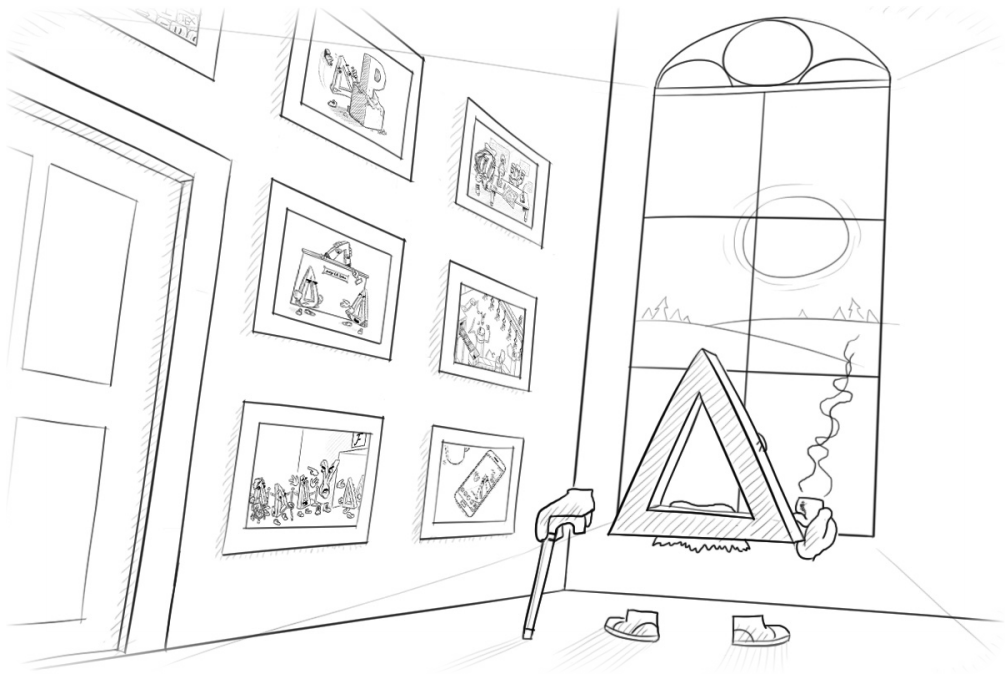
Abstract delta modeling : software product lines and beyond

Helvensteijn, M.

9

Conclusion

A Look Back and a Look Forward



9.1 A Look Back

This is the final chapter. It is time to reflect on what has been achieved and to determine the best way to go forward from here. We first revisit the goals and contributions of Chapters 2 to 8. This section stays away from formalization and focuses instead on motivation and summary.

Chapter 2: Algebraic Delta Modeling

This chapter introduced the basic building blocks of delta modeling. One of those building blocks is the *product*. This rather abstract concept represents the kind of artefact that needs to be manufactured. In practice, a product is built up out of many smaller artefacts. Common examples are packages, classes, methods and fields, in an object oriented programming language, together forming a program. Since this is the original motivation behind delta modeling, and a concept well understood by the target audience of this thesis, the running example of the thesis is based upon this sort of product.

The problem is that the artefacts in such a product almost never map directly to the higher level concept of 'feature'. Indeed, a feature can relate to many classes, and a class can relate to many features. This brings us to the goal of *feature modularity*:

> ***Goal:*** *Find a way to 'group together' code related to the same feature.*

To accomplish feature modularity, *deltas* are introduced. Deltas are an abstract concept embodying the changes to a product —necessary to implement certain functionality— that a developer might make. When a developer needs to implement a feature in a product, he or she is able to modify any number of artefacts in order to do so. Similarly, deltas, too, should be able to specify modifications that break encapsulation and artefact boundaries.

That way, all code related to a feature can be gathered in one place: the delta. In this vision, the rôle of the human developer would change from making changes to the product to writing deltas that do. A separate but related goal is *separation of concerns*:

> ***Goal:*** *Find a way to 'separate' code belonging to different features.*

Deltas are to be true units of functionality, in that they should not implement more than one piece of functionality; that is, they would ideally contain the smallest set of changes that make sense in isolation —and do something useful— but no more, and this way achieve a separation of concerns. This carefully phrased ideal allows for scenarios in which some features extend or depend on others, or are conceptually independent but require access to the same resource. Chapter 3 was dedicated to these kinds of issues.

