A Patterns based reverse engineering approach for Java source code

Rui Couto, António Nestor Ribeiro and José Creissac Campos
Departamento de Informática/Universidade do Minho & HASLab/INESC TEC
Campus de Gualtar, 4715-057 Braga, Portugal
Email: {rui.couto, anr, jose.campos}@di.uminho.pt

***Abstract*—The ever increasing number of platforms and languages available to software developers means that the software industry is reaching high levels of complexity. Model Driven Architecture (MDA) presents a solution to the problem of improving software development processes in this changing and complex environment. MDA driven development is based on models definition and transformation. Design patterns provide a means to reuse proven solutions during development. Identifying design patterns in the models of a MDA approach helps their understanding, but also the identification of good practices during analysis. However, when analyzing or maintaining code that has not been developed according to MDA principles, or that has been changed independently from the models, the need arises to reverse engineer the models from the code prior to patterns' identification. The approach presented herein consists in transforming source code into models, and infer design patterns from these models. Erich Gamma's cataloged patterns provide us a starting point for the pattern inference process. MapIt, the tool which implements these functionalities is described.**

I. Introduction

Paying more attention to the modeling phase during the software development process has shown multiple benefits. However, to achieve such benefits there are some prerequisites. A necessary requisite, to allow correct system implementations, is to have the models as complete and correct as possible. This results in time consuming modeling tasks and usually, once written, a model is never updated again. This lack of attention to the models makes them obsolete and therefore useless [11]. So, the full advantages of the modeling process are not attained. To overcome this issue, the Object Management Group (OMG) proposed a new software development methodology, the Model Driven Architecture (MDA) [10], [12]. This methodology is based on the definition of models, and their transformation into new models or source code.

Usually, on traditional approaches to software development, the time spent writing models is considered wasted. This happens because only code writing is considered software production. The MDA turns the time "spent" writing models into software production, by allowing code to be generated from those models. It defines the standard way to develop model based software solutions. The main objective of the approach, is to raise the relevance of the modeling phase, and even, to make the modeling process the only needed work to create software solutions.

The growth of the number of languages, tools and platforms (for example, for user interfaces development) is leading the software industry to high levels of complexity. This is mainly caused by the tools' lack of capability to hide this complexity. Raising the abstraction level has always been the solution to the problem of dealing with complexity. Hence, MDA appears as a solution to this issue in the broader area of software engineering [10], [12]. Two specific types of models are particularly relevant in this context: Platform Independent Models (PIM) and Platform Specific Models (PSM). PIM models are expressed at a level of abstraction that makes them independent from any concrete deployment technology or computational platform (e.g. J2EE or .NET). PSM models are another kind of software model, with a lower abstraction level than a PIM, closer to the final software. PSM models are typically derived from PIM models and represent an instantiation of the latter for a specific platform and technology [15].

In a typical MDA driven project PIM models will be evolved into PSM models and then into source code. However, for existent projects, or when projects are updated directly in the source code, tools are needed that are able to generate models based on their source code. The OMG, however, does not define the reverse process from code to models. It only defines the models to code process. In order to address this limitation, we have started studying a reversed MDA process. Hence, the MDA process on the reverse form will be this paper's focus. With it, it is possible to easily create, evolve and migrate software on a model based approach setting.

Having the reversed models, other kind of operations may be performed on them. Of particular interest is the ability to support the understanding of the proposed solution.

Christopher Alexander described a design pattern as the core of a solution for a given recurrent problem. Thus, they may be used to solve a problem multiple times, without repetition [5]. Although the original definition was relative to buildings, it is also considered to be valid for software engineering. A design pattern describes a simple and elegant solution to a well known problem. Their importance having been proved, it is relevant to use them for software development.

Having reverse reengineered the models, an approach to infer patterns was developed. We have started with Erich Gamma patterns [5] as we wanted to develop an approach that could be as generic as possible, but other collections may be adapted later (for example, [1], [4]). Hence, this work's main objectives may be summarised as follows:

1) Generate high level models (cf. PSM) in the Unified Modeling Language (UML) from Java source code;
2) Infer design patterns on those models;
3) Abstract PSM into PIM.

As result, a tool (MapIt) which implements these functionalities was developed.

The remaining of the paper is organized as follows. Section 2 presents some of the work already done on this area. Section 3 presents the expected challenges. In section 4 the achieved tool will be presented. The achieved tool implementation and a case study will be discussed on the section 5. Finally, section 6 concludes the paper.

