

FireWorks: A Formal Transformation-Based Model-Driven Approach to Features in Product Lines

Mark Ryan¹ and Pierre Yves Schobbens²

¹ School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK

² Institut d'Informatique, University of Namur, Rue Grandgagnage 21, 5000 Namur, Belgium

Abstract. We summarise here the FireWorks approach to features. This approach is well-suited to a product line evolving unpredictably due to intense market pressure. Since it is formal, it can propose automated tools based on model-checking that help the product line manager to detect and correct or avoid feature interference.

1 Introduction

This article recapitulates in a short and accessible form the FireWorks approach to features. While it has been published already in major scientific journals [8] and conferences [3] since its definition in 1997, it deserves the attention of a larger public, we believe.

2 Basic concepts

The FireWorks approach was built based on the observation and experience gained during the development of the new generation of telephone systems. We believe that similar trends can be observed in many markets where complex products enter the market and are expected to evolve rapidly, as they shape and are shaped by the market. This includes the software market (e.g. in banking, games, personal assistants) and mass-markets (automobile, domestic appliances, alarm systems, lift systems) where products must stand out by their individual adaptations, that become specially numerous and complex since these product embed a large amount of software.

The main lessons of experience can be summarized as follows:

1. Ideally, the evolution should be carefully planned and all products should be designed initially to allow maximal reuse of development in the next generation of products.
2. However, the new needs are often unexpected. Careful planning often turns out to be useless, and even harmful since it can delay the time to market and thus the feedback from customers.
3. The new generation of products thus has to reuse development artifacts that have not been designed with (this type of) reuse in mind.
4. The market rapidly tends to be very fragmented and the products have to offer many different mixes of features to different market niches. Unexpected combinations are the norm rather than the exception.
5. New features should be developed so as to be integrated in all these different mixes with minimal human effort. A lot of effort was historically wasted on the *feature interference* problem, where a feature developed for a specific system at a very low level of abstraction was carelessly ported to a similar system but with different features, resulting in unexpected and often very harmful interference between the features. Due to the large number of possible combinations of features and of the programs that implements them, testing has shown its limits: it becomes very time-consuming, specially if the tests are developed by hand.

To alleviate these problems, we propose the following tenets:

1. Formal methods are usually considered as difficult and time-consuming. However, they can be very efficient when applied to abstract models. We propose thus to follow a model-driven approach (MDA) where the systems are first modelled at a very high level of abstraction, which is usually closer to the understanding of the customer or marketing department, to correctly identify the basic needs. These models are then progressively refined until code can be automatically derived. (This means until assembly language for some critical parts).
2. New unexpected needs must still, in principle, follow this development by a cascade of refinements. However, it is highly accelerated by the fact that a very similar family of existing systems is available. Developers then tend to adopt a “cut-and-paste” development style. While it is precisely this style that led to the feature interference problems when performed at the lowest level of abstraction, the fact that we use many layers of abstraction allows one to cross-check extensively and automatically the systems in the family and/or the various abstraction levels. Rather than forbidding the “cut-and-paste” style (that would cause a rejection of the method by the developers), we develop tools to make it safe and to automate it when possible. The editing operations are tracked so as to be reusable, and are recorded as non semantic-preserving model transformations. The parts on which they operate are recorded as their *requires* part (what is required from the input description for this transformation to be applied).
3. If needed, old products can be refactored to ease product evolution. This refactoring can also be tool-supported: the refactoring can be recorded as a transformation, and it can be checked that it preserves the semantics of the current product, or whether it is semantics-preserving on any product.
4. New features are developed on basis of the simplest system possible (its *requires* part) making the job rather easy. They are then combined automatically and only some troublesome combinations need to be examined. This often leads to the refactoring of an existing feature. The combination works as follows:
 - At any given point of time, we have a set of features. Each feature is described by a family of transformations, one for each level of abstraction. Each transformation is usually a sequence of more basic transformations. We can compute the *requires* part of a sequence from its components. For instance, a transformation could be “add the boolean attribute **under maintenance** to each **telephone switch**”. This requires that the product before application of the feature has the class **telephone switch**.
 - Then we compute the possible sequences of feature applications $F_n(\dots(F_1(F_0)))$. A sequence is *possible* if the *requires* part of each feature is provided by the previous features. For uniformity, the simplest products F_0 are considered as features without *requires* or *modifies* parts.
 - We define an *interference of Type I* as a mismatch between levels: a lower-level description is not a refinement of a higher-level description. For instance, a logical description of the black list feature could state that somebody on the black list (*Black*) must be unable to call the given telephone, but its implementation could mistakenly let forwarded calls originating from *Black* in by checking only the forwarding person. An *interference of Type II* is a mismatch between the possible reorderings of the same feature list. This is based on the idea that the customer is usually not aware of the ordering needed to obtain the set of features he would like. The tool will compute the possible orderings by considering the *requires* part of each feature. Several orderings might still be possible: we consider that either they should give the same result, or the designer must select the “good” result and eliminate the other by enriching the *requires* part. In this way, his analysis will also be useful for many other feature combinations.
5. All allowed combinations of features are permanently checked automatically for compatibility. All examples of interference are submitted to human judgment, since all automatic tools can do is to detect mismatches. The correction can be: forbid this combination of features by enriching the *requires* part, modify or generalize some transformations.

