Exploring Inter-Module Relationships in Evolving Software Systems

Mircea Lungu *and* Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland

Abstract

Many of the existing approaches to reverse architecting –the reverse engineering of the architecture of software systems– are based on software exploration tools which provide interactive ways of exploring the system. These tools start with high-level views of the system and refine them with drill-down techniques applied on the high-level entities such as modules and packages, leaving aside valuable information contained in the dependencies between them.

In this article we argue that the visualization of inter-module relations bears great potential for supporting the understanding of large evolving software systems. We present two concrete examples of such visualizations. The first, The Semantic Dependency Matrix is a technique for displaying details about a dependency between two modules which groups together classes with similar behavior. The second, The Edge Evolution Filmstrip presents the evolution of an inter-module relation through multiple versions of the system. Based on our experience with the Edge Evolution Film Strip, we propose a pattern language for inter-module relationships. We exemplify both the visualizations and the pattern language with examples from two large open source software systems.

1. Introduction

The IEEE 1471-2000 standard defines software architecture as "the fundamental organization of a [software] system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."

An architecture can be recorded by a written description that is organized into one or more constituents called architectural views[18], such as the module view, the component-and-connector view, and the allocation view[3]. In the context of this paper we adopt the module view. However, even when the architecture is documented at the time of its development, evolutionary processes lead to a decay of the initial design and results in a separation between the *as-designed* and *as-built* architectures [6].

Because the architecture of the software systems is an important asset for many software engineering tasks such as migrations, impact analysis or feature additions, recovering the architecture is an important reverse engineering activity[18, 20, 19] which has to rely on the only truthful source of information: the source code.

Some of the existing approaches to architecture recovery are highly dependent on visualization and interaction [7, 17, 16, 24]. Out of these, a distinct class is the top-down exploration tools which use interactive exploration techniques to navigate hierarchical decompositions of software. Many existing exploration approaches put little emphasis on the importance of the dependency edges between the modules. Tools such as Bauhaus[11] and Rigi[15] that recover dependency graphs from the systems usually focus on the nodes in the graph and represent inter-module dependencies as mere arrows.

In this paper we argue that dependencies are important for the understanding of the system and that more elaborate representations[12] of the dependencies are useful to understand inter-module relationships which ultimately represent architectural ties. We present a technique for the visualization of inter-module dependencies based on multi-version analysis called *film strip*. Visualizing the inter-module relationships by means of the film strip leads to the detection of evolutionary patterns in the relationships between modules. We apply the technique in a reverse engineering context by analyzing large open-source systems.

Structure of the paper. In Section 2 we introduce our model of inter-module relationships and the way that we use it in the context of software exploration. In Section 3 we present the Semantic Dependency Matrix and in Section 4 the Edge Evolution Film Strip. Section 5, presents a set of patterns of evolution for inter-module relations that we have defined based on our experience of applying the Edge Evolution Filmstrip on several large software systems. In Section 6 we discuss our approaches . We end by looking at related work in Section 7 and conclude by summarizing our contributions and looking at possible continuations of the

work in Section 8. Appendix A presents details about ArgoUML and Azureus, the two systems we analyzed.

2. Software Exploration

Some programming languages feature modules as first-level entities, while others provide constructs that do not directly map to a module in the sense of an architectural component. In these cases, the module structure is piggybacked on other mechanisms such as Java packages or C/C++ directories. Our only assumption about modules is that they are containers for other modules[1] or software artifacts.

Based on our assumption, the modules are only containers, and there are no *explicit dependencies* between them, such as the dependencies between Java packages defined by the *import* declaration. On the other hand, *explicit dependencies* exist between the artifacts contained in the modules, such as class inheritance, method invocations and variable accesses. Based on these low-level, explicit dependencies, we define high-level, *implicit dependencies* between modules as aggregations of the low-level dependencies. In this work we distinguish between two types of implicit relations between modules: *dependencies* and *relations*.

Inter-Module Dependencies. We consider a high-level, implicit dependency *of type* T between a client module (M_c) and a provider module (M_p) to be a relation defined by the set of all the relations of type T between all the low-level artefacts contained in M_c and all the low-level artefacts contained in M_p. It follows that there are various types of implicit dependencies between two modules (*e.g.*, inheritance dependencies, invocation dependencies, *etc.*) and that a dependency between two modules is directed. From the various properties that can be defined for an implicit dependency, we use the *cardinality* of the set of low-level abstracted dependencies as a measure of the strength of the dependency between the two modules.

Inter-Module Relations. We define the set of all the implicit dependencies between the two modules as an *inter-module relation*. Due to the recursive nature of our definition of module, implicit relationships exist between modules residing at any abstraction level in the module hierarchy. A relation between two modules exists if there is at least one implicit dependency between them.

Figure 1 presents two modules between which the low-level dependencies can be abstracted to three implicit dependencies ID_1, ID_2 and ID_3.

1 The contained modules do not need to be necessarily of the same type as the containers, *e.g.*, in Java packages are containers for classes and other packages while classes are containers for methods and attributes.

We argue that detailed visual representations of inter-module relations and inter-module dependencies are useful for the understanding of the system and its dynamics.