II. State of the art

Models' definition and transformation are the essence of the MDA process. To have these models, we need a standard way to define them. This is achieved through a metamodel. The metamodel is a "well defined language" which allows us to create both PIM and PSM models. We are particularly interested in PIM and PSM because of their distinct abstraction levels. The standard modeling language adopted to create these models is the UML, but that is not a restriction. Having these models, the transformations are the next step. Then, having transformation definitions, it is possible to transform one model into another at a different abstraction level. By reversing these rules, we may expect reverse transformations.

A MDA transformation tool is what transforms a PIM model into one (or more) PSM models. From that PSM, and using a tool (either the same, or another) it should be possible to create compilable source code. These model transformations are MDA's essence, and tools the technology which implements them.

A transformation definition is a set of transformation rules. The information about how a model element (a PIM element, for instance) should be represented in another model (a PSM, for instance) consists in a transformation rule. In the direct way, PIM to PSM as defined by the MDA, these rules intent to lower the abstraction level. These rules may have reverse representations, which allow reverse transformations and raise a model's abstraction level.

A. Reversing the MDA process

Several MDA implementations have already been proposed by some authors, resulting in tools and algorithms to go from models to code [10], [7]. As already mentioned, reversing the MDA process is very useful, not only to include existent software in the MDA process, but also to keep consistency between source code and models. However, the OMG does not define the reverse process from code to models. The reverse MDA process has already been object of study, but usually in a simple way. For example, via the integration of an existent system into another under development by considering the first as a "black box" which exposes some of its particularities to the outside. A complete integration of the two systems would be more useful, and some authors have also studied this issue.

To start the reverse MDA process there are two possible approaches. The first one consists in a static analysis which mainly extracts the structural aspects, producing PSM and PIM models. The second approach analyses software behavioral aspects, which provide another kind of information about the software comportment.

The most common methodologies suggest the following approach: first, the source code must be analyzed (be it text or bytecode); second, relevant information must be extracted to create an intermediary representation (for example, a syntactic tree or a graph) [16]; finally, as these representations contain a high level of detail, they must be simplified [13], [3]. This approach allows not only the integration of existing software into the MDA process, but also the execution of high level operations over the models. For example, carrying out pattern inference over the models to better understand the rational behind the implementation.

The tool presented here is based in the static approach just described. Performing the basic structural analysis was a relatively simple task, because it relies only in the (textual) source code. On the other hand, inferring the relation between the model elements was the hardest task (apart from the dynamic information). Some studies stated that while binary associations can be directly extracted from source code, $n - ary$ associations require more work to infer. In the presented tool, only the "zero or more" $(0..*)$ multiplicity was considered. This multiplicity has shown to be sufficient to achieve the proposed objectives. The static analysis process allows the creation of (possible incomplete) UML class diagrams. To understand this process' difficulty, one of the most precise recovering tool (according with the author), is capable of extracting only 62% of UML elements [6].

There were two possible starting points for the analysis process. One was to start from the textual source code, as programmed by the developer. The other was to start from the Java bytecode. The chosen approach was the first one, the second being considered less suitable for the tool's proposed objectives. Indeed, the textual approach has shown to be more adequate in this context. First, the source represents the system without compiler optimizations. Second, if the developers resort to obfuscation techniques, some details may be lost. Also, by using the textual source code, it is possible to integrate this tool in the development process, providing the functionalities while creating code.

B. Pattern Inference

The importance of design patterns is demonstrated by the number of patterns found on software developed nowadays. Some of the advantages their use offers include the fact that they allow a higher abstraction level vision of the software, and that they can be used as quality (or lack of quality) measures. When patterns are not documented in a software, an inference process is needed to retrieve them [6].

A pattern inference functionality offers multiple advantages. By offering a higher level view of a system's implementation, it makes it easier to understand it. It provides measures of

the quality of an undocumented implementation, support for defect detection, and it is useful on the project's maintenance phase. The second component proposed for our tool is the pattern inference capability. After reversing source code into PSM and PIM, the tool should be able to infer patterns, based on these models.

Existing studies on pattern inference provide some guidelines about how to implement this functionality [10], [12], [7]. The suggested approach is the following: a given software should be analyzed and mapped onto an adequate, and previously defined, metamodel. This metamodel should contain static and structural information, along with some dynamic information (such as method invocation). From this representation, a knowledge base of facts and rules representing the system information will be extracted into an external format. Then, these facts will be analyzed, searching for design patterns. A set of rules representing design patterns should be defined beforehand, to allow pattern searching. Basing the pattern inference process in a previously obtained knowledge fact is not a new approach. For instance, a similar technique was presented as viable by Roel Wuyts [17].

The intermediary data representation used differs from approach to approach. These intermediary representations may be organized according to the notation used to express them: graphs which preserve the elements' hierarchy, matrices, markup languages, or programming languages (such as Prolog). Having this intermediary representation defined, it is then possible to start the pattern inference process. This process consists in the comparison of the intermediary representation, against previously defined patterns, in the same language.

