Abstract delta modeling : software product lines and beyond

Helvensteijn, M.

5

LaTeX Meets Delta Modeling

Deltas for Document Generation and Package Management

5.1 Introduction

In this chapter we take a small break from the Editor example, and discuss the contributions of delta modeling to LaTeX, the typesetting language [118].

It has been have mentioned before that ADM, being an abstract formalism, applies to a domain wider than just software. Indeed, the principles of ADM may prove quite valuable for preparing families of technical documents. For example, it is often wise to tailor your curriculum vitæ (or résumé, if you prefer) to the position that you are applying for. A university course may ask students to buy a textbook, but then only use a small part of it; in fact, many textbooks include course outlines or annotations for level of difficulty [21, 36, 116, 155], already assuming that only a part of it will be perused, basically wasting paper. Save the planet — use delta modeling!

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

LaTeX makes a particularly suitable language to showcase delta modeling. The TeX language [115], upon which LaTeX is based, is a domain specific language meant for preparing technical documents, but it also happens to be Turing complete. At its core it is a macro language capable of manipulating arbitrary strings (more accurately called *token lists*). The relatively recent LaTeX3 programming layer [137, 176] includes what is essentially a **while**-language supporting various data-structures.

Moreover, most of the TeX language can be redefined from within the language itself, giving programmers an inordinate amount of control; a fact famously demonstrated by Carlisle [49] when he wrote the following TeX program which generates the full lyrics to "The Twelve Days of Christmas":

```
\let~\catcode~`76~`A13~`F1~`j00~`P2jdefA71F~`7113jdefPALLF
PA''FwPA;;FPAZZFLaLPA//71F71iPAHHFLPAzzFenPASSFthP;A$$FevP
A@@FfPARR717273F737271P;ADDFRgniPAWW71FPATTFvePA**FstRsamP
AGGFRruoPAqq71.72.F717271PAYY7172F727171PA??Fi*LmPA&&71jfi
Fjfi71PAVVFjbigskipRPWGAUU71727374 75,76Fjpar71727375Djifx
:76jelse&U76jfiPLAKK7172F7117271PAXX71FVLnOSeL71SLRyadR@oL
RrhC?yLRurtKFeLPFovPgaTLtReRomL;PABB71 72,73:Fjif.73.jelse
B73:jfiXF71PU71 72,73:PWs;AMM71F71diPAJJFRdriPAQQFRsreLPAI
I71Fo71dPA!!FRgiePBt'el@ lTLqdrYmu.Q.,Ke;vz vzLqpip.Q.,tz;
;Lql.IrsZ.eap,qn.i. i.eLlMaesLdRcna,;!;h htLqm.MRasZ.ilk,%
s$;z zLqs'.ansZ.Ymi,/sx ;LYegseZRyal,@i;@ TLRlogdLrDsW,@;G
LcYlaDLbJsW,SWXJW ree @rzchLhzsW,;WERcesInW qt.'oL.Rtrul;e
doTsW,Wk;Rri@stW aHAHHFndZPpqar.tridgeLinZpe.LtYer.W,:jbye
```

Because of the great expressiveness of the language, LaTeX delta modeling can be implemented from within LaTeX itself —a practice known as *monkey-patching* [138]— though the operating domain for the deltas will need to be limited to specific subsets of the language.

This great expressiveness can also be a problem at times. Since LaTeX has no encapsulation or formal namespacing, LaTeX packages written by different people often conflict with each other — a well-known problem with the LaTeX ecosystem. Luckily, we know how to deal with conflicts.

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

In accordance with the two goals described above, this chapter introduces two LaTeX packages. We first look at the `delta-modules` package in Section 5.2, which can be used to specify families of technical documents and includes an implementation of automated document generation. As a case-study we use this very PhD thesis, a member of what we shall call the *Thesis product line*. Other documents belonging to this product line skip certain topics for a selective reading experience. The source and the members of the Thesis product line can be dynamically generated and downloaded from the following URL:

`http://www.mhelvens.net/phd-thesis`

Section 5.3 describes the ADM-based package manager `pkgloader`, thereby addressing the package management problem. Finally, Section 5.4 offers concluding remarks and Section 5.5 briefly discusses related work.