We define thus a *feature* as “an added functionality that has added value to stakeholders”. In contrast, Kang *et al.* define a feature as “any prominent and distinctive aspect or characteristic that is visible to various stakeholders (i.e. end-users, domain experts, developers, etc.)” [6]. Kang’s definition is thus descriptive (which is reflected by its weak formal semantics of features: just symbols), while ours is constructive: a feature is an addition (and is reflected in our semantics: a feature describes how to build itself). Also note that our features are wanted (so that they exist also at the stakeholder requirements abstraction level) while Kang’s one are just visible and might have no value to anybody.

3 Formal methods

Formal methods are simply considered as the basis needed for the development of automated tools. In a given domain of application, one or several diagrammatic or textual languages are selected as the most appropriate for the task. Each language must come with its syntax and semantics. Formally, they should be described as an *institution* [4].

Then, the semantics of each language should be related to the semantics of the others: This is done by a common *semantic refinement*. It must be a preorder (reflexive and transitive). If two objects refine each other, they are deemed *semantically equivalent*. For instance, the highest levels of abstraction will often ignore real-time issues, deadlocks, improper memory usage that only belong to the lowest levels. Since each language is equipped with the simplest adequate semantic domain, tools are easier to develop and the reasoning is also more intuitive. These domains linked by their refinements implicitly construct a common semantic domain. Refinement guides the development: a system description is a series of descriptions at increasingly low levels of abstraction, each one refining the previous one. Refining transformations are provided to help in this process. In the best case, they are fully automatic and act as a compiler.

For instance, in [8], we have integrated the temporal logic CTL, the concurrent language CSP, the parallel imperative language SMV, and the programming language C. They all shared the same semantics: transition systems.

We are now building another instance [10] using the language of MSC at the most abstract level, the automata-based language StateCharts at the next level, and the distributed programming language Promela (a distributed extension of C) at the lowest level. We equip all these languages with a game semantics [1, 9].

In each case, the verification is fully automated thanks to model-checking.

We illustrate these concepts on a concrete example in Fig 1.