This chapter thoroughly explores the interaction between deltas and the interaction between a delta and a product, thereby introducing the fundamentals of *Abstract Delta Modeling (ADM)*, built upon by chapters to follow. The notion of *deltoid* is introduced, which contains the full sets of products and deltas representing a specific domain, as well as the semantics

of deltas: how they modify products. By working abstractly, ADM is ready to encode any domain, not limited to any specific programming language, nor even to software itself.

To jumpstart the running example introduced in Section 1.4 —the Editor product line— a concrete deltoid was defined based on an object oriented programming domain. Many subsequent concepts were illustrated through this example.

Various aspects of delta semantics were discussed, such as *partial definedness*, *non-determinism* and *correctness* with regard to a relational specification. A number of *algebraic operations* —such as composition, choice and consensus— are introduced in order to allow syntactic reasoning over deltas. Certain expressiveness properties and a refinement relation are then introduced in order to classify deltoids by what they can do. Finally, it is shown how deltas can encode quarks, a similar concept introduced in related literature.

Chapter 3: Delta Models

In some ways, this chapter described the most fundamentally novel contribution of ADM: *delta models*, which organize deltas into a strict partial *application order*. One delta may be dominant over another, or two deltas may be unrelated by the order. This helps developers express their design intentions, and to contain the complexity of large system.

Goal: *Find a way to mediate between non-commuting feature modules.*

Let's say there are two feature modules (the more general term for what we call deltas), each implementing a different feature in a way that preserves modularity and separates concerns. It is possible that both need access to the same artefact, causing a *conflict* if both are applied together, even if each works fine in isolation. It is for this reason that separation of concerns is not easy to achieve. This chapter proposes three possible ways of mediating such a conflict:

1. Make (minimal) changes to one or both deltas.
2. Impose an application order between the two deltas. The 'dominant' delta is applied last so it can override some of the changes. It should be designed to expect the other delta to go first.
3. Write a *conflict resolving delta*, ordered last so it can make the appropriate changes allowing the original two deltas to work together.

Each is applicable in different situations. For instance, if the conflict is merely an accidental name-collision caused by a lack of communication, the issue is quickly solved by making minimal changes.

Perhaps one of the two features is really a *subfeature* of the other, and it rightly *should* override modifications performed by the other. In that case the deltas should be ordered.

But often enough, neither applies. In this case a conflict-resolving delta is the only way to resolve a conflict properly. It allows the original deltas to remain as they are, and introduces the necessary 'glue code' to facilitate their coexistence. How exactly this is done is a design choice. Development of a conflict resolving delta cannot generally be automated.

Goal: *Find a way to avoid overspecification of the structural organization between feature modules.*

Many features in a system can be *conceptually independent*. This means that they make sense —and should work— in isolation. Ideally, these are even *developed* in isolation (more on this in Chapter 7). When the implementations of two such features are in conflict, this is known as the *optional feature problem*.

Unfortunately, a number of existing systems and formalisms do not have a partially ordered structure between modules, but a linearly ordered one: between any two modules, one can override the changes of the other. If two features are conceptually independent, one of their modules overwriting the changes of the other is most likely a *bug*. Forcing such modules into a linear order is called *overspecification*, and can obscure such bugs. After all, an automated system assembling these feature modules can hardly be expected to know the difference between an accidental overwrite and an intentional one.

By allowing two deltas to be *unordered*, it becomes possible to express that they implement conceptualy independent functionality. If there is ever a conflict, developers can be warned.

Goal: *Find a way to avoid code duplication through the structural organization between feature modules.*

Other existing systems, perhaps in an effort to avoid the overspecification problem, take the opposite approach and do not allow *any* feature module to interfere with any other.