5.2 `delta-modules`: Deltas for Document Generation

We now introduce the LaTeX delta modeling functionality that was used to organize the content of this thesis. This functionality is provided by the LaTeX package called `delta-modules`. The following subsections describe the process of building a LaTeX product line implementation (Section 4.3) using this package, with the structure of this thesis as an example. The underlying deltoid is implied by the available LaTeX commands, rather than defined formally.

5.2.1 Building the Feature Model

Users of the package declare a feature model using the **`\DeclareFeature`** command:

▷ **5.1. Definition (`\DeclareFeature`):** The **`\DeclareFeature`** command offers the following syntax for the declaration of features and feature-relations:

$$
\begin{array}{rcl}
\langle \textit{declare-f} \rangle & ::= & \texttt{\textbackslash DeclareFeature}\ [\ \texttt{*}\ ]\ \texttt{\{}\ \langle \textit{fid} \rangle\ \texttt{\}}\ \{\ \langle \textit{f-relation} \rangle\ \} \\
\langle \textit{f-relation} \rangle & ::= & \texttt{extends}\ \ \texttt{\{}\ \langle \textit{fid} \rangle\ \{\ \texttt{,}\ \langle \textit{fid} \rangle\ \}\ \texttt{\}} \\
& | & \texttt{requires}\ \texttt{\{}\ \langle \textit{fid} \rangle\ \{\ \texttt{,}\ \langle \textit{fid} \rangle\ \}\ \texttt{\}} \\
& | & \texttt{excludes}\ \texttt{\{}\ \langle \textit{fid} \rangle\ \{\ \texttt{,}\ \langle \textit{fid} \rangle\ \}\ \texttt{\}}
\end{array}
$$

where $\langle \textit{fid} \rangle$ represents a feature name, which can be an arbitrary string, apart from some LaTeX-specific exceptions. ⌟

Each use of the command declares a feature and its relation to other features. **`extends`** indicates that the left-hand feature is a subfeature of the right-hand feature. **`requires`** and **`excludes`** take their respective meanings from feature diagram terminology (Table 4.1). The optional asterisk to the right of the command name indicates that the declared feature is mandatory, relative to its superfeatures, if any.

▷ **5.2. Example (Thesis Feature Model):** The following was used to specify the feature model of the thesis:

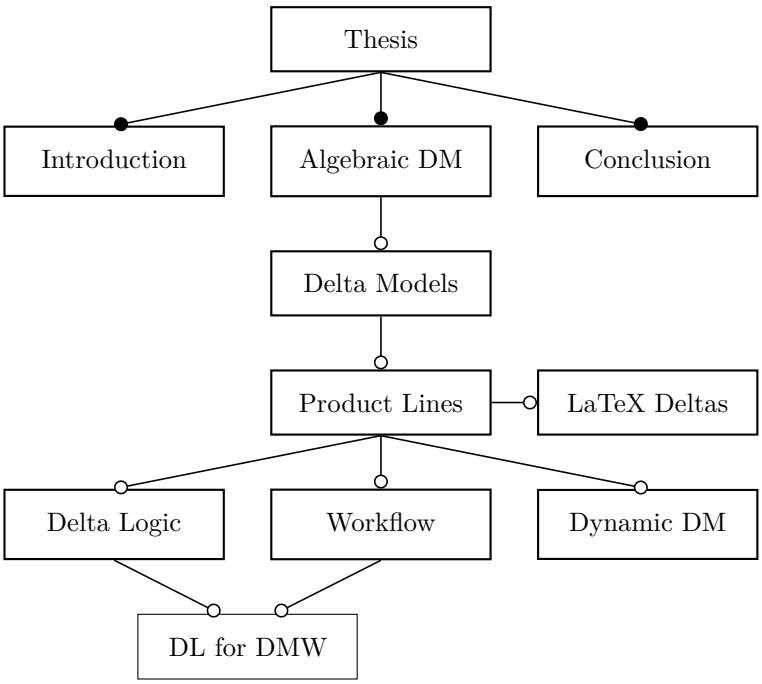

Figure 5.1: The feature model of the Thesis product line.

```
1  \DeclareFeature*{Thesis}
2  \DeclareFeature*{Introduction}  extends {Thesis}
3  \DeclareFeature*{Algebraic DM}  extends {Thesis}
4  \DeclareFeature {Delta Models}  extends {Algebraic DM}
5  \DeclareFeature {Product Lines} extends {Delta Models}
6  \DeclareFeature {LaTeX Deltas}  extends {Product Lines}
7  \DeclareFeature {Delta Logic}   extends {Product Lines}
8  \DeclareFeature {Workflow}      extends {Product Lines}
9  \DeclareFeature {Dynamic DM}    extends {Product Lines}
10 \DeclareFeature*{Conclusion}    extends {Thesis}
11 \DeclareFeature {DL for DMW}    extends {Delta Logic,
12                                          Workflow}
```

The mandatory `Thesis` feature represents the basic document structure, specified outside of any delta, i.e., in the core product. The `Introduction` and `Conclusion` features represent their respective chapters, both also mandatory. All other features except DL for DMW represent research topics, each of which also has a chapter to itself, but the selection of which also influences the content of other chapters. DL for DMW represents the option of using Delta Logic (Chapter 6) in the formalization of the Delta Modeling Workflow (Appendix A). It is the first feature we have seen with more than one direct superfeature. Altogether, the thesis product line supports 22 products.

The corresponding feature diagram is shown in Figure 5.1. Note that it closely follows the suggested reading order on Page 14. But the conclusion, while being last in the narrative structure, should be included regardless of which other features are selected.

5.2.2 Building the Delta Modules

Deltas take the form of a LaTeX environment. This means that their main content is delimited by a **\begin** and an **\end**:

▷ **5.3. Definition (DeltaModule):** LaTeX delta modules have the following syntax:

⟨*d-module*⟩ ::= **\begin{**DeltaModule**}** **{**⟨*did*⟩**}**
{ ⟨*condition*⟩ | ⟨*order*⟩ }
{ ⟨*operation*⟩ }
\end{DeltaModule**}**

⟨*condition*⟩ ::= **if { (** ⟨*ϕ*⟩ **) }**

⟨*ϕ*⟩ ::= ⟨*fid*⟩ | ⟨*ϕ*⟩ **&&** ⟨*ϕ*⟩ | ⟨*ϕ*⟩ **||** ⟨*ϕ*⟩ | **!** ⟨*ϕ*⟩ | **(** ⟨*ϕ*⟩ **)**

⟨*order*⟩ ::= **before { (** ⟨*did*⟩ { **,** ⟨*did*⟩ } **) }**
| **after { (** ⟨*did*⟩ { **,** ⟨*did*⟩ } **) }**

where ⟨*did*⟩ represents a delta name. Deltas occupy a separate namespace from features, so a delta and a feature can share the same name. ⌟

Every delta has a unique name ⟨*did*⟩. An *application condition* is specified through the ⟨*condition*⟩ clause. Its Boolean expressions use the syntax established by LaTeX3 [176], which is the same as for the C language. If multiple ⟨*condition*⟩ clauses are used for a single delta, their disjunction is used. The *application order* is specified through the ⟨*order*⟩ clause, using **before** and **after** to indicate the delta order. An error message is displayed if the eventual application order is not a strict partial order, i.e., if it contains a cycle.

5.2.3 LaTeX Delta Operations

We now specify the possible ⟨*operation*⟩s. What can a LaTeX delta module do? The answer is: anything LaTeX can do. The full language is available within a delta module, though no content should be typeset directly, since delta modules are usually specified in the document preamble (before **\begin{**document**}**). Rather, they should modify commands that are later used to typeset content.