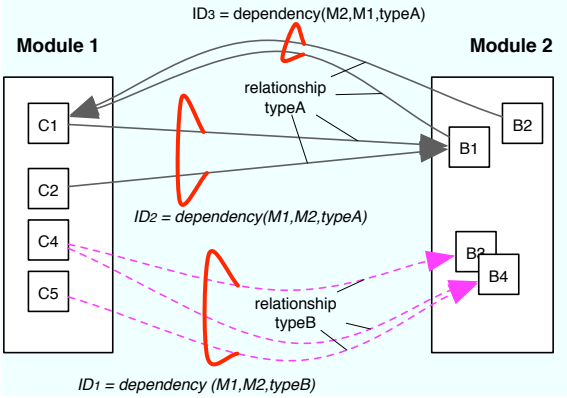

Figure 1. The relationship between Module 1 and Module 2 is the set containing the three implicit dependencies (ID_1, ID_2 and ID_3).

Exploration perspectives. In this paper we provide examples from our exploration prototype named Softwarenaut[13, 22] (see Figure 2), aimed at exploring hierarchical decompositions of software by starting with a high level view and continuously refining it using exploratory operations and filtering. The goal of the exploration process is to obtain views which are relevant for the architecture of the system.

Softwarenaut provides complementary perspectives on the analyzed system. The main perspective is the *Exploration View* where an enhanced graph-based representation is used to present views of the interactions between the modules in the system. In Figure 2 the Exploration View is the left panel in the figure. Each module is represented by a figure which combines metric and structural information by overlapping the representation of the module hierarchy as a tree on top of a colored square proportional to the size of the module. This representation allows for example us to observe that the biggest module in the analyzed system is *org.argouml.uml*. The module plays a central role in terms of interaction with other modules because of the many incoming and outgoing arrows.

The arrows between modules represent the relations between them. Every arrow between two modules represents an implicit dependency between the two modules. Different types of implicit dependencies are represented with different colors (red for the inheritance dependencies, black for the invocation dependencies). The width of each arrow is linearly proportional to the cardinality of the set of low-