The way to resolve a conflict in such a system is to completely extract the artefacts that clash, and put them into a dedicated module, in such a way that both features work. But in doing so, both modularity and separation of concerns are violated. And when creating a product line, code needs to be duplicated between modules that implement the same artefact for different configurations. Because delta models allow deltas to be ordered when necessary, they do not share this problem.

Apart from introducing the general concepts of delta model, conflict and conflict resolver, this chapter introduced conditions based on these concepts to ensure *unambiguous* product generation.

It then extends the software deltoid to allow *fine-grained* modifications, i.e., manipulating individual statements in methods. This is often neglected in compositional approaches like delta modeling, because unlike classes, methods and fields, statements have no names by which a delta can target their position. *Conjunctive delta model semantics* are then introduced to take advantage of fine-grained modifications. The operation of inserting a statement in a non-deterministically chosen location avoids another type of overspecification by representing the intention: “this method should run this statement at some point; I don’t care when”. It reduces the likelihood that two changes to the same method are seen as a conflict. Finally, the chapter introduced *nested delta models*, which increase expressiveness of a deltoid and offer a useful new modularization technique.

Chapter 4: Product Lines

We've spoken of features before now, but this chapter is where the concept of *feature* is formally introduced and integrated in ADM. These features are merely labels, but play a prominent rôle throughout the rest of the story. They are traditionally used in a *feature model* as a way to identify the possible products of a *product line*, which is defined as a set of products with well-defined commonalities and variabilities. Ideally, a product line should be able to produce any of these products, given only the desired feature selection.

> ***Goal:*** *Develop a technique for organizing a product line code-base in such a way that product generation can be a mechanical process.*

Given that deltas are our feature modules, producing the product corresponding to a specific feature selection is really just a matter of applying the right set of deltas. In ADM, this is done by annotating each delta with an *application condition*: a propositional formula representing the set of feature selections for which it is applicable.

The main challenge here is that each delta must be able to deal gracefully with the presence and absence of other deltas. So developing a product line in which every product behaves properly is the ultimate test of modularity and separation of concerns, because if those principles are adhered to, robust deltas should be an automatic consequence. This does beg the question: what does 'behave properly' even mean?

> ***Goal:*** *Develop a formal concept of product line specification, to be used both in verifying product line correctness and in guiding the implementation process.*

The naive way of giving a product line specification would be to give a separate specification for each of its products. But specifications should be modular and compositional, just like deltas. It is better to write a separate specification for each *feature*. However, an important observation made in this chapter is that it is more realistic to write specifications for *feature combinations* instead. Often, two features that are otherwise independent need to satisfy additional requirements when they are selected together. This is not about conflicts; those are purely an implementation issue. This is about features that inherently interact but shouldn't, or don't inherently interact but should. (Formally speaking there is little difference between the two.)

Apart from providing a characterisation of product line correctness, this chapter lifted a number of concepts from Chapter 3 to the product line level, such as unambiguity and nesting, and introduced a number of other useful concepts. Of particular note is that of *parametric deltas*. Sometimes the delta language is much better at resolving conflicts and implementing interaction than the delta model structure. For those eventualities, deltas can be given access to the feature symbols to be used as Boolean constants. This brings some of the power of annotative variability approaches to the compositional technique of delta modeling. However, caution is adviced in using this technique, as annotative techniques have their disadvantages.

Chapter 5: LaTeX Meets Delta Modeling

Several publications on ADM make the claim that deltas can be used to modularize any kind of artefact — not just source code. An example occasionally brought up is documentation. Indeed, the abstract nature of ADM should allow this, but it had not yet been demonstrated. So what better language for which to implement and demonstrate deltas than the one used to write this very thesis? TeX is a fascinating language; functional by nature, but with the unusual characteristic that practically the entire language can be redefined from within. This brings two opportunities. First, it is a way for deltas to hook into document generation without requiring outside tools; deltas can just be defined in a LaTeX package. Second, the power of the language has caused a number of problems in the LaTeX ecosystem: conflicts between independent packages that access the same resources. The conflict and dependency model of ADM can be adapted to mediate between such packages and, hopefully, alleviate much frustration in the LaTeX community.

