A Formal Foundation for Object-Oriented Software Evolution

Tom Mens*
Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, BELGIUM
Tom.Mens@vub.ac.be

Abstract

My PhD thesis [7] claims that the principles behind object-oriented software evolution are independent of a particular domain or phase in the software life-cycle. To validate this claim, a formalism based on graphs and graph rewriting was developed and applied to a particular aspect of software evolution, namely the problem of software upgrading and software merging. When the same piece of software is modified in parallel by different software developers, unexpected inconsistencies can arise. Formal support can be provided to detect and resolve these inconsistencies in a general way.

1. Thesis statement

In recent years, a lot of effort has been put in trying to make object-oriented software more *reusable*. Examples are the acceptance of application frameworks, the introduction of component-based development and aspect-oriented programming, the widespread use of all kinds of patterns, and many more. Nevertheless, in order to create adequate reusable software, *evolution* is crucial, because good reuse can only be achieved after several iterations over the software. It is inconceivable to predict all possible uses of a reusable component upon its conception. Moreover, in order to prevent software aging, the software must continue to evolve to adapt to the ever-changing software requirements.

Unfortunately, there are still many difficulties related to software evolution. Problems with version proliferation, change propagation, software aging, software upgrading and software merging are frequently cited in the literature. To cope with the latter two problems, the *reuse contracts* technique has been introduced [13]. Reuse contracts have been applied to evolving object-oriented class hierarchies [1], collaborating classes [6], UML interaction diagrams [9,10], software architectures [11] and even evolving software requirements [2].

Although these results indicate that reuse contracts are general enough to provide support for software upgrading and software merging in all phases of the software life-cycle, from requirements specification to implementation, this claim still needs to be validated. Until now, each time reuse contracts were applied to a different domain, the following questions needed to be readdressed:

- What do the software entities in the considered domain look like?
- How can simple software entities be composed into more complex ones?
- How can software entities be modified and reused?
- What are the possible relationships between software entities?
- What are the potential conflicts or inconsistencies in related software entities when one of them evolves?

While the answers to these questions often are partly specific to the considered domain, we have observed many similarities between all the different domains. By defining a *formal foundation* of reuse contracts, we can capture these similarities, and illustrate that the principles behind software evolution can be expressed in a domain-independent way. More precisely, we can define a general formalism for detecting (and resolving) evolution conflicts. For each considered domain, the formalism can be instantiated by providing domain-specific evolution operations and well-formedness constraints. This significantly reduces the amount of work needed to

* Postdoctoral Fellow of the Fund for Scientific Research – Flanders (Belgium)

provide support for software upgrading and software merging.

The formalism can also help to address the *scalability* issue of reuse contracts. Reuse contracts can be taken to a higher level of abstraction by providing appropriate abstraction mechanisms (such as nesting and encapsulation), and by allowing *arbitrarily complex reuse contracts* instead of only primitive ones.

This led us to the following thesis statement [7]:

> A formal foundation of reuse contracts allows us to deal with (object-oriented) software evolution in a domain-independent and scalable way.

2. Scope

Since this thesis statement was too ambitious to be proven in general, we made the following research restrictions:

- Only consider the problem of software upgrading and software merging
- Use category theory as an underlying formalism
- Use graphs to represent software entities
- Use conditional graph rewriting to represent evolution of software
- Express reuse contracts in terms of graphs and graph rewriting

Because we restricted our scope to software merging and software upgrading, we selected a technique that had already proven its use in this context, namely reuse contracts. The reason for this selection was based on a number of criteria: *(a) familiarity*: the approach was developed in our lab, and the original researchers were available for feedback; *(b) applicability*: the approach has been shown to be applicable to a variety of domains, ranging from implementation level over design level to requirements level; *(c) simplicity:* it is a lightweight approach based on some very intuitive ideas; *(d) relevance:* despite its simplicity, the approach addresses relevant evolution problems and achieves useful practical results.

The decision to use *category theory* as an underlying formalism is based on a number of reasons: (a) category theory provides an excellent basis for dealing with *structural relationships*, thus avoiding the need to introduce explicit structuring primitives; (b) the abstractness of category theory allows us to express all ideas *independent of a specific domain*; (c) category theory provides powerful and general support for *composition mechanisms*, which can be used to address the scalability issue.

On the other hand, category theory has the disadvantage that it is difficult to understand because it is a very abstract branch of mathematics. Therefore, a more concrete layer should be defined on top of it to represent software entities. For this purpose we proposed *nested labelled typed graphs*. *Graphs* are an intuitive, visually attractive, general and mathematically well-understood formalism. A *typing* mechanism allows us to distinguish different types of nodes (software entities) and edges (software dependencies) with similar characteristics. A *nesting* mechanism provides an encapsulation and abstraction mechanism to reduce the complexity and to hide unimportant details.