C. Available tools

We analyzed some of the tools in the MDA context, and we present some of the most relevant conclusions. First, these tools may be categorized in "round-trip" and "reverse engineering", depending on their objective. Some offer Integrated Development Environment (IDE) capabilities, allowing some degree of code and model manipulation to develop software (such as ArgoUML and Fujaba [8]). Other tools are focused on a more efficient analysis, offering higher level operations on the models (like Ptidej [6]). Still other tools combine both functionalities (for instance Together and Visual Paradigm). The Ptidej tool claims to be the most precise tool, being able to recover the highest level of UML elements [6]. Regarding pattern identification, the most relevant tools are Reclipse [16] and Ptidej. Both are able to infer patterns on UML models. Only Fujaba and Ptidej are able to perform higher level operations on the models.

Despite these tools' utility, none of them were able to combine, in one development environment, the three proposed functionalities: to obtain PSM models from source code, to infer patterns in those PSM models and to abstract PSM models into PIM models. The tools closer to that objective are Ptidej, Fujaba and jGrasp. Being these tools an "incomplete" solution, the proposed tool tends to be the integration of the spare functionalities in a single environment.

III. Transformations

The topics discussed above can be reduced to the three main functionalities that the tool should implement (see Figure 1). They are resumed as: the reverse MDA process from source code into PSM, the design patterns inference on a PSM, and the reverse MDA process from PSM to PIM. These approaches are meant to be used in separate accordingly to the needs of the developer.

A. Source code to PSM

The first functionality corresponds to the source code to class diagram (PSM) abstraction, represented in Figure 1 on the left. The process to do such should start with the analysis of the project's source code. This requires a project's structural analysis, analyzing then each of the Java files. To achieve such goal, a semantic analysis is used on the source code, by using a Java parser. The extracted information must be mapped onto an intermediary representation. An adequate way to represent it is by means of a metamodel. This metamodel must be complete, accurate and at the same time not overloaded with useless information. But, even if not all of the software information is needed for the proposed objectives, the metamodel should be ready for future functionalities.

Another tool requirement is a visual representation for this metamodel. The metamodel elements are mapped onto UML elements: for each metamodel element, there will be a visual UML element, shown to the user. It is also required for this diagram to be interactive.

In conclusion, a relation between the metamodel elements, the UML entities and a visual representation needs to be specified. This allow us to define how each Java element will be represented and displayed to the user.

B. PSM pattern inference

Pattern inference on models allows us to extract a higher level information representation of the original model. This process, represented in Figure 1 on the top right, occurs over previously reversed information. That information may also be used for other purposes, in this case, to generate the PIM (see below). Not only the inference process needs to be specified, but also how to customize it. It is required that this process be parameterizable and easily extensible.

The pattern inference process starts with the knowledge base creation. This process depends on two major factors. First, it depends in correctly defining the patterns used in the inference process. Second, it depends in achieving an appropriate knowledge base, representing correctly all the system information. Creating the knowledge base must be a careful step, since overloading the knowledge base with useless information will slow down the inference process.

It is easy to understand that the intermediary representation, the knowledge base, and the patterns are somehow associated. Such relation is the possibility to achieve transformations between them, on the presented order. To start, a knowledge base, based on the source code is created. To handle such

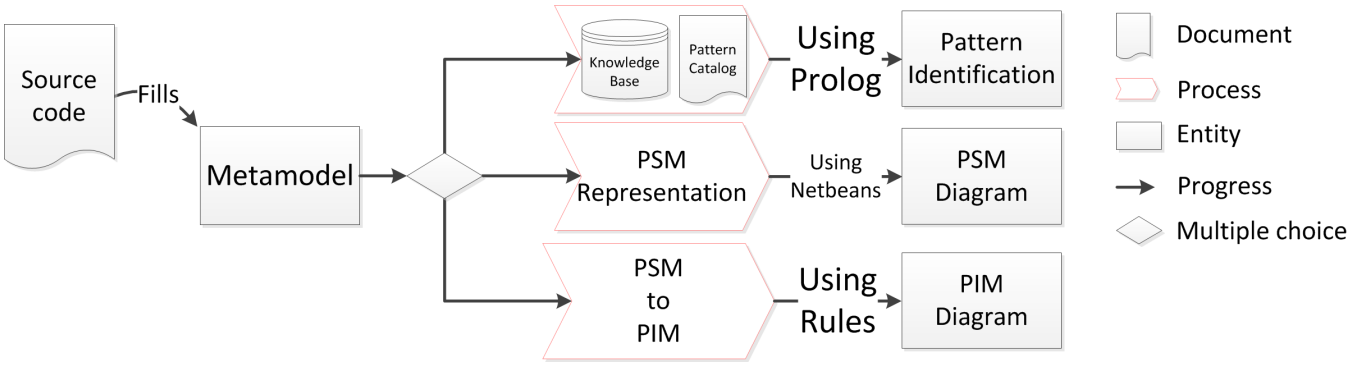

Fig. 1. Representation of the tool functionalities.

representation, the use of an external technology will be considered. So, a tool which implements information mapping and exchanging with that external tool needs to be implemented.