> ***Goal:*** *Implement delta modeling for the LaTeX language.*

The first part of the chapter introduced `delta-modules`, a new LaTeX package that brings the main ADM concepts —deltas, partial application orders, feature models— to the LaTeX language, and supports operations quite similar to those of fine-grained software deltas. The package is introduced in a software documentation style, with crosslinks to the relevant formal concepts of the thesis. And what better case study than the thesis itself? (I *am* a fan of self-reference. [95]) In practice, the package may become useful for preparing families of text-books, tech manuals and résumé.

> ***Goal:*** *Use ADM principles to manage dependencies and conflicts between independent LaTeX packages.*

The solution to this problem also takes the form of a package. It is called `pkgloader`, and is similar to `delta-modules` in many respects. But this package has two additional challenges to overcome. First, package authors are not delta authors. We cannot rely on the fact that packages in the wild will limit their tampering to specialized delta operations. And with the full power of a Turing complete language behind them, this means that the problem of detecting conflicts is undecidable. Second, document authors should not be bothered with product line concepts. Ideally, they would just load `pkgloader`… and that's it; things should just work.

To address the first challenge, a centralized knowledge-base is maintained with known package conflicts and resolutions. LaTeX has an active community that can be relied upon to keep such a database up to date. To address the second challenge, the package takes control of the ubiquitous **`\usepackage`** and **`\RequirePackage`** commands. Document authors are already familiar with those, and use them to load packages. When they do, this will be interpreted as selecting a feature. By default, this just loads the package. But with a well-maintained database, the loading order between packages may be changed, and specific glue-code inserted where necessary.

Chapter 6: Delta Logic

Much of ADM is dedicated to the goal of developing syntactic languages and techniques for semantic concepts. Deltas are syntactic. But products (from an ADM point of view) are semantic concepts. Consequently, reasoning about the semantics of deltas requires semantic proof machinery.

> ***Goal:*** *Create a modal logic for reasoning syntactically about the semantics of deltas and their effects on product properties.*

Modal logic fits this problem like a glove. Given any kind of decidable specification language for the product domain, wrapping a multi-modal logic around it enables us to prove that certain deltas implement certain features, that they do not break existing features, and so on. Modal logic was invented specifically to reason about labeled relations, and deltas fit the bill. It is also possible to reason about the algebraic delta operators introduced in Chapter 2. The result is a language reminiscent of dynamic logic, but lacking a construct for iteration. This turns out to be a great advantage, because it keeps the logic decidable. A proof of *strong completeness* is given based on a straightforward translation to a plain multi-modal language, allowing us to simply invoke the completeness of **K** with regard to the class of all frames.

The other two contributions of the chapter are these: First, a proof system for *delta correctness* with regard to modal formulas in the form of Hoare triples, including a proof of its soundness and strong completeness. Second, a proof system for reasoning about specific features on the level of *Kripke models*. The proof system on the Kripke frame level cannot be used because of uniform substitution. We solve this with a translation to *nullary modalities*.

Chapter 7: Delta Modeling Workflow

The formalisation of ADM thus far has been *descriptive*, describing what deltas are, how they work and how they are selected. The other side of the story is *prescriptive*: how are delta models intended to be used? In what way and in which order should a product line be developed so that the full advantage of delta modeling is exploited?

> ***Goal:*** *Describe how delta-based product lines might be built.*

There may be many ways to use delta modeling to good effect. Indeed, many tools are eventually put to innovative uses that were initially unintended. Let's just say that ADM lends itself naturally to a certain way of working which happens to exhibit favorable properties. This chapter presents that workflow under the moniker of *delta modeling workflow (DMW)*. It is split up into well-defined jobs derived from the product line specification.

This formulation makes it explicit that independent features can be developed concurrently and in isolation, allowing for maximal throughput. This way of working counts on the eventual collaborative development of conflict resolving deltas and feature interaction deltas to integrate these

individual efforts. If local constraints are respected, two properties are guaranteed to hold by construction: global unambiguity (i.e., all conflicts are resolved) and total correctness with regard to the specification.

An important concept of the chapter is that of *locality*. Any delta under development need only take into account the existing deltas that occupy subordinate positions in the delta model. Those are the ones the new delta has control over and no other knowledge is required to satisfy local constraints.

