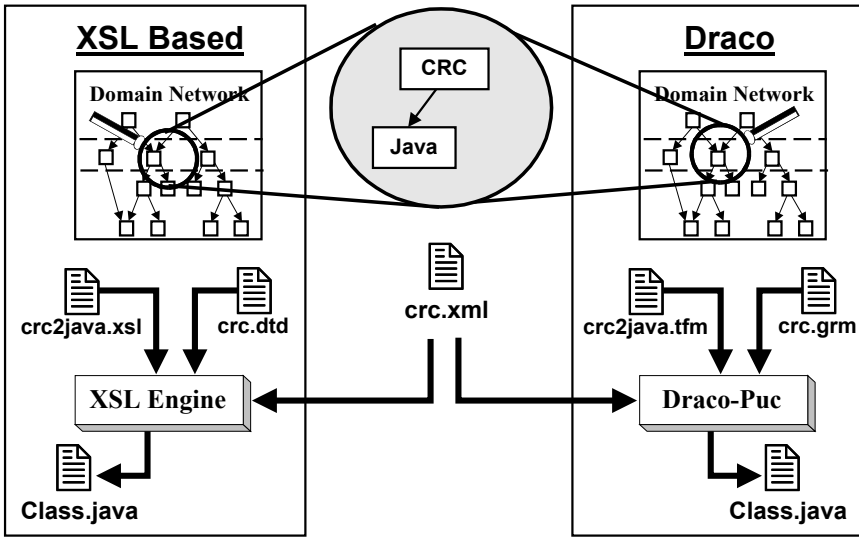

Fig. 10. Using the domain network combined approach

3. Examples of Domain Networks in the Software Development Process

Domain networks in the software development process has a very large range of applications. We highlight two cases of the DN use: identification of models regions where some predefined heuristic can be applied (the identify-and-apply application) and traceability. These cases will be presented and will be compared with other proposed approaches.

Another example shows the role of DN in software maintenance and can be characterized by extracting information from a legacy system by analyzing their source files and by making some transformations to solve a particular problem. This example has been presented previously by the authors in [1].

3.1 Identify-and-Apply Applications

In this kind of applications we group the work done by the developer when he identifies a particular situation an apply a correspondent rule, where the situation and the rules are previously specified by experts. Examples of these applications are heuristics, a guidance for making design decisions, Design Patterns[4], which provides solutions to a recurring problem, and Anti-Patterns[5], which describes a bad solution to a particular problem.

The example presented in Figure 11 shows how to use a DN to apply the heuristic *from the scenario model an actor can be derived in a class*[6]. The Draco-Puc apply a

transformation which finds an actor in the scenarios, asks the developer if he wants to create a class for it and creates the correspondent class.

The source code transformation (using the Draco-Puc TFM domain, a domain to define transformations) which applies this heuristic is presented bellow. When the recognition pattern (lhs) matches some piece of code (an actor) the Post-Match control point is executed to ask the developer if he wants to create the correspondent class. Finally, the replacement pattern is inserted in the crc.xml file creating a new class. Figure 12 shows the DN used in the identify-and-apply example.