We may only expect to achieve good results with this process, if we have a standard and well defined way to represent a design pattern. Also, it should be possible to interpret an external pattern definition, and use it on the analysis process, to make this process parametrizable. These representations should then be used on the external tool, which should contain the knowledge base. This external definition is called the "pattern catalog". It consists in a set of user defined design patterns. With this approach, a customized set of rules may be created and used to identify those patterns on a model. The implementation of the catalog is achieved by means of an external file, defined by the user.

It would be easier to understand the visual representation of the patterns, if the diagrams follows the same representation as the PSM. As such, it would be interesting, for instance, to highlight the inferred patterns in that model.

C. PSM to PIM

The model abstraction process is usually related with information simplification or reduction. So, this process mainly consists in reducing the elements information, quantity, or even change their information. As result, a PSM is transformed into a more generic model, the PIM. This is represented in Figure 1, on the bottom right.

Regarding the models, the abstraction process can be treated as model filtering or as a model transformation. This process produces also a model, and as such, a visual representation is needed. Since we consider this process as a transformation process, the possibility for future improvements should be taken into account. This process will be based on the MDA Explained book [7] approach. That book presents a set of transformation rules, in the direct way. Their reverse form will allow a fully automated reverse transformation. For each metamodel element, a transformation rule will be defined, such that the rule will transform that element considering the whole application context.

IV. The MapIt tool

We have completely defined the mapping process needed to achieve source code into PSM transformations. To start, a parser analyzes the source code, and preserves the extracted information in a metamodel. This metamodel has a representation for each Java element, as well as the information needed to preserve their hierarchy.

The concept of Java element was created to represent Class and Interface abstractions. The used metamodel contains a mapping for each Java attribute and method which is part of a Java element. Java Classes and Interfaces have a respective mapping as well. All these elements are assembled in a Java package representation.

Inferring the cardinality of relationships is widely recognized as difficulty. Hence, for simplification purposes, only two different relationship cardinalities were considered. Association will then be considered as "one-to-one" relations (1..1). Composition/aggregation as "one-to-many" relations (1..$*$). Additionally, method invocation information, from which class and to which class (or interface) invocations are made, was considered as relevant and included in the metamodel.

As the result of the static analysis, a set of Prolog facts are produced creating the knowledge base. The knowledge base is achieved by parsing all the elements and filtering them. These facts will be used on the pattern inference process.

We may also achieve incomplete patterns inference, by resorting to Prolog anonymous variables. To do such, for a given pattern, each of its arguments may be replaced by an anonymous variable, in its representation. Making all the permutations among all the pattern elements, all possible incomplete patterns will be found.

A. PSM to PIM abstraction

The PIM consists in a model which has no relation with the target platform details. However, the PIM models obtained with this tool are not true PIM, since they may contain some (Java) platform details. However, those models' objective remains the same — to raise the abstraction level.

To achieve PIM from PSM, a set of transformation rules is needed. Only by having both PIM and PSM completely specified can transformations from PIM to PSM be defined.