In Appendix A, the states and progression of the workflow are represented with a *structural operational semantics*. This is used to prove its beneficial properties.

> ***Goal:*** *Test the delta modeling workflow on an industrial scale system in order to evaluate its practical applicability.*

As a member of the HATS project, I had the opportunity to describe the delta modeling workflow for the *Abstract Behavioral Specification (ABS)* language. I was also in a position to work together with Fredhopper on the *Fredhopper Access Server*, an industrial scale case study which helped validate and improve the workflow. Lessons learned from the case study include a confidence in the thoroughness of the workflow — no features, conflicts or interactions fell through the cracks. Collaboration was possible with the workflow, but this was not yet apparent from the formalism, which was revised accordingly. Flexibility is still a problem. Therefore, adaptation to a more agile approach is planned as future work.

Chapter 8: Dynamic Product Lines

The penultimate chapter of the thesis takes ADM to *runtime*, as deltas are used to update a product to new feature configurations while it is still running. This had already been discussed in previous work. In particular, there has been some effort towards keeping objects in the heap up to date with the latest feature configuration. There has been a noticable lack of work, however, in coming up with strategies for keeping the running product itself up to date. Doing it in the 'static way', applying all deltas every time the feature configuration changes, is too slow. And storing every possible product in advance would require too much memory, as the number of products is generally exponential in the number of features.

> ***Goal:*** *Formulate efficient strategies for reconfiguration of the running product in an ADM-based dynamic product line.*

First, an operational semantics is set up as a framework in which to discuss possible strategies. The abovementioned 'static-style' strategy is formulated and proved correct as an example. A case is made for keeping track of the *differences* between subsequent feature configurations, allowing the system to figure out the minimal delta that needs to be applied to bring the product up to date. However, this still leaves a lot of possibilities. A Mealy machine

model is introduced to compare the pros and cons between various 'difference-based' strategies. In this model, each state represents a feature configuration and each transition represents both a feature configuration difference and a corresponding delta to be applied to the running product.

Eventually, this leads to a strategy that balances the number of stored deltas with the desired runtime efficiency. This strategy is subsequently proved correct using a number of techniques introduced step-by-step throughout the chapter.

Finally, a specialized optimization opportunity is presented. In order to detect feature configuration changes, the environment needs to be monitored. This is naturally modeled with the Mealy machine, in which each feature configuration difference (represented as an input symbol on certain transitions) represents a set of 'sensors' that need to be engaged when occupying certain states. The optimization technique consists of discarding certain transitions from the Mealy machine that are irrelevant, saving energy for the average state occupation of the model. This allows us to segue into the final goal:

Goal: *Develop a profile management app for Android based on the dynamic product line strategies explored in this chapter.*

The software product model defined for the running example of the thesis were intended for structural modification, not to reason about running programs. It has no syntax defined below the statement level, let alone a memory model. It is therefore not a useful example in this chapter. Additionally, the main contribution of the chapter is separate from any issues specific to software. Such issues were already explored by other researchers. In the trend set by the rest of the thesis, the dynamic delta modeling formalism is abstract by nature and can potentially support any domain. I therefore chose a model that is formally simple, yet able to directly illustrate the practical use of the theory.

The case study used to illustrate the formal concepts in this chapter is a mobile application for managing the settings of a smartphone based on any kind of sensory input. While somewhat untraditional as an example of a product line, it actually fits the mold quite readily. Features are represented by predicates over specific environmental quantities, such as GPS location, battery level and calendar appointments. Products are represented by the possible configurations (or *profiles*) of the settings on the phone, such as volume, screen brightness and chat status. By having deltas applying changes to the running profile based on specific environmental conditions (i.e., feature configurations) specified by the user, we essentially have a simple dynamic product line running on the smartphone, as well as a case study with actual practical value: the idea was developed into a working Android application. The 'energy saving' optimization techniques are employed to preserve battery life.

9.2 A Look Forward

This thesis, rather than focusing deeply on any one topic, covers a broad area of research and application. As such there is a great deal of potential for future work. This section shares a glimpse of the possibilities.