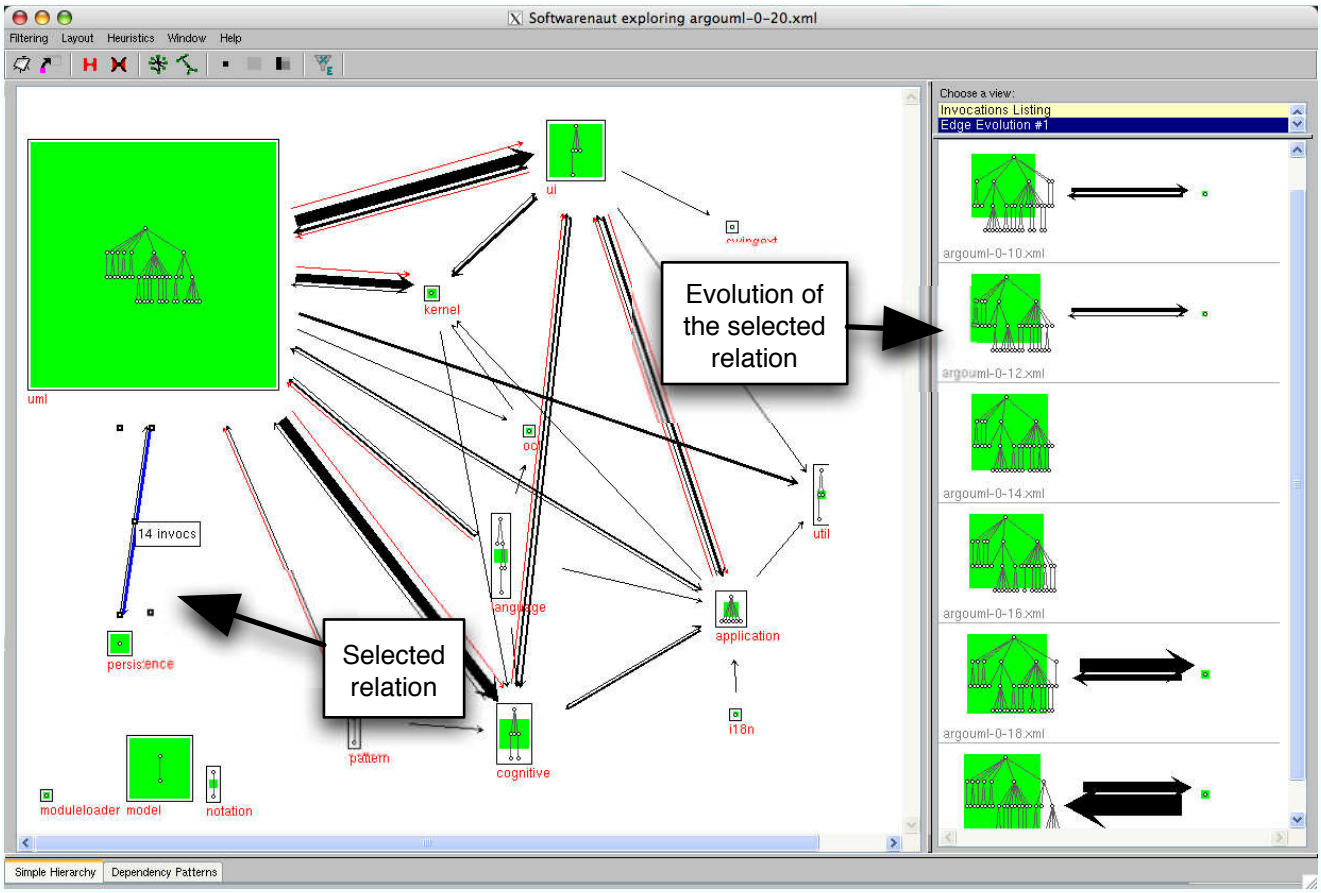

Figure 2. Softwarenaut exploring ArgoUML v.0.20. The edges are filtered by their age, with only the ones that exist in both the first (0.10) and the last (0.20) analyzed version visible. The red edges represent inheritance dependencies and the black ones invocation dependencies. The right panel presents the evolution through 6 versions of the selected dependency.

level dependencies abstracted in the corresponding edge.

To support the navigation in the exploration perspective, Softwarenaut provides the following operations:

- *Expand.* A node is replaced by the nodes representing its children and the edges between them.
- *Collapse.* A module node, together with all the nodes representing the sibling modules, is removed from the view and replaced by the parent module node.
- *Filter.* To tackle information overload, during the exploration process filters need to be applied such that only specific types of elements are visible. Filters can be applied to modules or edges and can be defined based on metric or structural properties. In the example from Figure 2 we applied a filter that hides all the dependency edges which are not present in both the first and the last versions of the system. This allows us to start the analysis with the relationships which are probably more important to the architecture of the system.

Complementing the main exploration perspective, the *Detail View* (the right panel in Figure 2) provides a wide range of detail views for the node or edge in focus. In this way, during the exploration of the system, the user can visually query various elements for details.

In this paper we argue for the importance of the detail views on both *relations* and *implicit dependencies* between the modules. In the following sections we present two such detail views: the first, the *Semantic Dependency Matrix* is aimed at providing a visual summary of an implicit dependency while the second, the *Edge Evolution Filmstrip* is targeted at visualizing the evolution of an inter-module relation.

3. The Semantic Dependency Matrix

Construction Principles. The Semantic Dependency Matrix is a visualization technique used for providing a high-level perspective on an implicit invocation dependency between two modules, and in the same time, a starting point for further exploration of the methods and classes involved in the dependency.

The matrix has two main properties: (1) it emphasizes the important properties of the classes and methods from the client and the provider modules that are involved in the dependency, and (2) it groups the client classes such that the ones which have similar interaction patterns with the provider, and therefore are probably semantically similar, are in spatial proximity.

The matrix is built by rendering the client classes in columns and the provider methods, grouped by class, in rows (alternate horizontal bands of gray and white represent classes). If a client class invokes a provider method at least once, we put a mark at the intersection of the corresponding column and row. The shape of this mark presents information about the provider method and its color intensity represents the number of times a given class invokes a certain method.

This information is useful in understanding the composition and the strength of the dependency. However, in order to understand the reasons behind a dependency we group the client classes based on their patterns of interaction with the provider. To do this, we define for every client class a signature which is a column in the matrix. Then, by hierarchically clustering the classes using as distance the hamming distance between signatures, and then arrange the client classes along the x-axis.

The matrix is interactive so the user can select classes or methods of interest and jump to the code or spawn new visualizations of them.

Interpretation. Figure 3 presents the Semantic Dependency Matrix of the dependency between the `com.elitis.azureus` and `org.gudy.azureus2.core3` modules in Azureus. Figure 3 illustrates how the construction algorithm groups together the client classes with similar interaction patterns. The group marked (1) consists of classes which only depend on a single method in the provider (the *printStackTrace* method). Group (2) represents a cluster of classes which depend mainly on a group of methods from a memory allocation class. The group marked as (3) uses a wide range of functions in the provider package. At closer inspection we found out that the client classes in this group are at the core of the system, such as AzureusCoreImpl (the class which initializes the system), DHTControllerImpl (class implementing the DHT subsystem), or PiecePickerImpl (class which manages the download of parts of torrents).

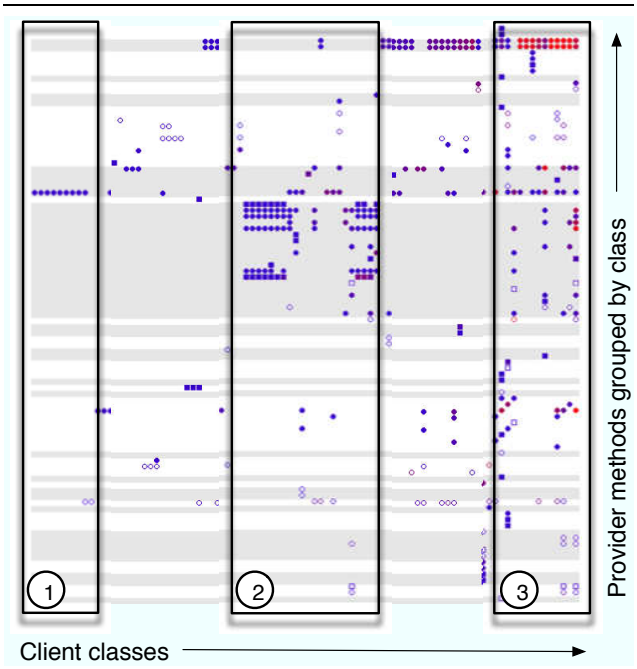

Figure 3. The Semantic Dependency Matrix for the dependency between com and org.gudy.azureus.core3 in Azureus