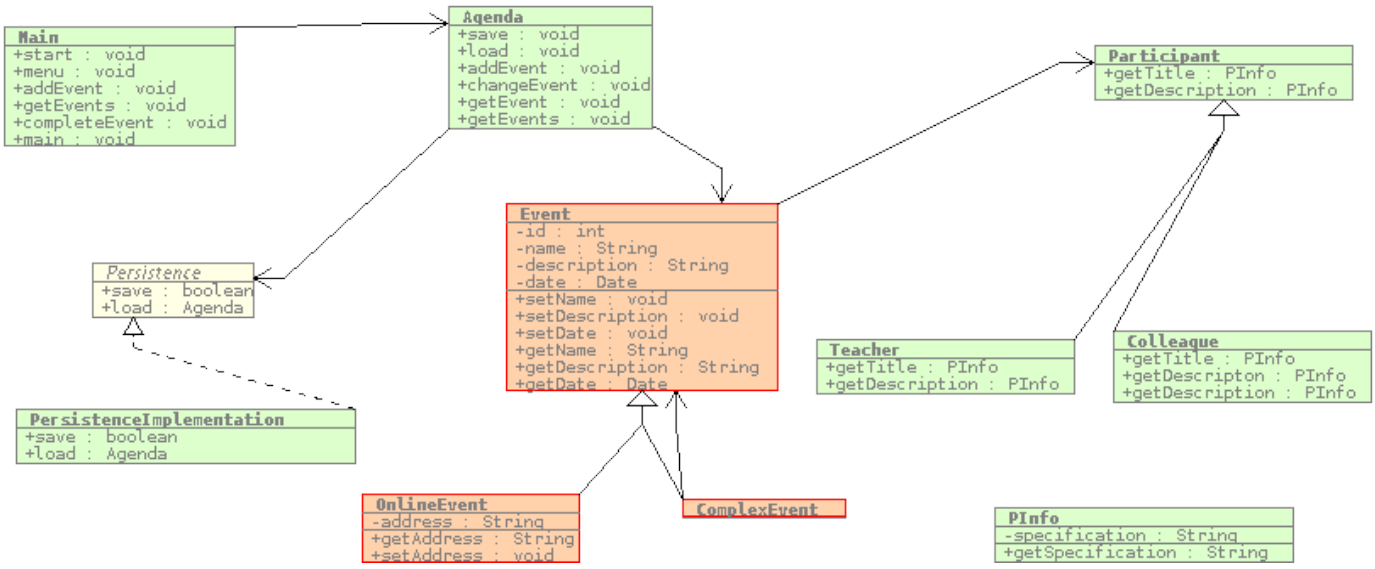

Fig. 2. Example of identified pattern in a PSM diagram.

To achieve this, they are both modeled in UML. In short, it is possible to say that PIM models derive from PSM models by removing the Java platform specific elements from the PSM.

As described in MDA Explained [7], the models transformations are achieved by applying well defined rules on model elements. These rules describe how elements from one model are mapped into another model, at a different abstraction level. That book presents a set of rules to achieve PIM into PSM transformations. A suggested approach is to define the book rules on the reverse form. The transformation process is then defined as the process of removing or filtering elements for a given model. All of the PSM elements will be processed and transformed into higher level elements, resulting in a PIM model

The used transformation rules are encoded in Java methods, and for each element, it should exist a different rule. Each method (for each PSM element) receives as argument not only the information about the element being processed, but also the information about all their hierarchical superior elements. This way, we may achieve model-wide transformations.

B. PSM pattern inference

In the pattern inference process, the PSM is analyzed, and the relevant structural information is extracted. This information is preserved in an internal representation, the basis for the pattern inference process. The representation is handled by a Prolog interpreter, which creates a knowledge base from the information therein.

Here, the suggestion is to include the Prolog technology to handle these representations. Thus, the Prolog knowledge base will keep all the software facts, extracted from an intermediary PSM representation. There are a set of facts that are generated from the software representation, described as Prolog facts. A prolog fact is described as `name/arity`, where arity means the number of arguments of the fact. A class existence information will be expressed in a `class/1` fact. For an interface, the same approach is taken with `interface/1`. Aggregation, composition or association information is represented as `contains/2`, between two classes or a class and an interface. Heritage relationships are represented with the `extends/2` fact. Implementation properties are also considered, as `implements/2`. Invocation information is represented with the fact `calls/3`, between two classes (or a class and an interface) and one method.

Once the Prolog knowledge base is populated, it can be questioned for patterns. As soon as the pattern catalog is parsed, this module will interact with the Prolog tool to search matching patterns with the provided catalog rules. It is possible to conclude that this might be considered a property satisfaction problem, for a selected rule, on a given knowledge base.

Figure 2 illustrates the identification of a pattern in a PSM. In this figure (generated by the MapIt tool) the composite pattern was inferred, where `OnlineEvent` and `ComplexEvent` extend `Event`. `ComplexEvent` contains one-or-more `Events`, creating then the composite pattern among these classes.

The rules' precision level will define the results' quality. A lenient rule will find more patterns with a lower precision level. A more strict rule will find less patterns, but with more precision.

C. Implementation

The parsing process is composed by two phases. First, the information is extracted as represented from the source code, and converted in metamodel elements. Only then it is possible to establish the relations between the elements. The information extracted from the parser is then mapped into an adequate metamodel, since metamodels are recognized to be the best information representation for pattern inference [6].

The Prolog integration uses the GNUProlog interpreter and engine to handle the knowledge base. The two previously presented functionalities are abstracted by a module which handles interaction with the interpreter, and loads also the pattern catalog.