To represent *evolution* of software entities we chose the *algebraic single-pushout* approach towards *conditional graph rewriting* [3,4,5], where application conditions are used to determine when a certain graph production (read: evolution step) is applicable to a given graph (read: software entity). This is essential to detect conflicts between incompatible evolutions of the same software entity.

3. The formalism

By using conditional graph rewriting we can rely on existing theorems and properties like confluence, parallel and sequential independence of graph derivations, pushouts and pullbacks, to provide better support for evolution conflict detection. Due to space limitations we can only present the general idea here. For a more detailed treatment we refer to [7,8].

In order to formally characterise evolution conflicts, we need the notion of parallel and sequential independence. Two graph derivations $G \Rightarrow_{p1} G_1$ and $G \Rightarrow_{p2} G_2$ starting from the same graph G are *parallel independent* if they can be applied one after the other. A similar notion of *sequential independence* means that the order in which two graph productions p_1 and p_2 are applied in a derivation sequence $G \Rightarrow_{p1} G_1 \Rightarrow_{p2} G_2$ is irrelevant. Under certain injectivity constraints, two parallel independent derivations can always be sequentialised, and lead to a unique result graph that is independent of the order in which the productions are applied. This property is called *local confluence*, and is

essential when detecting conflicts between parallel evolutions of the same software entity. An essential distinction can be made between *syntactic* conflicts and *semantic* conflicts.

When two parallel graph derivations $G \Rightarrow_{p1} G_1$ and $G \Rightarrow_{p2} G_2$ are not parallel independent, they cannot be sequentialised, because p_1 is not applicable after p_2 or vice versa (due to a breach of an application condition). If this is the case, we say that a *syntactic conflict* has occurred. Typical examples of this are name conflicts when the label or type of the same node or edge is modified twice, or dangling references when a node is removed while independently an edge to this node was added. By providing a primitive set of graph productions, a complete characterisation can be given of all possible syntactic conflicts in the form of a conflict table (or merge matrix). Alternatively, the conflict table can be defined in terms of the application conditions that are breached. This allows us to facilitate conflict detection significantly.

When the graph derivations $G \Rightarrow_{p1} G_1$ and $G \Rightarrow_{p2} G_2$ can be sequentialised, local confluence guarantees a unique result graph H (by applying $G \Rightarrow_{p1} G_1 \Rightarrow_{p2} H$ or $G \Rightarrow_{p2} G_2 \Rightarrow_{p1} H$). Nevertheless, this graph can still contain semantic incompatibilities because of unexpected interactions between both graph derivations. If this is the case, we say that a *semantic conflict* has occurred. Because detection of such conflicts is inherently undecidable and can depend on the particular situation, we can only take a conservative approach by generating conflict warnings rather than actual conflicts. Formally, a potential semantic conflict can be detected using the category-theoretical notions of *pushout* and *pullback*. While the merge of two graph derivations is defined by the pushout of $G \Rightarrow_{p1} G_1$ and $G \Rightarrow_{p2} G_2$, a semantic conflict warning is issued if the corresponding pullback is not empty, i.e., if the two graph derivations make parallel changes involving the same element.

To summarise, our formalism enables detection of syntactic and semantic evolution conflicts during software merging and software upgrading, by relying on formal properties of graphs and graph rewriting.

4. Experiments

In order to be useful in practice, automated support should be provided. Based on our formal model, a number of useful algorithms were outlined in the thesis: (a) a *conflict detection* algorithm to check syntactic as well as semantic conflicts, and an extension of this algorithm to deal with sequences of evolution steps; (b) a *normalisation* algorithm to remove redundant information in an arbitrary evolution sequence, thus making the evolution process easier to understand and speeding up conflict detection; (c) an *extraction* algorithm to extract evolution transformations if only the original and revised version of a software entity are provided.

Another essential algorithm that would be needed in an industrial setting would be a *conflict resolution* algorithm, but this is much more difficult to define in a domain-independent way. Also, to become of practical value, all these algorithms need to be incorporated into a CASE tool or integrated development environment.

In order to validate the claims of the thesis, we implemented a prototype of the reuse contract formalism and the above algorithms in Prolog. This logic framework was customised to the domains of class diagrams and software architectures to validate the domain-independence. For each customisation, the following actions were performed: (a) define a domain-specific *type graph* by specifying the domain-specific meta model in terms of node types and edge types; (b) specify additional domain-specific well-formedness rules on top of this type graph; (c) translate the domain-specific naming scheme in terms of the domain-independent primitives; (d) specify which of the evolution conflicts generated by the formalism may be ignored in the specific domain, and define all domain-specific evolution conflicts that cannot be detected by the generic formalism; (e) specify domain-specific conflict resolution rules.