Even so, if we want to be able to reason about delta commutativity (Definition 2.40) —to perform conflict analysis— or take a delta consensus (Section 2.6.2), the full LaTeX language is too rich and unpredictable. We would need to restrict delta modules to simpler operations with more predictable behavior. Since LaTeX processes token lists (which can contain both code and data), the package introduces some delta-aware token list operations, similar to the method- and statement-level operations of fine-grained software-deltas (Section 3.4):

▷ **5.4. Definition (Delta Aware Operations):** The delta-aware operations available as of writing this are the following:

⟨*operation*⟩ ::= **\Replace** ⟨*command*⟩ **with {** ⟨*tokenlist*⟩ **}**
| **\Prepend {** ⟨*tokenlist*⟩ **}** **to** ⟨*command*⟩
| **\Append {** ⟨*tokenlist*⟩ **}** **to** ⟨*command*⟩
| **\Insert {** ⟨*tokenlist*⟩ **} into** ⟨*command*⟩

A ⟨*tokenlist*⟩ is a fragment of LaTeX code, which can later be interpreted as either code or data. A ⟨*command*⟩ is not delimited by braces and is of the form ****⟨***commandname***⟩. (**\Replace**, **\Prepend**, etc. are themselves commands, but they are prohibited from modifying themselves to safeguard the consistency of the running package.) ⌟

Even though the `delta-modules` package currently employs disjunctive semantics (Definition 3.25), the **\Insert** command is already provided for when conjunctive semantics (Definition 3.26) is implemented in the future.

Apart from tracking modifications for conflict analysis, the first three operations map quite directly to LaTeX3 token list commands. **\Insert**, however, needs some special attention. It would not be useful to insert new material between two arbitrary tokens; every letter and command constitutes a token. Instead, **\Insert** interprets the content of ⟨*command*⟩ as a comma-separated list, and inserts the given material at an arbitrary position in that list (including an extra comma). If a comma should not be interpreted as a separator, it can be protected with braces: **{,}**.

▷ **5.5. Example:** Each of the nine deltas implementing a specific thesis chapter look something like this:

```
\begin{DeltaModule} {⟨fid⟩ Delta} if {⟨fid⟩} after { ··· }
   \Insert {⟨fid⟩} into \vpFeatureList
    ⋮
\end{DeltaModule}
```
⌟

The **\Insert** operation above is how we can say that the document you are reading was generated with the features Thesis, Introduction, Algebraic DM, Delta Models, Product Lines, LaTeX, Delta Logic, Workflow, Dynamic DM, and Conclusion. The previous sentence was generated using the following code:

```
\FormatSequence \vpFeatureList { and } {, } {, and }
```

which interprets the given token list as a comma separated sequence, discards redundant commas, then joins the list together using the specified separators.

▷ **5.6. Example:** This is the delta module integrating the LaTeX Deltas feature:

```
\begin{DeltaModule} {LaTeX Deltas} if {LaTeX Deltas}
                                  after {Product Lines}
   \Insert {LaTeX Deltas} into \vpFeatureList
   \Insert {\include{chap:latex-deltas}} into \vpChapters
   \Insert {
      \DescribeChapter{chap:latex-deltas}
```

```
7        This chapter demonstrates the \LaTeX\
8        implementation of delta modeling by
9        documenting two new \LaTeX-packages.
10        ⋮
11    } into \vpChapterSummaries
12    \Insert {
13       \ChapterSummarySubsection{chap:latex-deltas}
14       Several publications on ADM make the claim
15       that deltas can be used to modularize any
16       kind of artefact — not just source code.
17        ⋮
18    } into \vpConclusionSections
19    \Insert {
20       \subsection {\LaTeX\ Deltas}
21       Future work related to the \LaTeX\ packages is
22       likely to be of a \emph{development-} rather than
23       a research nature. As the code is open source,
24        ⋮
25    } into \vpFutureWorkSections
26    ⋮
27 \end{DeltaModule}
```