4. The Edge Evolution Film Strip

Construction Principles. The film strip is a composite visualization of the evolution of a *relation* between two modules. Figure 4 represents an enlarged view of the right panel from Figure 2, which presents a film strip of the relation between the ArgoUML modules `uml` and `persistence`[2]. The figure presents the way the relationship between the two modules evolves during six versions of the system (versions are aligned vertically from top to bottom in chronological order).

In the film strip, the arrows between the modules represent *implicit dependencies* of different types (the invocation dependencies are represented in black and the inheritance dependencies are represented in red). The width of the dependency arrows is proportional to the number of explicit, low-level dependencies abstracted in the corresponding implicit dependency. The representation of the width of the dependencies provides insight into the quantitative dynamics of the inter-package relationship.

Using this type of visualization one can understand the dynamics of a relation but if he wants to understand in de-

2 See the Appendix for details about the sizes and the choice of versions for the two discussed systems

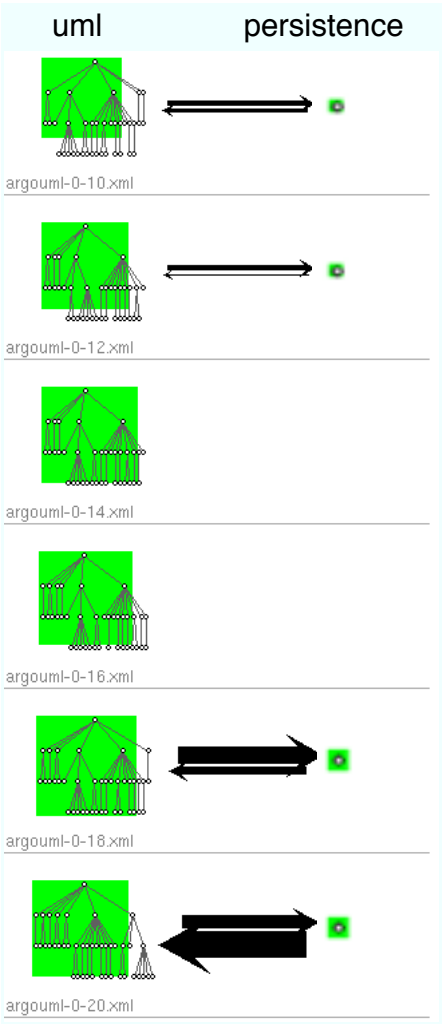

Figure 4. The film strip of the relation between org.argouml.uml and org.argouml.persistence.

tail the reasons for the existence of a specific implicit dependency (*i.e.*, a single arrow in the figure) he would have to use a more detailed visual representation, such as the Semantic Dependency Matrix presented in Section 3.

The need to switch between representations makes the interaction an important part of the Edge Evolution Film Strip. The user can select any of the edges and query it for its contents or spawn new exploration sessions on its contents.

Interpretation. Figure 4 represents a detail view of the dependency edge between `org.argouml.uml` and `org.argouml.persistence`, two of the packages in the ArgoUML case study.

The `uml` module is on the left side of the image and the `persistence` module is on the right. The difference in size between the two modules is due to the fact that the `uml` module is much larger (52KLOC in the last version) than the `persistence` module (3KLOC in the last version). We can see that while the `uml` structure and size developed slowly over time, the `persistence` module disappeared between versions 0-14 and 0-16. We can see how during the lifetime of the system, the relation between the two modules was continuously changing: if at the beginning there were weak invocation dependencies in both directions, later the dependencies disappeared and in the last two versions appeared again but stronger.

The film strip tells the story of the evolution of certain relationship between two modules. When the reverse engineer needs to understand a particular implicit dependency, he needs another visualization technique which will be specific to that type of dependency. In this case, by inspecting the dependency between the two packages in version 0.12 we found out that the `uml` package depends on `persistence` package mainly because of the functionality provided by the *DBLoad* and *DBStore* classes, which handle saving and loading a model from a database. By inspecting the dependency in version 0.18 when it appears again stronger, we found that this time one of the main actor classes in the provider (*i.e.*, persistence) is the *PersistenceManager* class, a singleton that handles saving and loading the models in various formats like XMI, UML, *etc.* The functionality remained the same, the implementation changed.

5. Inter-Module Relation Evolution Patterns

Using the film strip, we distilled a set of patterns with respect to the way that inter-module relations evolve. The patterns are not mutually exclusive, but they can be divided in two categories and inside the categories they are mutually exclusive. The two categories are:

1. *Age-related patterns*, based on the historical presence of an inter-module relation. We distinguish between (1) persistent, (2) recent, and (3) fossil.
2. *Dynamics-related patterns*, based on the semantics of the relations, *i.e.,* the contained dependencies and the way they evolve. We distinguish between (1) stable, (2) instable, and (3) intermittent.