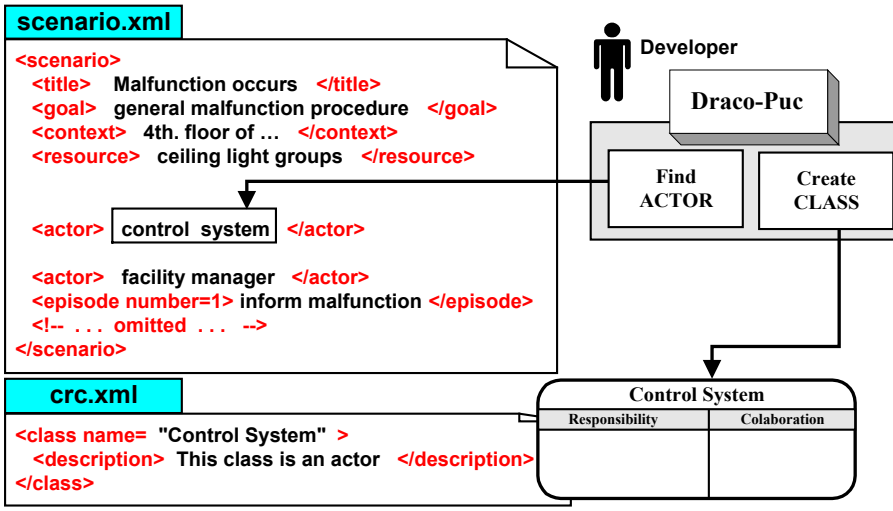

Fig. 11. Apply heuristic example

```
TRANSFORM createClassFromActor
LHS: {{dast scenario.actor
   [[ID name]]   /* recognition pattern */   }}
POST-MATCH: {{dast cpp.stmt_list /* control point */
   COPY_LEAF_VALUE_TO(str1,"name");
   if(askToUser("Create class for" + str1)) {
     FixName(str1);              /* Normalize string */
     SET_LEAF_VALUE("CLASSNAME",str1);
   } else SKIP_APPLY;
}}
RHS: {{dast crc.class       /*  replacement pattern */
<class name= [[ID CLASSNAME]]>
   <description>This class is an actor</description>
</class>
}}
```

Related Work

OOPDTool[7] is a tool designed to support design expertise reuse by detecting good (Design Patterns) and bad (anti-patterns) OO design constructions and suggesting some hints for a better solution. This tool uses deductive databases to store Design Patterns, Heuristics and Anti-Patterns, as well the facts extracted from a design model recovered from OO source code. To identify where the recovered design model implements some (anti-)pattern or heuristic, it uses an inference machine and machine learning techniques. Our approach has a more powerful way to identify patterns and heuristic application points: first, we can use a Knowledge Base in Draco-Puc machine (it has an inference machine implemented); second, we can use our own transformation engine to recover the design models; and finally, our search engine has a more power to identify points in the design models where patterns and heuristics can be applied. Our approach also allows an automatic application of the patterns.

Some others proposed approaches[8][9], but don't provide an automatic way to identify and/or apply heuristics or (anti-)patterns.

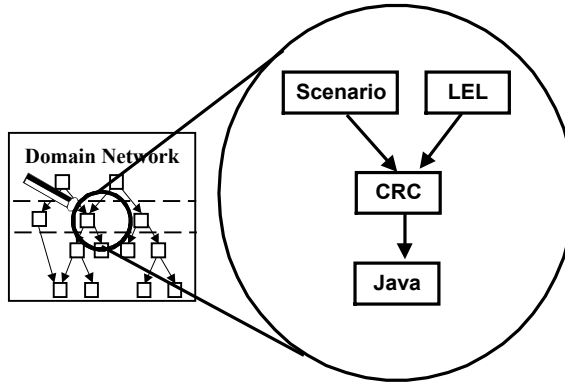

Fig. 12. Domain network used

3.2 Traceability

In their work about reference models for requirements traceability, Ramesh and Jarke[10] define a traceability system as a semantic network in which nodes represent objects among which traceability is established through links of different types and strengths, and classify these links as: Satisfaction Links used to ensure that the requirements are satisfied by the system; Evolution Links used to document the input-output relationships of actions leading from existing objects to new or modified objects; Rationale Links which represent the rationale behind objects or to document the reasons for evolutionary steps; and Dependency Links between objects.

Murphy et al. [11] put that despite the effort spent in generating and validating trace information, they are often mistrusted since they may have become obsolete due to separate evolution of models and systems. We believe that the only approach to

avoid this issue is by automatically generating trace information. We will show how to use DN in the implementation of a traceability system which provides an automatic way to capture trace information about the evolutionary and rationale links.

Figure 13 shows this system. It stores trace information as transformations between models and has three main activities: Generate Trace Transformation, Create Plan and Refine.