The presented pattern catalog is a textual file, containing a set of Prolog rules (in a specific format). Each line of the file should define a rule using the following format: `pattern_name/arity#(prolog_rule)`, were `pattern_name` is the design pattern's name, `arity` is number of arguments of the rule, and `prolog_rule` is the Prolog rule, representing the design patten. Figure 3 depicts an example of a Prolog rule (for the Composite pattern).

The final user interface, enabling access to the functionalities presented above, is provided by a NetBeans plugin. With this approach, a development process supported by the MapIt tool is done in the same environment already used by developers. This way, the user may take advantage from the IDE functionalities while using the tool. An effort was made to make the Prolog module as generic and reusable as possible by all the functionalities, so they may be improved or changed with less effort. This was mainly achieved due to the metamodel capability to express both PIM and PSM, so only one metamodel representation was made.

The last implemented functionality was the models transformation automation. This required a set of transformation rules to be fully specified beforehand (these rule may be easily extended). These rules were implemented as follows. A Java class is mapped onto a PIM class. Also, a Java association is mapped on a PIM association. For a Java public attribute, a PIM public attribute is created. A Java method is transformed onto a PIM operation, in an adequate way. A private Java attribute which contains the correspondent getter and setter, is transformed into a public PIM attribute. Also, for each element, properties such data type should be removed or adjusted. Absent PIM elements such as Java interfaces or other project files (Extensible Markup Language (XML) files, for instance) will not exist in the PIM.

By following the approach detailed above, we were able to achieve a fully working tool. In several tests, it was possible to successfully generate models for multiple kinds of software. On some of those models, we were also able to infer design patterns (some documented patterns, other developed by us). Finally, we successfully raised the abstraction level of the analyzed systems, with the achieved PSM.

V. Discussion

We have used two distinct softwares in order to test the achieved tool. A smaller software was used for a more detailed analysis, and a bigger software to test a more complex case. The smaller software consists in an agenda which allows events and participants management. It has some inferable design patterns and at the same time does not have a large set of classes. With this software it was possible to make more accurate tests. The more complex software was JHotDraw. Apart from this software functionalities, there are some facts to justify its choice. First, it is an open-source software, available to the general public. Second, this software has an considerable size, with about 160 classes and 9000 lines of code. This makes it a good study case. Also, this software has the property of being developed by a team which includes Erich Gamma. This results in a design pattern based software. Hence, this software allowed us to test the tool behavior in a more complex environment.

Both source code to PSM and PSM to PIM functionalities produced the expected results. They generated diagrams containing the Java elements and their relations. Also, both functionalities are fully automated as proposed, with no software size restrictions, working in the same way for both projects. The code to PSM functionality produced an UML class diagram as expected. All the predicted UML elements and their relations were represented, based on the metamodel instance. Also, this process does not require user interaction (apart from selecting the desire functionality), as proposed.

As expected, the JHotDraw software analysis resulted in a larger number of elements shown in the PSM diagram. As the representation is interactive, it allows adjusting the elements position. This functionality eases the analysis of larger softwares diagrams, meaning there are no restrictions for to the maximum size of the analyzed software. Another functionality developed to facilitate the analysis process allows to simplify the diagrams, by representing the metamodels elements only by its name. The PSM to PIM module produced similar results, producing a higher lever UML class diagram. This model is obtained by applying previous defined rules to the PSM elements. Here again, the software dimension is not a restriction to this use this functionality, and the produced results are similar.

When using the pattern inference functionality, the user must select if he/she wants to use the embedded pattern catalog or, otherwise, select an external pattern. Once again, this functionality is software size independent. All the inferred patterns are identified and listed to the user. The higher the number of patterns in a software, the more useful this functionality becomes. Some of the analyzed tools presented all the patterns at once. That makes it hard to understand the patterns arrangement. The achieved results proved the viability to include the pattern inference functionality on a model analysis tool. Also, this shows how it is possible to include an external technology (Prolog) in this process, taking advantage from the language's capabilities.

All the functionalities were implemented in a NetBeans plugin. An effort was made to integrate the tool in the IDE, allowing it to be used during the development process. When comparing the resulting tool, against the proposed objectives, we can make some considerations. Firstly, the proposed functional objectives were achieved overall, and the viability of the approach is considered as proven. Then, other non-functional objectives such as usability improvement (against other tools), or facilitating the tool's installation and usage were also addressed by resorting to Netbeans. This IDE is a widely known tool, so installing and using the plugin will be