Lifetime Relation. A lifetime relation exists both in the first and in the last analyzed versions of the system. Such relations are relevant for reverse engineering because they are very probably part of the original architecture of the system. This may be especially true for systems developed using agile methodologies which encourage the implementation of core functionality at the beginning of the systems life [4]. Both relations a) and c) from Figure 5 are lifetime relations while b) is not because there was no relation between

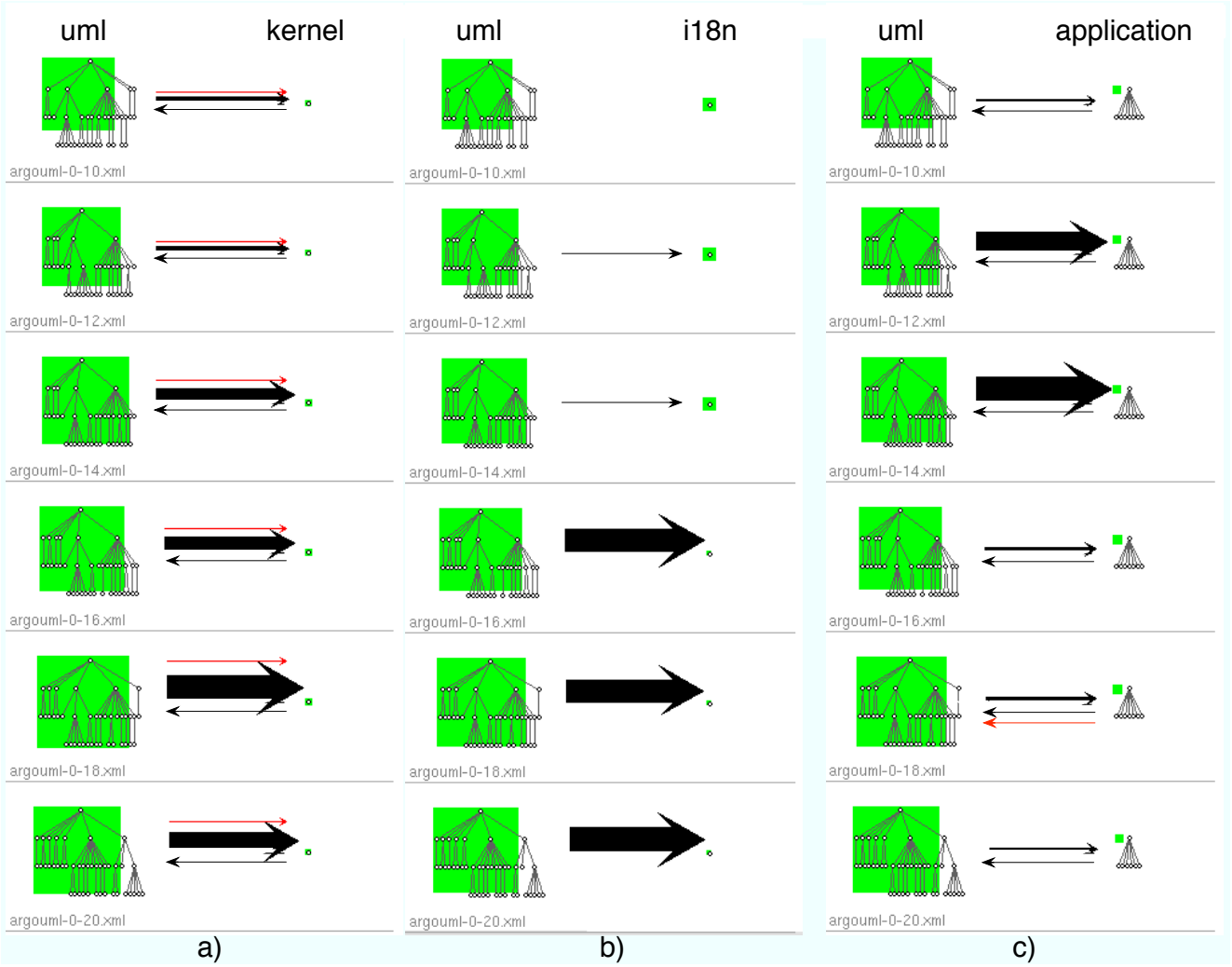

Figure 5. The evolution of the relations between org.argouml.uml and three other packages showcases three types of relationships: a relationship which is intermittent (a), a relationship which appears later in the project (b) and a relationship which is very instable (c)

the modules in the first analyzed version of the system. This pattern does not say anything about the evolution of the individual dependencies composing the relation; it only captures the fact that a relation has existed between the two modules throughout the entire system's lifetime.

Recent Relation. Recent relations, as their name suggests, are relations which appeared sometime after the first version of the system and remained in the system until the last version. In our cases studies we have observed that the percentage of recent relations is usually greater than of lifetime relations. A special category of recent relations are the *newborn* relations, which appeared in the last version of the system. In a system in which the architects monitor the progress of the system such as the one presented by Rötsche and Krikhaar [21], special attention must be dedicated to newborn relations, as they are candidates of enforcing the architecture guidelines. Of similar interest are also the newborn dependencies (for which the definition parallels the one for newborn relations). Figure 5 c) shows how an inter-module inheritance dependency between `org.argouml.uml` and `org.argouml.application`, which in version 16 was a newborn dependency, is refactored in the subsequent version.