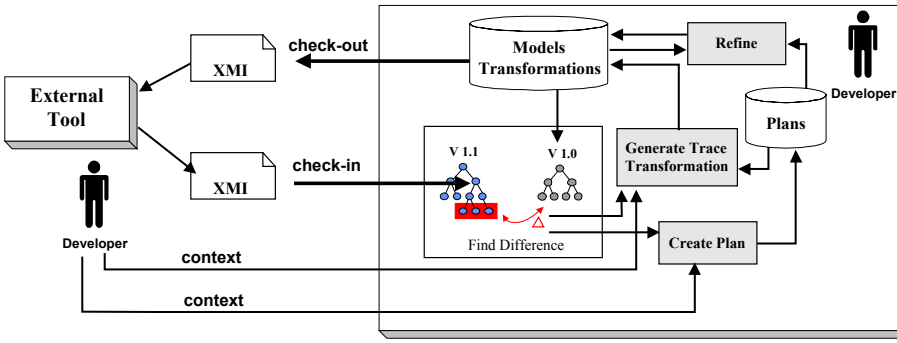

Fig. 13. DN applied to traceability

The Generate Trace Transformation is done by identifying the differences (the delta) between an artifact before it check-out from the system and after the check-in. The delta corresponds to the changes made by the developer. Next, the system applies a Plan Recognition[12] technique into the delta and the context information about the current work of the developer, to infer the intentions of the developer. Finally, it generates a new transformation which can be applied in the original artifact to create the modified artifact. This transformation is then stored in the transformations base. If any plan is selected during this activity, a new plan can be created by the developer and used in other interactions.

The Refine activity provides a way to reuse the stored plans to refine a model. The developer specifies the work context and the system selects the plans which can be applied. The developer selects a plan and the system applies it to the model.

Figure 14 shows the DN used in the traceability application.

Related Work

Egyed [13] presents a scenario driven approach which generates trace information by observing test scenarios that are executed on running software systems. This approach requires the design models and an executable version of the software system. The generated information is restricted to requirements traceability and shows which model elements implement each test scenario. Our approach can be used in any phase of the development and provides not only requirements traceability, but also design traceability which shows the relationship between the design models.

Pinheiro[14] uses a formal approach to maintain traceability information as relations between artifacts. This information must be manually entered by the developer. The DN approach provides an automatic way to capture these pieces of information.

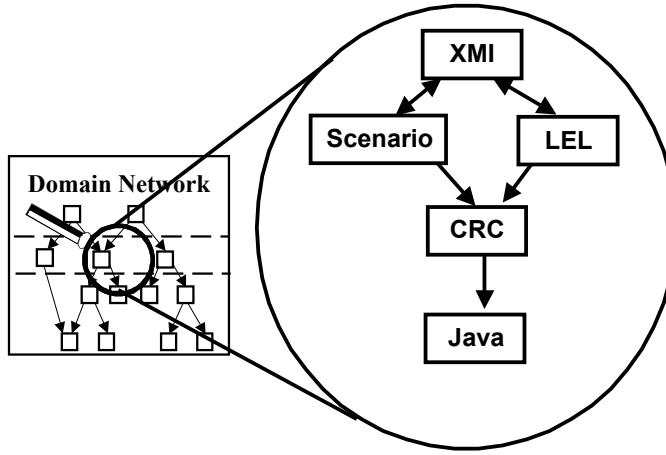

Fig. 14. Domain network for the traceability application

Haumer et. al.[15] propose the use of extended traceability to manually recording concrete system usage scenarios using rich media and interrelating the recorded observations with the conceptual models, in a manner which helps the reviews.