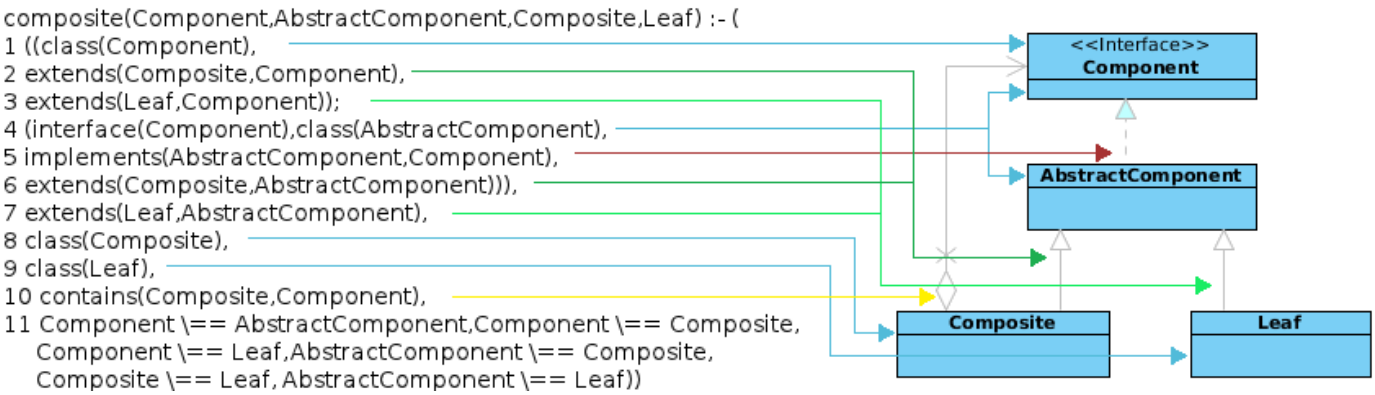

Fig. 3. Prolog rule representation for the Composite pattern.

familiar to developers.

As presented here, integrating the proposed functionalities into one single tool was overall an achieved objective. We were able to successfully create a tool, satisfying the presented requirements. We have made an effort to make the Prolog module reusable in other contexts, since it offers an interaction with Prolog, by allowing to assert facts and questions. Also, we tried to provide an efficient way to easily extend both the Prolog and the parsing module for changes in further work.

When comparing the obtained results against the other tools' results, some considerations can be made. Comparing the achieved models' details, the other tools' models were generally less detailed. Even if all the tools (apart from Reclipse/Fujaba) were able to recognize all the Java elements (classes and interfaces), many of them were not able to correctly recognize their relations (specifically ArgoUML, jGrasp and Ptidej). The Reclipse/Fujaba tool was able only to represent the inheritance relationship, and the Ptidej tool missed some relations. Once again it was possible to conclude that collection inference is hard to achieve, since all these tools showed difficulties when performing this task.

Regarding relations between elements in the models, ArgoUML and Reclipse were not able to recognize typed collections. In the presented tool, none of these problems is present, so the results concerning models' elements and their relations are considered satisfactory.

The models elements' quality was also compared in the obtained models. Only two tools achieved satisfactory results, producing models' elements in UML notation, being them Reclipse/Fujaba and Ptidej. It was possible to conclude that Ptidej achieved the best results. Even if Fujaba/Reclipse represented detailed diagrams, some information (about relations) was missing. Also, even if the Ptidej tool achieved the best results, the produced diagrams are static (making it impossible to rearrange the elements on the screen). Only a few analyzed tools were able to infer patterns in the diagrams, specifically Reclipse/Fujaba and Ptidej.

The pattern inference (and representation) process occurred over the obtained models. Tools supporting this step also allowed the use of external catalogs to define the patterns to infer. However, these tools used hard coded representations, such as, for instance, representations in Java code. In the tool presented herein a more flexible format is adopted (by using Prolog rules), making it simpler to extend the patterns catalogue. Since the Model and Pattern Inferring Tool (MapIt) tool used a similar approach in the pattern inference (apart from the issue of patterns representation), similar results were achieved.

None of the analyzed tools have the model abstraction functionality, so it is not possible to compare it. However, regarding the examples presented in the MDA Explained book, it is possible to conclude that the obtained models have a higher abstraction level, close to a PIM (as expected).

VI. Future work

We are now able to generate UML models, as well as identify architectural patterns on those models. However, there is room for further work both in terms of the development of the tool, and in terms of its use and the type of analysis it might support.

Regarding the tool, a number of improvement should be addressed. The pattern inference module may be improved expanding the pattern catalog, simply by adding new rules. The fact that the pattern collection is represented as a Prolog knowledge base means, however, that we are not tied to any specific collection of patterns. The model abstraction module can handle different implementations for the transformation process, requiring these transformations to be written at the code level. Changing the language supported by the tool may have one of two consequences: changing the parsing module, or, changing the parsing module and the metamodel (if the new language does not match with the metamodel).

Regarding the information obtained with the tool, we can consider, for instance, the possibility to perform qualitative measures with that information. We can also consider what type of systems might be analyzed with this approach.