9.2.1 Darcs Patch Theory

There is a lot of similarity between delta modeling and Darcs patch theory [97], yet they have very different purposes.

Deltoids can be designed with smart, domain specific operations tailored to the product domain, allowing deltas to be more robust under changing circumstances. This approach might add something to patch theory and version control systems. Conversely, a fundamental aspect of patch theory is that the application of any patch can be reversed. This relates to one algebraic operator that wasn't well-covered in Section 2.6: the converse operator $\breve{}$. Studying the impact of this idea on delta modeling could yield useful results.

Additionally, Darcs patch theory deals with a naturally occurring, partially ordered structure very similar to delta models: that of branches and merges in a version control system. The most significant similarity is that it deals with *conflictors*, which are entities for resolving conflicts. They are quite similar to conflict-resolving deltas, though they seem to have a more complex set of properties due to the added structure of their core setting.

All in all, making a more detailed comparison promises to be a worthwhile persuit.

9.2.2 A Constructive Relation Algebra

Section 2.6.2 briefly discussed the constructivism of the algebraic operators of the relation algebra pioneered by Tarski [101, 102, 175]. Relation algebras (Definition 1.35, page 24) are not constructive, because they contain Boolean algebras (Definition 1.34), which, in turn, contain a non-constructive axiomatisation for the negation operator $^-$ and the full element $\top$.

Slightly weaker than Boolean algebras, and widely known to be constructive, are *Heyting algebras* [92]. They still have a negation operator and a top element, but not as fundamental ingredients. They contain an *implication* operator $\Rightarrow\colon S \times S \to S$ instead. The semantics of $^-$ and $\top$ are weakened to what may be deduced from the axioms $e \Rightarrow g = e^- \sqcup g$ and $\bot^- = \top$.

For delta modeling, however, we suspect that a different approach would be more useful. Rather than use an implication operator as a fundamental ingredient, a *difference operator* $-\colon S \times S \to S$ could be used. This structure is called a *co-Heyting algebra* (or *Brouwer lattice*) [34, 177]. Co-Heyting algebras are the dual of Heyting algebras, and employ the axiom $e - g = e \sqcap g^-$. The difference operator seems to have an intuitive interpretation for deltas, semantically corresponding to set difference $\setminus$.

We have not been able to discover any work applying this idea to full relation algebras, i.e., forming 'co-Heyting relation algebras' and exploring the implications, particularly with regard to the converse operator $\breve{}$. Such research would have a direct and potentially large impact on delta modeling.

9.2.3 Deltas and Traits

Section 2.10 compares deltas with *traits* as a means of implementing product line features. The conclusion was reached that traits are not suitable for the task, at least not in and of themselves. They were designed to enhance code reuse as an alternative to the (inappropriate) use of class inheritance.

Software deltas, on the other hand, were designed to contain the code implementing a specific feature (combination). They were *not* meant for code reuse — at least not in the same sense (sharing code across different products in a product line might also be called reuse). It may be valuable to look at traits and deltas as solving orthogonal goals, and to consider combining them, e.g., to allow deltas to manipulate and insert traits.

9.2.4 A General Development Framework

Now follows one of the more ambitious future work proposals: the implementation of a *general development framework* for building and analyzing delta-based software. This framework should address two problems in particular.

The first problem is that in a compositional approach such as delta modeling, a lot of code will need to be written outside the context where it is eventually applied. This decoupling is a great advantage in the fight against complexity, but programmers are not used to going back and forth between various modules to understand the behavior of a single class or method. They require tool-support to help them reason about a modification in any desired context.

The second problem is that many existing AOP and compositional SPL variability tools only work on a single programming language at a time. Many software features, however, are expressed in multiple languages. For example: HTML for the logical structure of an interface, CSS for its styling and Javascript for its behavior. But current approaches to feature-based modularity require a team to either restrict themselves to one language (per feature) or to manage the variability for each language separately — a maintenance nightmare.