Fossil Relation. The fossil relations are relations which are not present anymore in the system's last version. As

a result, it is likely that they are of little interest for the reverse engineer. However, they can provide information about the history of the system to the system architect because they are a sign of architecture restructuring. In the two case studies we studied, we encountered large differences in the percentage of fossil relations out of all the existing inter-package relations (see Table 1). It would be interesting to study wether the percentage of the fossil relations from all the relations in the system can be useful in characterizing the system's evolution.

Stable Relation A stable relation is one relation for which although there are new *explicit* dependencies of various types appearing and disappearing during the evolution of the relationship, there are no new *implicit* dependencies.

The stable relations are important because, if a stable relation is also a lifetime relation, it is very likely that it represents an architectural foundation of the system, although we can not rule out completely the possibility that the relation represents dead code that was not detected and removed. Information about the development process of the system can bring more light on the importance of the relation: in a system developed using agile methods it is less probable that a relevant dependency remains stable compared to a system developed using a more conservative model the dependency.

Figure 5.a) is an example of a stable relation. From the first analyzed version to the last, the configuration of invocations and inheritances stays the same. However, it changes quantitatively: the reliance of `uml` on `kernel` becomes stronger. From the names of the modules we can infer that they are important for the architecture of the system. Indeed, in our case the stable lifetime relation is important from the architectural point of view.

Instable Relation. It changes both quantitatively and qualitatively during the system's evolution. A qualitative change involves the appearance of new types of inter-module dependencies: during the evolution of the instable relation new *explicit dependencies* are introduced which trigger the apparition of new *implicit dependencies* between the modules. The existence of instable relations in a software system might be a sign of a continuous struggle to find the right design, and are a good starting point to investigate what are the system's problems. Figure 5.c) shows the history of an instable relation. The relation between the modules `org.argouml.uml` and the `org.argouml.application` was initially a mutual dependency. The main reason for the relationship was the reliance of `uml` on the functionality provided by `notation` to generate UML notation. Beginning with version 12, there is a strong surge in the number of invocations to `application`. By inspecting the contents of the strong invocation dependency between `uml` and `application` we learn that the main reason for the surge is the increase in the number of calls to internationalization services which are localized at this moment in the `application` package. Another change in the relationship happens in version 18 when the earlier dependency weakens again: the internationalization functionality was moved to the `i18n` package[3]. This can be corroborated with Figure 5.b) where the dependency between `uml` and `i18n` also experiences a surge. After closer inspection we discovered that a similar surge appeared also between `ui` and `i18n` as a result of adopting system wide internationalization support.

Intermittent Relation. A particular case of instable relation is an *intermittent relation* for which there are versions in which the relation disappears completely as there are no more dependencies between the two modules. Figure 4 shows an intermittent relation which is the result of the complete removal from the system of the persistence package during versions 0-14 and 0-16 of ArgoUML.

5.1. Quantitative Evaluation

One of the problems of many approaches based on visualization is that they are difficult to evaluate in an automated way. In our case, we wanted to find out how relevant are the patterns presented in this section for real-world systems. To do this, we first formally defined the patterns so we were able to encode them to be used by a query engine. Using the query engine, we extracted statistics about the frequency of occurrence of patterns in various software systems.

System	*Azureus*	*ArgoUML*
Versions	5	6
Dependencies	4060	760
Lifetime	1543 (38%)	207 (28%)
Fossil	198 (4%)	257 (33%)
Recent	2319 (58%)	296 (39%)
Stable	1444 (36%)	157 (20%)
Instable	2616 (64%)	603 (80%)
Intermittent	12 (<1%)	28(<1%)

Table 1. Pattern frequency in Azuerus and ArgoUML.

Table 1 displays a quantitative overview of the frequency of occurrence of inter-module relation evolution patterns in two open-source case-studies. The table shows that Azureus

3 Internationalization is often abbreviated as i18n where 18 refers to the number of letters omitted (nternationalizatio)