A very similar approach to ours is presented by Antoniol et. al. [16] which compute delta between versions to help the user to deal with inconsistencies by pointing out regions of code where differences are concentrated. They use an intermediate representation, the Abstract Object Language (AOL), and compute de delta by analyzing this code. This work differ from ours because we compute the delta directly by analyzing the abstract trees for each version and, consequently, we don't lose any information when translating to an intermediate representation. Also, the AOL used by Antoniol handle only aspects related to Class Diagrams and we can compute delta in any model. The use of the inferred information also differs because we want to use it to maintain the relationships between models and not only to show inconsistencies between versions to the user. Antoniol approach is only for OO systems and ours can be used by others development methods as well.

4. Conclusion

Integrating XML related technologies into the concept of Domain Network opens a new opportunity to the use of transformation technology. Since XSL has limitations,

we believe that using the Draco-Puc transformation engine on XML languages is a considerable gain in expressiveness power. By using the concept of Domain Network and of Domain Specific Languages we have shown that the integration can be performed in a seamless manner.

We have shown that the DN concept can be applied to different aspects of a software development process. In particular we have shown the capability of manipulating software architectures (design patterns) and software processes (traceability).

Our experiments so far are very positive, but we need to augment our transformation library and use this concept in other applications, mostly some large and complex to verify its applicability in the real world. Future work will continue to try to make the Draco-Puc transformation system more usable and available for a more widespread use. We understand that making possible to use Draco-Puc transformation engine on XML based descriptions will empower software developers in several ways.

References

1. Bergmann, U., Leite, J.C., From Applications Domains to Executable Domains: Achieving Reuse with a Domain Network, in Proceedings of the 6th International Conference on Soft-ware Reuse, 2000.
2. Neighbors, J., Software Construction Using Components, PhD thesis, University of California at Irvine, 1980. http://www.BayfrontTechnologies.com/thesis.htm
3. Wirfs-Brock, R., Wilkerson, B., Wiener, L., Designing Object-Oriented Software, Prentice Hall International, Englewood Cliffs, NJ, 1990.
4. Gamma, E., et. al., Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley Co., 1994.
5. Brown, K., et. al., Anti-patterns - Refactoring Software, Architectures and Projects in Crisis, Wiley Computer Publishing, 1998.
6. Leite, J.C., et. al., Enhancing a Requirements Baseline with Scenarios, in Proceedings of the Third International Symposium on Requirements Engineering, IEEE Computer Society, pp. 44--53. (1997).
7. Correa, A., Werner, C., Zaverucha, G., Object Oriented Design Expertise Reuse: An Approach Based on Heuristics, Design Patterns and Anti-patterns, in Proceedings of the 6th International Conference on Software Reuse, 2000.
8. Cinnéide, M., Nixon, P., Program Reestructuring to Introduce Design Patterns, in Proceedings of the Workshop on OO Software Evolution and Reengineering, ECOOP98.
9. Zimmer, W., Experiences using Design Patterns to Reorganize an Object Oriented Application, in Proceedings of the Workshop on OO Software Evolution and Reengineering, ECOOP98.
10. Ramesh, B., Jarke, M., Toward reference Models for Requirements Traceability, IEEE Transactions on Software Engineering, 27(1), 2000.
11. Murphy, G. C., Notkin, D., and Sullivan, K.: "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," in Proceedings of the 3th ACM SIGSOFT Symposium on the Foundations of Software Engineering, New York, NY, October 1995, pp.18-28.

12. Kautz, H.A., Allen, J.F., Generalized Plan Recognition, in Proceedings of the 5th Nat. Conf. AI, pp 32-37, 1986.
13. Egyed, A., A Scenario-Driven Approach to Traceability, in Proceedings of the International Conference on Software Engineering, 2001.
14. Pinheiro, F., Goguem, J., An Object Oriented Tool for Tracing Requirements, IEEE Software, 13(2), 1996.
15. Haumer, P., et al., Improving Reviews by Extended Traceability, in Proceedings of the 32nd Hawai International Conference on Systems Science, 1999.
16. Antoniol, G., Canfora, G., De Lucia, A., Maintaining Traceability During Object-Oriented Software Evolution: a Case Study, in Proceedings of the International Conference on Software Maintenance, 1999.