5. Contribution

For various reasons the thesis provided a relevant, important and novel contribution to the object-oriented research community, the software evolution community, and even the graph grammar community.

The lack of adequate mechanisms for software evolution is one of the main causes for the current software crisis. Problems typically arise when upgrading to new versions of software, or when merging parallel evolutions during collaborative software development. Object-oriented analysis and design CASE tools, which are commonly accepted

and used to improve the software development process, provide no or poor support for evolution. The thesis addressed this lack of evolution support by providing a formal foundation to deal with specific evolution problems. Based on the formalism, algorithms were defined to provide more automated support for software evolution.

The relevance to the graph rewriting research community is in the practical application of graph rewriting [8]. Even after three decades, this community still focuses more on theoretical rather than practical results. Fortunately, the tide seems to be turning, due to the emergence of efficient and expressive working implementations of graph rewriting systems such as PROGRES [12].

The novelty and importance of the thesis lied in the fact that we showed the feasibility and usefulness of a domain-independent formalism for software evolution. Domain-independence has the main advantage that we can ignore domain-specific details. In order to add support for evolution to a particular domain, it suffices to instantiate the formalism to the specific domain, and all the techniques and formal results for dealing with evolution are immediately applicable to this domain. As such, significantly less effort is required to support evolution than if we would have to implement everything from scratch.

A final technical contribution of the dissertation is that it pays attention to the scalability of reuse contracts, an issue that was not addressed before in full detail. In general, scalability is an important characteristic, in order for any approach to be applicable to large industrial software development.

6. Future work

Despite all these contributions, a lot of work remains to be done:

- Because the main focus of the thesis was on the formal aspects, the results have not yet been applied to large-scale industrial case studies.
- With our current formalism we are only able to detect a restricted set of semantic conflicts. Further research is needed to detect more complex and more interesting kinds of semantic inconsistencies.
- To further validate the general claim of the thesis, we also need to apply our ideas to other aspects of software evolution, such as software restructuring, change propagation, impact analysis and effort estimation.
- Another avenue of research is to apply our ideas at higher levels of abstraction such as design patterns and typical transformations thereof [].

References

1. W. Codenie, K. De Hondt, P. Steyaert and A. Vercammen, “From custom applications to domain-specific frameworks”, *Comm. ACM*, 40(10), 1997, pp. 71-77.

2. M. D’Hondt, *Managing evolution of changing software requirements*, Dissertation, Department of Computer Science, Vrije Universiteit Brussel, 1998.

3. A. Habel, R. Heckel and G. Taentzer, “Graph grammars with negative application conditions”, *Fundamenta Informaticae*, 26(3,4), Special issue on graph transformations, 1996, pp. 287-313.

4. R. Heckel, *Algebraic graph transformations with application conditions*, Dissertation, Technische Universität Berlin, 1995.

5. R. Heckel and A. Wagner, “Ensuring consistency of conditional graph grammars – A constructive approach”, *Lecture Notes in Theoretical Computer Science* 1, 1995.

6. C. Lucas, *Documenting reuse and evolution with reuse contracts*, PhD Thesis, Department of Computer Science, Vrije Universiteit Brussel, September 1997.

7. T. Mens, *A formal foundation for object-oriented software evolution*, PhD Thesis, Department of Computer Science, Vrije Universiteit Brussel, September 1999.

8. T. Mens, “Conditional graph rewriting as a domain-independent formalism for software evolution”, *Proc. Int. Agtive ’99 Workshop: Applications of Graph Transformations with Industrial Relevance*, LNCS 1779, Springer-Verlag, 2000, pp. 127-143.

9. T. Mens, C. Lucas and P. Steyaert, “Supporting disciplined reuse and evolution of UML models”, *Proc. Int. Conf. «UML»’98 - Beyond The Notation, Selected Papers*, LNCS 1618, Springer-Verlag, 1999, pp. 378-392.

10. T. Mens and T. D’Hondt, “Automating support for software evolution in UML”, *J. Automated Software Engineering* 7, Kluwer Academic Publishers, 2000, pp. 39-59.

11. N. Romero, *Managing evolution of software architectures with reuse contracts*, Masters Thesis, Department of Computer Science, Vrije Universiteit Brussel, 1999.

12. A. Schürr, “Introduction to the specification language PROGRES”, *Building tightly-integrated software development environments: the IPSEN approach*, LNCS 1170, Springer-Verlag, 1996, pp. 248-279.

13. P. Steyaert, C. Lucas, K. Mens and T. D’Hondt, “Reuse contracts: managing the evolution of reusable assets”, *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages and Applications*, ACM SIGPLAN Notices, 31(10), ACM Press, 1996, pp. 268-286.