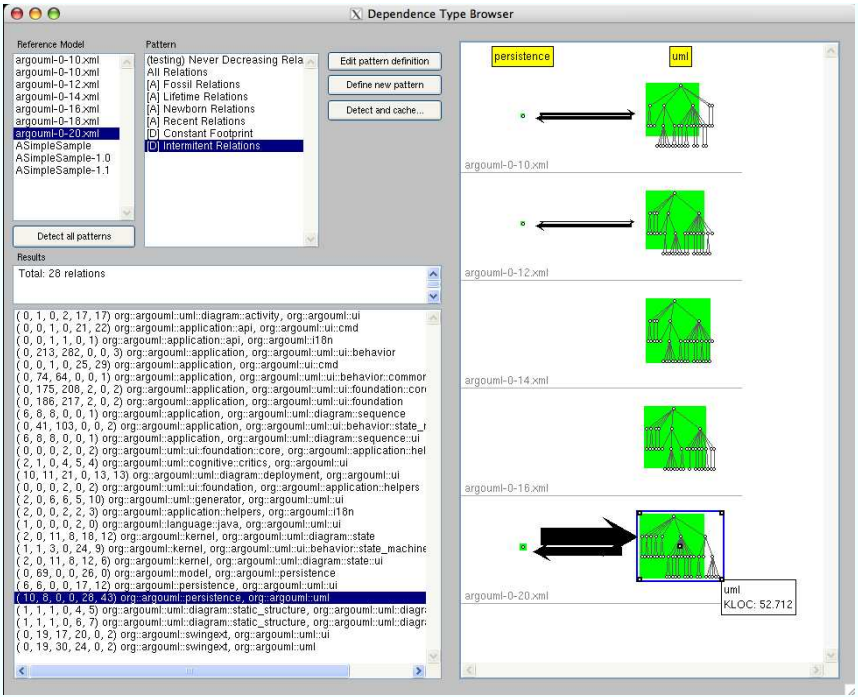

Figure 6. The Edge Evolution Pattern Browser during an ArgoUML analysis session

has undergone a less dramatic evolution as only 4% of the dependencies have disappeared, compared to ArgoUML in which more than 30% have disappeared. Another observation regarding the age-related patterns is that in both systems there are less *lifetime* relations than *recent*— relations, property which makes the multi-version information a criteria for filtering less relevant information during system understanding.

The table also shows that the majority of the relations in a software systems are instable. We can observe that ArgoUML has many more instable relations, denoting a heavily evolving architecture.

6. Discussion

Relationship Visualization. We have shown how the Edge Evolution Filmstrip and the Semantic Dependency Matrix are useful in understanding the relations and dependencies between modules. However, the techniques also have limitations, and one of them is scalability. For dependencies with many clients and providers the matrix can be too big; for analysis which involves many versions, the filmstrip would need scrolling, which would affect its capacity of characterizing a relationship at a glance. However, during our analysis scalability was not a problem.

The Edge Evolution Filmstrip presents another limitation: the module detection is done completely based on the module name, meaning that if a module was renamed in one version we will not detect it in subsequent versions.

Evolution Patterns. The presented relationship evolution patterns can be useful both during the analysis and development of a software system. Several applications are:

Reducing information overload. The edge evolution patterns are integrated in Softwarenaut's filtering mechanism. This means that the user can reduce the number of visible relations depending on the type of activity he is doing. For example, for a first encounter with a system, it is reasonable to limit the display to the lifetime relationships. Table 1 shows that in some cases this can significantly reduce the displayed information.

The Edge Evolution Pattern Browser. We have seen how visualizing the evolution of a relation is useful in understanding the relation. A totally different usage scenario is the one in which the user wants to browse the relationships in the system for special cases, such as problematic relations or relations which conform to specific patterns. Figure 6 presents the Edge Evolution Pattern Browser, a tool that provides the possibility of querying the system for relation patterns. For example, detecting instable dependencies can be a starting point for problem detection.

Automatic Problem Detection. A possible extension of the previous idea is a system which monitors the evolution of a project and automatically detects when dependencies which correspond to certain patterns appear.

Case Studies. The systems that we have analyzed are high-quality open-source systems and it is probable that the results would be slightly different for industrial systems where we expect to encounter more problematic cases.

7. Related Work

There is a number of software exploration tools, such as Bauhaus[11], SHriMP[14] and the seminal Rigi[16]. Rigi features a sophisticated scripting engine which gives it a great deal of flexibility. One impressive experiment with it was performed by Riva[19] who used Rigi to reverse architect an industrial system. Still, Rigi emphasizes the (aggregation of) nodes. SHriMP provides more focus on the dependencies than the usual dependency graphs. It can display only some of the various types of dependencies and offers the possibility of visually aggregating the dependencies between low-level entities to higher-level dependencies. Sotograph[23], a commercial software analysis workbench, which also includes inter-module dependency visualization.

A contribution of our work that distinguishes it from the afore mentioned tools, is the possibility of selecting an individual edge and providing detailed visualizations for it. Another distinct feature is the fact that, based on the dependency patterns, we can define filters which take into account evolutionary information.

An interesting case of applying architecture evolution analysis tools in the reengineering of an large industrial system was presented by Rötschke and Krikhaar [21] who applied trend metrics. The toolset presented by Rötschke and Krikhaar can present details about a dependency between two module in html format.

Holt and Pak [10] were among the first to attempt to visualize the evolution of software systems at the architectural level. They implemented GASE, a tool which highlights the differences between two versions of a system by simple visual means (*e.g.*, color coding the differences between two versions). Their approach was focused towards visualizing the evolution of the dependencies between two versions.

An approach which treats histories of entities as a whole is taken by Gîrba *et al.* [8] who characterize the evolution of whole class hierarchies by combining metrics and visualization. Their work focuses only on the evolution of inheritance relations and proposes a visual representation for summarizing the evolution of the class hierarchies.

Hassan *et al.* [9] annotate dependency graphs with information retrieved from version control repositories.

The idea of representing software relations using incidence matrices was used also by Pauw *et al.* [5] to represent the relations between classes in a software system. Their visualization is similar to ours, but, in their case, the matrix has the same elements on the rows and columns which makes it useful for understanding the relationships between the classes. In our case the elements on the columns represent the client and the ones on the rows represent the provider, so the matrix speaks about the relation between the two.