The \vpChapters command on line 4, as one of the most coarse-grained variation points, inserts the actual chapters into the thesis structure. The \vpChapterSummaries command (line 11) inserts the chapter summaries of Section 1.5.3. \vpConclusionSections and \vpFutureWorkSections (lines 19 and 12) contain subsections for Chapter 9. ⌟

5.2.4 Parametrized Delta Modules

LaTeX delta modules are parametrized, as described in Section 4.5. Anywhere in the document, the **\IfFeatureSelected(TF)** commands can be used:

▷ **5.7. Definition (\IfFeatureSelected(TF)):** The following commands allow inline branching based on the feature selection:

⟨*if-f-selected*⟩ ::= **\IfFeatureSelectedTF** **{**⟨*ϕ*⟩**}** **{**⟨*true*⟩**}** **{**⟨*false*⟩**}**
\IfFeatureSelectedT **{**⟨*ϕ*⟩**}** **{**⟨*true*⟩**}**
\IfFeatureSelectedF **{**⟨*ϕ*⟩**}** **{**⟨*false*⟩**}**

⌟

For the thesis, this is used for the odd crossreference to an optional chapter. It provides more clarity than using an abstractly named variation point.

5.2.5 Choosing a Feature Configuration and Generating the Document

The final delta-related command in the document preamble should be the **\SelectFeatures** command, making the final selection:

▷ **5.8. Definition (`\SelectFeatures`):** The feature selection is made with the following syntax:

⟨*select-f*⟩ ::= **`\SelectFeatures {`** ⟨*fid*⟩ { **`,`** ⟨*fid*⟩ } **`}`** ⌟

Similar commands are provided for reading the feature selection from a file or to request it by standard input from the command-line.

At this point an error message is displayed if the given selection is not a valid feature configuration or if there is no acyclic delta derivation. If no problems arise, the applicable deltas are executed in the proper order. After this, the normal LaTeX compilation process can resume.

▷ **5.9. Example:** The following command was used in the generation of this thesis:

```
1 \SelectFeatures {Thesis, Introduction,
2                  Algebraic DM, Delta Models,
3                  Product Lines, LaTeX,
4                  Delta Logic, Workflow,
5                  Dynamic DM, Conclusion}
```

The code above is also different for each thesis product (besides possibly not being numbered 5.9, which is handled by LaTeX natively). Making sure the code sample is generated with nice formatting for all feature configurations took some effort. Future versions of the package will be equipped with commands specifically meant to make that sort of task easier. ⌟

The application of delta modeling to document preparation may offer a number of practical benefits. The introduction to this chapter introduced the idea of a line of textbooks, each tailored to a specific curriculum or level of difficulty, as well as the idea of maintaining a number of specifically targeted versions of ones CV. Another is the preparation of technical manuals that come with many consumer products. The dominant practice right now is to include the same manual for a range of products, confusing customers as to the features of their new purchase. The preparation of the various versions of this thesis is meant as a proof-of-concept for such use-cases.

5.3 pkgloader: An ADM-based Package Manager

LaTeX can be extended by loading packages, which can add new definitions, and remove and modify existing ones. The `delta-modules` package described in Section 5.2 is one example. Packages can implement domain specific languages, monkey-patch the core language to hook into existing commands, and even change the meaning of individual symbols. Put simply, LaTeX packages have free rein. This power can be quite useful, but makes it too easy for independent package authors to step on each others' toes. CTAN (the Comprehensive TeX Archive Network [168]) is full of conceptually independent packages that cannot be loaded together, or break if they are not loaded in a specific order.

This problem sounds familiar. Let us look at it from a delta modeling perspective. We can see the runtime state of the LaTeX language as a product, package-names as features and package implementations as deltas.[1] From this new perspective, we can describe the problem in more familiar terms. The LaTeX eco-system is suffering from the optional feature problem [111]. But there is no automated package management to speak of. Document authors are told to avoid certain package combinations, or to load packages in some specific order (Action 3.9, page 77). Some of the larger packages are designed to test for the presence of other packages in order to circumvent known conflicts (Action 3.8, page 77). But solving these problems on a case-by-case basis takes time and effort for both document and package authors. It pollutes the code, makes maintenance more difficult, and confuses new users. This is an opportunity for delta modeling to shine. Enter `pkgloader`.