One particular area that interests us is that of the analysis and reengineering of user interfaces. UML, while a *de facto* standard in object oriented modelling, is not especially targeted at user interfaces. There are a number of approaches that

address this (see [14] for an overview), but we would like to consider a different approach and study the possibility of applying a transformation on the UML model to generate a new type of model on a user interface specific modeling language (e.g. UsiXML [9]). This way, we would avoid using specific UML profiles, and would use instead the more appropriate language for each task.

A similar approach to the reverse engineering performed here was used in the GUIsurfer tool [2]. GUIsurfer is capable of reverse engineering the user interface layer of JavaSwing applications. However, from the experience of developing and using the tool, a number of issues were identified. The tool uses an *ad hoc* language to represent extracted models. Ideally it should use well known modeling approaches and languages to ease, not only communicating the models to third parties, but also the use of models by third party tools. The models cover the user interface layer of the system only. This is a limitation in that in many cases the logic of the interface is governed, or at least influenced, by the business logic. In this context, the use of architectural design patterns is considered as a promising approach, and one that can help bridge the gap between the user interface and business logic layers.

VII. Conclusion

In this paper the reverse MDA process and a pattern inference process were approached. Three distinct functionalities to support these processes were analyzed, detailed, and implemented on the presented tool. The first functionality was the code based (UML) diagram generation. The second was the inference of design pattern in the generated models. The third and last functionality was the PSM into PIM model abstraction.

The presented tool's major purpose is to help in two distinct scenarios. The first one is to help in the maintenance of legacy systems, by easing software analysis. The second purpose is to help model oriented software migration, by integrating software in the MDA process, always resorting to high level data.

During this work, some tools were analyzed and it was possible to conclude that some of them tried to implement the presented functionalities. As described, available tools presented several shortcomings, where our tool is able to produce adequate results. They fail (as described) on may points, where the MapIt tool is able to succeed. The use of Prolog (and the catalog customization) to help the pattern inference process, in particular, improved the achieved results.

Support for other languages (such as c#, c++, etc) is left as future work. Also, extending the pattern catalog (by using other pattern catalogs) is suggested. Migrating the plugin to other IDEs (such as Eclipse) will allow more users to have access to it, and is considered as future work as well. The integration of the Prolog inference engine on the plugin should also be considered. The Prolog knowledge base provides a source of qualitative data. Now that we are able to achieve such representations of the systems, we must analyze what kind of operations are we able to perform over such data.

The work reported can be seen as a first approach to pattern-based reengineering of software systems taking an holistic approach to code. We have focused on well know patterns from software engineering, as that enabled us to develop the technology. At this stage a fully working tool was achieved. We aim to extend this approach to other areas such as user interfaces analysis and migration, among other possibilities

Acknowledgments

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-015095.

References

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[2] J.C. Campos, J. Saraiva, C. Silva, and J.C. Silva. GUIsurfer: A Reverse Engineering Framework for User Interface Software. In A.C. Telea, editor, *Reverse Engineering - Recent Advances and Applications*, chapter 2, pages 31–54. InTech, 2012.

[3] James Corbett, Matthew Dwyer, John Hatcliff, Shwan Laubach, Corina Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on Software engineering*, pages 439–448. ACM, 2000.

[4] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Yann-Gaël Guéhéneuc. *Un cadre pour la tra caabilité des motifs de conception*. PhD thesis, Université de Nantes, 2003.

[7] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[8] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 22–. IEEE, 2002.

[9] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Vctor Lpez-Jaquero. UsiXML: a Language Supporting Multi-Path Development of User Interfaces. In *Engineering Human-Computer Interaction and Interactive Systems*, volume 3425 of *Lecture Notes in Computer Science*, pages 134–135. Springer-Verlag, 2005.

[10] Stephen Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[11] Naouel Moha and Yann-Gaël Guéhéneuc. PTIDEJ and Décor: identification of design patterns and design defects. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 868–869, New York, NY, USA, 2007. ACM.

[12] Oscar Pastor and Juan Carlos Molina. *Model-Driven Architecture in Practice*. Springer-Verlag, 2007.

[13] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. University of Tampere, 2000.

[14] Hallvard Trætteberg. Model-based User Interface Design. Phd thesis, Norwegian University of Science and Technology, May 2002.

[15] Frank Truyen. *The Fast Guide to Model Driven Architecture - The Basics of Model Driven Architecture*. Object Management Group, 2006.

[16] Markus von Detten, Matthias Meyer, and Dietrich Travkin. Reverse engineering with the reclipse tool suite. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 299–300. ACM, 2010.

[17] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In *In Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE, 1998.