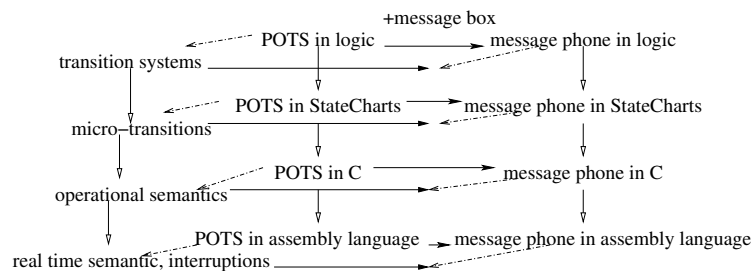


Fig. 1. levels of abstraction and features are orthogonal

The horizontal dimension is that of features: on the left, we have a base system, the Plain Old Telephone Systems (POTS). On the right, we have added to this system the feature “message box”, so that all phone can now retain telephonic messages. The horizontal arrows, labelled with the same ‘+ message box’, are the transformations that add a message box system to an arbitrary phone system. However, they are developed on basis of the simplest possible system, their *requires*

part. Together, these arrows constitute the feature “message box”. These transformations must preserve semantic refinement, but they may operate arbitrary changes. In order to avoid burdening the developer with such proofs, we provide him with a set of basic transformations that have this property, so that any combination of the basic transformations also has this refinement property. In [3] we proposed four types of transformation for game semantics that have this property, but probably more will be found.

The vertical dimension is that of abstraction (upwards) or refinement (downwards). It can, but need not, also be associated to a change of language, and also to a change of semantic domain. The semantic domains are related by the refinement relation. For instance, deadlocks are nonexistent at the level of logic, but are very significant at the level of C programming, and thus have to be introduced in the semantics.

The oblique dimension maps syntax to semantics. Since the features are required to preserve refinements, they will map semantically equivalent base systems to semantically equivalent featured systems, and thus define a function at the semantic level. Thus our drawing is copied in the semantic plane.

The drawing is simplified in one respect: in reality, we do not have one feature, but an ever-increasing set of features that combine in unexpected ways. This flexibility is allowed by the use of transformations, that are defined on all descriptions that satisfy their *requires* part, including unforeseen systems.

In summary, we write a feature as a family of transformations, indexed by the language and abstraction level. Each transformation is written as:

1. a *requires* part stating the minimal elements that the base system to be transformed should possess. The requirements will be matched against the actual system. It might be larger than what is needed to apply the *modifies* part.
2. the *introduces* part states the new elements that the feature introduces to observe the base system or to keep the information it needs to remember. A feature that only has these two first part is monotonic, and this property facilitates the development when present. However, it is seldom the case.
3. its *modifies* part, which transforms the *requires* part. The basic idea, when writing this part, is to avoid to reprogram behaviours that are already in the base system. Thus feature programming will typically modify a very small part, then resort to the base system for most treatments, which makes programming easy. In principle there could be a *delete* part, but we never met this need in practice.

For graphical languages, a graph rewriting approach can be used to express transformations, i.e. the *modifies* part. It is actually also useful for non-graphical languages, since it is more convenient to work on a very abstract syntax expressed as a graph in a metamodel, e.g. MOF [7] or EMF [2].

4 Tool support for feature development

Given this basic setting, we can now provide automatic support.

1. We provide a library of transformations:
 - (a) for refinement in the same language;
 - (b) for translating a language into another, while refining the semantics of the source. Often, we can provide semantics-preserving transformations.
 - (c) for the *modifies* part of features: These need not be semantics preserving nor be a refinement as above, but we impose that a refinement of the source will give a refinement of the target.
2. We can provide rules for sequences of such transformations that compute what is preserved by which sequence, where elementary transformations may belong to different categories sketched just above.

3. We provide automatic checking of refinement by using model-checking or SAT-solving. When a refinement does not hold, a counterexample showing an incorrect behaviour of the implementation will be produced.
4. Features can be proved correct “absolutely”, i.e. without reference to a specific base system. Two proof obligations arise: First, given any base system where each level satisfies the corresponding *requires* part, and with refinements between its levels, one must show how refinements of this base system are mapped to refinements in the featured system. We proposed in [3] a set of transformations that have this property for game semantics. Second, it must be shown that the feature will commute with any other feature that does not depend on it. The same set of transformations can be proved to have this property under easily checkable conditions.
5. Since we assume to have already many refinements available from the development of previous similar systems, a new refinement can often be obtained by replaying a sequence of refining transformations that has proven successful for a similar system, or by slightly modifying it. By tracking these transformations, the transformations that define the feature can often be built automatically.
6. Sometimes, reverse engineering is needed for features that were only developed at the lowest level. This can be done by antirefining transformations. Refining transformations applied backwards already provide a large number of such transformations. They can then be replayed in forward mode for new developments.