With respect to the recovering of the architecture of software systems (also known as reverse architecting) the two PhD theses by Pinzger [18] and Riva [20] both addressed the problem in their own way, without however taking into account the semantics of the relationships between the architectural components.

8. Conclusions and Future Work

We started this paper by introducing a distinction between inter-module *dependencies* and inter-module *relations*. Then, we argued for the importance of enriching software exploration tools with the capacity of visualizing and analyzing both the relations and the implicit dependencies between the modules in a software system. As concrete examples of such visualization and analysis techniques we presented the Edge Evolution Filmstrip and the Semantic Dependency Matrix. The *Semantic Dependency Matrix* is a technique for displaying the details about the interaction between two modules by grouping the classes which interact in similar ways together in order to encourage the emergence of interaction patterns. The *Edge Evolution Filmstrip* visualization that we propose presents the evolution of a relationship between two modules through multiple versions. Based on the experience of using the film strip we extracted a set of inter-module dependencies evolution patterns. We showed how our exploration tool, Softwarenaut, makes use of the patterns to recover architectural information useful both for software architects and reverse engineers.

Future Work. We plan to continue our work on the relationship and dependency visualization as we think there are still better techniques to be discovered that would help in system understanding and quality assessment. One such technique could be visualizing the interaction between the natural language terms present in the two analyzed modules. Regarding the dependency evolution patterns, we plan to both look for more systems to analyze and other criteria for relationship classification.

Acknowledgements. We would like to thank Marco D'Ambros and Romain Robbes for comments and suggestions on previous drafts of this paper as well as the anonymous reviewers for their observations.

References

[1] Argouml, UML Modeling Tool. http://argouml.tigris.org/.

[2] Azureus, BitTorrent Client. http://azureus.sourceforge.net/.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[4] A. Cockburn. *Agile Software Development*. Addison Wesley, 2002.

[5] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pages 326–337, Oct. 1993.

[6] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[7] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, Nov. 1997.

[8] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[9] A. Hassan and R. Holt. ADG: Annotated Dependency Graphs for Software Understanding. In *International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE CS, 2003.

[10] R. Holt and J. Pak. GASE: Visualizing software evolution-in-the-large. In *Proceedings of Working Conference on Reverse Engineering (WCRE 1996)*, pages 163–167, Los Alamitos CA, 1996. IEEE Computer Society Press.

[11] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[12] M. Lungu and M. Lanza. Softwarenaut: Cutting edge visualization. In *Proceedings of Softvis 2006 (3rd International ACM Symposium on Software Visualization)*, pages 179–180. ACM Press, 2006.

[13] M. Lungu and M. Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006*, pages 349–350, Los Alamitos CA, 2006. IEEE Press.

[14] C. B. M.-A. D. Storey and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[15] H. A. Müller. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[16] H. A. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86. IEEE Computer Society Press, 1988.

[17] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[18] M. Pinzger. *ArchView – Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology, 2005.

[19] C. Riva. Reverse architecting: an industrial experience report. In *Proceedings WCRE 2000*, pages 42–50. IEEE Computer Society, 2000.

[20] C. Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Technical University of Vienna, 2004.

[21] T. Rötschke and R. Krikhaar. Architecture Analysis Tools to Support Evolution of Large Industrial Systems. In *Proc. IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 182–193, 10 2002.

[22] Softwarenaut, A Software Exploration Prototype. http://www.inf.unisi.ch/phd/lungu/softwarenautl.

[23] Sotograph, a commercial software visualization tool. http://www.software-tomography.com/html/sotograph.html.

[24] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 12–21. IEEE Computer Society, 1997.

A. Case Studies

The two case-studies in this paper are ArgoUML[1], a UML modeling environment, and Azureus[2], a BitTorrent client. In our analysis we considered only major versions of the systems. Because ArgoUML uses an odd/even release schedule with 0.even releases as stable releases and 0.odd releases as developer releases, we have chosen all the even releases between versions 0.10 and 0.20. For Azureus we considered the major releases between 2.1.0 and 2.5.0. Figure 7 shows the evolution of the number of classes in the two analyzed systems.

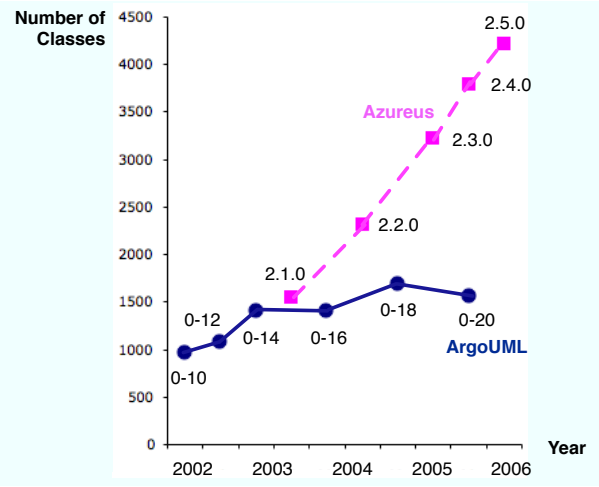

Figure 7. Size Evolution in the Case Studies