The development framework would likely take the form of a plugin for an existing IDE, such as Eclipse [139] or IntelliJ IDEA [99]. It should include the feature of *code views*: when editing a delta, the programmer would be able to edit it as a whole (as it is stored) or to edit fragments of it directly in the context where they apply. This would bring one of the main benefits of annotative variability techniques, and address the first problem. This concept of code views is reminiscent of CIDE [108] as well as of 'hyperplanes' [148]. But we expect the concept to be much more powerful when applied to the more expressive structure of ADM, leveraged to implement features, coordinate interaction and resolve implementation conflicts in a way more intuitive than has been possible before.

Figure 9.1 shows a mockup of what an Eclipse interface for code views might look like. The controls marked "Delta" show which code artifact is currently being edited (in this case, the `RulePriorities` delta). If it is a delta, a "Code View" can also be chosen, indicating the context in which to edit that delta, consisting of the 'core' code artifact (in this case the `EditGeneralFragment` class) already modified by a chosen set of deltas

from its *local delta model* (a concept described in Chapter 7; in this case, 3 other deltas). A code view is visible in the editor. The editable fragments of the delta appear in white blocks nested in their proper context.

A development effort like this is likely be the most valuable contribution to delta modeling that could be made right now.

9.2.5 LaTeX Deltas

Future work related to the LaTeX packages is likely to be of a *development-* rather than a research nature. As the code is open source, anyone and everyone is encouraged to contribute. One idea is to implement *conjunctive semantics* (Definition 3.26) for the `delta-modules` package.

The `pkgloader` package is still quite limited and there is much that can be done to improve it, though little having to do with delta modeling. But if `pkgloader` becomes widely used in the future, it would become worthwhile to start thinking about the creation of *delta-aware commands* for package authors to use.

9.2.6 Delta Logic

As explained in Section 6.3.4, the delta logics of Chapter 6 are too limited for practical use because the postcondition in a delta contract is not able to refer back to the original product, leaving delta contracts at the level of expressiveness of delta derivation (Section 2.4.3, page 46).

One way to solve this would be a *hybrid language* [22, 40]. Hybrid logics rely on a set of *nominals*, which are propositional variables that are true in exactly one world. They also offer one or more hybrid operators. For instance, the best known hybrid logic is $\mathcal{H}(@,\downarrow)$, which offers a *satisfaction operator* $@_{nom}$ for each nominal *nom*, acting as a sort of modality to travel to the world characterized by *nom*, and an operator $\downarrow nom$, which can dynamically bind a nominal symbol *nom* to the current world. We could use a nominal to characterize the original world, then travel back to it from the resulting world to make certain comparisons.

It turns out that $\mathcal{H}(@,\downarrow)$ is undecidable [43] (though still weaker than first order logic). Fortunately, we wouldn't need to dynamically bind nominals to worlds. For our purpose it would be sufficient that nominals are under implicit universal quantification, as all propositional variables are. The logic $\mathcal{H}(@)$ is decidable [39, 72, 149], making it a perfect candidate to explore in future work.

Additionally, a more concrete exploration of the concept is called for; one that applies delta logic techniques to the verification of actual software product lines. This will illuminate the challenges ahead in embedding other logics into the modal logic, to specify specific software properties.

9.2.7 Delta Modeling Workflow

The delta modeling workflow is really just a first step in defining good development practices for delta modeling. The creation of a development framework as discussed above would help enormously. But in the mean time there are a few improvements that can be made to the formalism.

The workflow assumes a sole-derivation semantics right now. A possibility for future work is to define a good workflow taking advantage of conjunctive semantics (Section 3.5).

Section 7.8 points out that the DMW does not yet conform to the modern practices of *agile development* [130]. In particular, demanding a full specification in advance is now considered unwise. Even so, a number of the core principles of ADM and DMW have great potential for an agile development workflow. The fact that features are isolated and modularized in the first place makes it easy to try new things —start on new features— while confident that it cannot have permanent impact on the code base. Deltas can simply remain inactive until they are deemed ready for production (without the hassle of branching and merging in a text-centric version control system). Moreover, features can be thoroughly developed and tested individually before developers have to worry about testing their interaction, making it easier to divide the work into clear steps. Recall the test driven development scenario described on page 108. As for DMW principles: enforcement of delta locality (Section 7.3) ensures that developers are warned automatically when one of the modules needs to be updated because of changes higher up the delta model, making delta models a safe environment for experimentation.