5.3.1 Package Description

LaTeX packages are generally loaded with either the **`\usepackage`** command or the **`\RequirePackage`** command. Similarly, document classes are loaded with **`\documentclass`** or **`\LoadClass`**. Normally when such a command is reached, the corresponding file is loaded right away. The idea behind `pkgloader` is to make it the very first file you load: before the document class, and before any other package. It can then intercept all document class and package loading requests, treat them as a feature selection and load them in the proper order.

▷ **5.10. Example:** The main file for a LaTeX document using `pkgloader`:

```
1  \RequirePackage{pkgloader}
2       ⋮
3      \documentclass{article}
4       ⋮
5      \usepackage{algorithm}      }
6      \usepackage{hyperref}       } any order
7      \usepackage{float}          }
8       ⋮
9  \begin{document}
10      ⋮
11 \end{document}                                   ⌟
```

[1] The analogy goes further. Both plain TeX and LaTeX are really extensions of the primitive language `INITEX`, the initial product (Definition 2.58). LaTeX packages are loaded after the main language extension, which we could reflect in the application order (Definition 3.2).

The area between lines 1 and 9 is called the *pkgloader area*. Inside this area, the loading of all packages and document classes is postponed. It may also be closed explicitly with the **\LoadPackagesNow** command, so that additional code can be run in the preamble. At line 9, a selected delta model is generated (Definition 4.8) and everything is loaded in some valid order, during which conflict resolving code may also be run. If the Example 5.10 code were compiled without `pkgloader`, the given order between `algorithm`, `hyperref` and `float` would cause an error. The main advantage to this approach is that the complexity of dealing with package conflicts is moved to the `pkgloader` package and handled in a systematic manner, taking this burden off the shoulders of the average user. If the package becomes well-used, package authors will be able to develop in a more modular fashion.

5.3.2 Conflict Analysis

Here is the main difficulty: in Section 5.2 we were able to depend on the delta operations of Definition 5.4, safe in the knowledge that there are no primitive TEX commands that might mess things up. But package authors are not delta authors. They can make use of the full TEX language. So the `pkgloader` package does *not* analyze the actual code of each package in order to detect conflicts. Package conflicts are technically what we would call *bad interactions* (Section 3.3), so they cannot be detected automatically.

The package manager is backed by a database of *rules* for recognizing and resolving known conflicts.

▷ **5.11. Example:** The following are examples of such rules:

```
1 \Load {float} before {hyperref}
2 \Load {algorithm} after {hyperref}
3 \Load {fixltx2e} always early
4       because {it fixes some imperfections in LaTeX2e}
5 \Load error if {algorithms && pseudocode}
6       because {they provide the same functionality
7                and conflict on many command names}
```

The first two rules encode some workarounds for the `hyperref` package, which is notorious for causing conflicts. The first one says that `float` must be loaded before `hyperref`. The second rule ensures that `hyperref` is loaded before `algorithm`. These are the rules that would allow the code of Example 5.10 to compile without problems. Note that neither rule actually loads any packages. They simply tell the package manager how to treat certain pairs of packages, should they ever be requested together in a single document.

The third rule states that `fixltx2e` must always be loaded, and must be loaded early. The fourth rule states that the `algorithms` and `pseudocode` packages should never be loaded together. These two rules also include a textual reason, to document the rule, and to include in certain error messages.

5.3.3 \Load Rules

The feature model, partial order and application function for the collective set of packages are built up manually through the **\Load** command. Each invocation sets up a rule. All rules together form the product line implementation (Section 4.3). In contrast to LaTeX delta modules, these rules can come from any number of different sources. A central registry will be maintained by the community, specifying well-known conflicts and resolutions. Individual package authors can supply their own rules, as can document authors. Though ideally, for the average document author, things should 'just work'.