5 Conclusion

We have briefly presented the concepts and principles underlying the FireWorks approach to features, without delving into details of its formal semantics, nor presenting examples in detail, for which the reader should consult the bibliography or the Web site: www.cs.bham.ac.uk/~mdr/fireworks.

We hope to have convinced the reader that this rich semantic framework gives a grasp to automated tools, and could thus support principled practical feature development, for which a larger collaboration is sought.

In particular, the following areas need further work:

1. Define a language for transformations that works on any base language (including graphical languages) and yet is readable and easy to use. Efforts in this directions are undertaken within the future QVT OMG standard.
2. Enrich the library of transformations.
3. Study these libraries theoretically: What would be a complete set of transformations? How to guide developer in the selection of adequate transformations? How to help him to generalise his low-level editing transformations so that it works on a large class of base systems [5]?
4. Define approaches to make the proofs easier.
5. Integrate more languages into this approach.
6. Define a game-based semantics for more languages used in practice.
7. Generalise our tool support to infinite systems. Our current tool support is based on model-checking or SAT-solving, techniques that require to introduce strong finiteness assumptions. Thus often our tools are unable to prove correctness for the full problem, but are very useful to build counterexamples for a small instance of the problem. For instance, in [8] we limited the number of telephones in the system to three. In [11] for instance, we propose a technique to deal with infinite systems while still producing counterexamples. More work is needed to overcome the limitations of such approaches.

References

1. Yves Bontemps and Pierre-Yves Schobbens. Synthesis of open reactive systems from scenario-based specification. In Johan Lilius and Felice Balarin, editors, *Proc. Third International Conference on Application of Concurrency to System Design*, 2003.
2. Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 75 Arlington Street, Suite 300, Boston, MA 0211, Aug 2003.
3. Franck Cassez, Mark Dermot Ryan, and Pierre-Yves Schobbens. Proving feature non-interaction with Alternating-Time Temporal Logic. In Stephen Gilmore and Mark Ryan, editors, *Language Constructs for Describing Features*, pages 85–104. Springer-Verlag London Ltd, 2000/2001.
4. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992. See also “Introducing Institutions,” in *Proceedings, Logics of Programming Workshop*, Edward Clarke and Dexter Kozen, editors, Springer Lecture Notes in Computer Science, Volume 164, pages 221–256, 1984.
5. Dimitar Guelev, Mark Ryan, and Pierre-Yves Schobbens. Feature integration as substitution. In Daniel Amyot and Luigi Logrippo, editors, *Proc. Feature Integration Workshop*, 2003.
6. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.
7. OMG. Meta object facility (mof) specification v 1.4. Technical Report formal/2002-04-03, OMG, April 2002.
8. M. C. Plath and M. D. Ryan. Feature integration using a feature construct. Science of Computer Programming, Elsevier Publishing, 1999.
9. Mark Ryan and Pierre-Yves Schobbens. Agents and roles: Refinement in alternating-time temporal logic. *Lecture Notes in Computer Science*, 2333:100–114, 2002.
10. Germain Saval, Yves Bontemps, and Pierre-Yves Schobbens. Model-checking of StateCharts through Promela. Mémoire de DESS Logiciels Sûrs, Institut d’Informatique, Facultés Universitaires de Namur and Université de Paris VI, Rue Grandgagnage 21, 5000 Namur, Belgium, Sept 2004.
11. Pierre Yves Schobbens and Pascal Urso. Automated verification of features with situation calculus. In A. Nunez, editor, *Proc. of FORTE 2004: Work in Progress*, Madrid, 2004.