Creating solid refinement and refactoring theory for delta models is probably the best place to start in adopting agile values and principles into the DMW. This would also help in converting legacy code bases into delta models through a gradual process.

9.2.8 Dynamic Delta Modeling

The hybrid operational semantics for dynamic software product lines presented in Section 8.5 should be much more thoroughly explored. At the moment, the work of Damiani et al. [63, 64] addresses the heap without addressing the product, and DDM addresses the product without addressing the heap. And a good comparison with the work of Muschevici et al. [141] has not yet been made either. Another interesting direction is to develop DDM support for open-adaptivity.

Finally, generation of the Mealy machine based on the static product line implementation is still done in a very roundabout way, assuming an implemented delta derivation operator. A more promising approach, perhaps, is to implement the delta converse operator $\smile$. This may help in effectively 'navigating' the product line delta model at runtime.

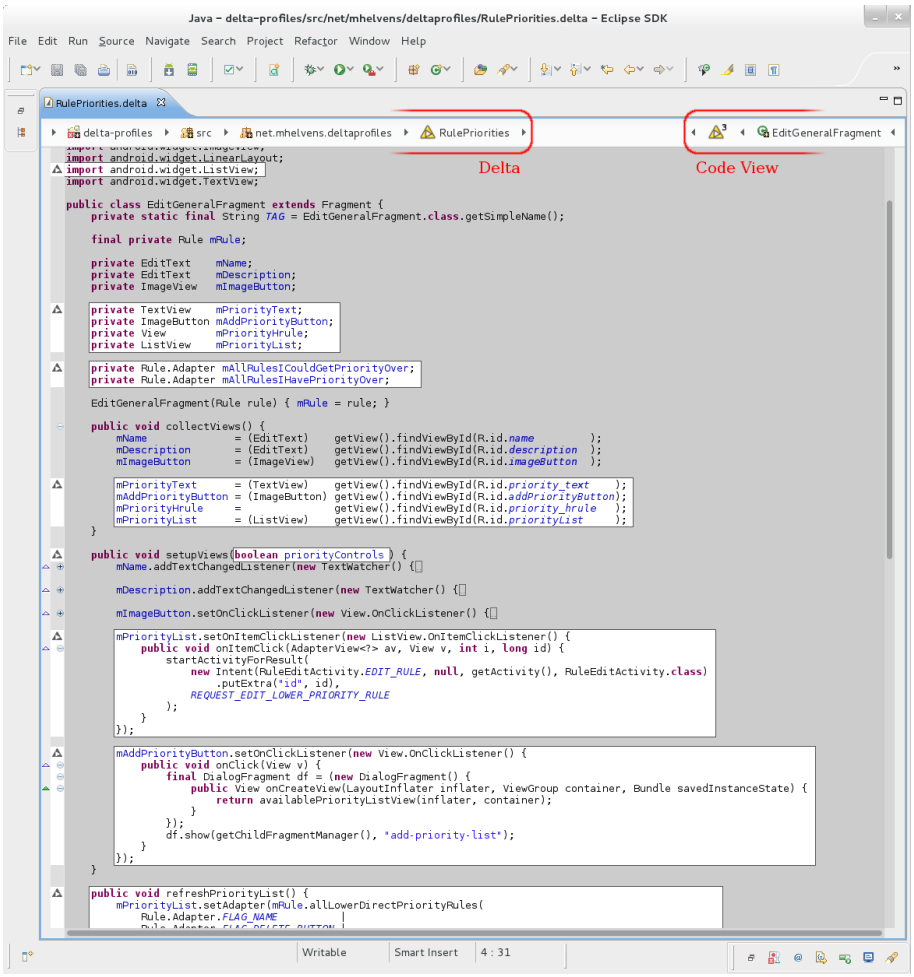


Figure 9.1: A mockup of a possible Eclipse interface for delta modeling with code views. (Incidentally, the code displayed here is part of the Android profile management application from Chapter 8.)