▷ **5.12. Definition (\Load):** The **\Load** command expects the following syntax, some of which inherits from Definition 5.3:

⟨*load-pkg*⟩ ::= **\Load** (⟨*package*⟩ | ⟨*error*⟩) [⟨*reason*⟩]
⟨*package*⟩ ::= [**class**] **{** ⟨*id*⟩ **}** { ⟨*p-condition*⟩ | ⟨*p-order*⟩ }
⟨*error*⟩ ::= **error** { ⟨*condition*⟩ }
⟨*p-condition*⟩ ::= **if {** ⟨ϕ⟩ **}** | **always** | **if loaded**
⟨*p-order*⟩ ::= ⟨*order*⟩ | **early** | **late**
⟨*reason*⟩ ::= **because {** ⟨*text*⟩ **}**

Package names play the rôle of both features and deltas (⟨*fid*⟩ and ⟨*did*⟩ in Definition 5.3). ⌟

We look at each of the clauses of the **\Load** command one by one.

It usually contains a *package description*, consisting of a name, a set of options and a minimal version, just like the **\usepackage** command.

```
1 \Load [options] {package-name} [version]
```

The application condition of every delta *id* is a propositional disjunction which is initialized to *id*, i.e., a package is loaded if it is selected. (To decide otherwise would contradict the expected behavior of **\usepackage**.) The disjunction can be extended by the *condition clause*:

```
1 \Load {pkgA} if {pkgB  pkgC && !pkgD}
```

Alternatively, the condition clause can be **always**, indicating that the rule should be applied under any conditions. Finally, the keywords **if loaded** can be used to apply the rule only if the package named in the package description is requested anyway. This is the default behavior, but the keywords can be included to make it explicit.

There is one exception to the structure described above. Instead of a package description, a rule can contain the **error** keyword, followed by a condition clause, to describe conditions that should never occur — usually invalid package combinations. This refines the feature model. Initially all package combinations are viable. But if two packages are irredeemably incompatible, their combination can be made to generate an error message as follows:

```
1 \Load error if {pkgA && pkgB}
```

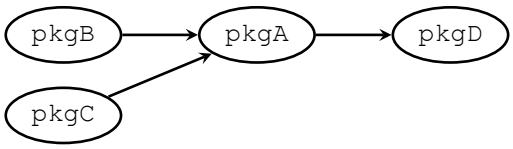

Figure 5.2: A delta diagram showing an example package-loading order.

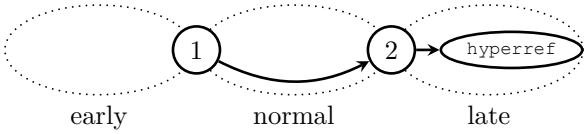

Figure 5.3: A global delta-diagram view of the `pkgloader` package loading order. it shows the two placeholder packages used to impose order between **`early`** and **`late`** packages.

A non-error rule may contain an *order clause*, which forms the application order of the delta model. The **`before`** and **`after`** keywords come from Definition 5.3 and have the same meaning as they do there.

```
1 \Load {pkgA} after {pkgB,pkgC} before {pkgD}
```

This particular rule ensures that if package `pkgA` is ever loaded, it is never loaded before `pkgB` or `pkgC`, and never after `pkgD`, as illustrated in Figure 5.2.

That can take care of specific known package ordering conflicts. But the set of LaTeX packages is constantly growing, and it appears that some big packages should almost always be loaded early in the process, and others should almost always be loaded late. Therefore the **`early`** and **`late`** stages are provided as a fallback mechanism. If two packages are not related by the application order, their loading order may still be decided by their relative stages: **`early`** before 'normal' before **`late`**. That way, conflicts are avoided in a majority of cases.

▷ **5.13. Example:** A typical example is the `hyperref` package, which should almost always be loaded late in the run:

```
1 \Load {hyperref} late                                    ⌟
```

The **`early`** and **`late`** clauses work by ordering the package relative to one of two placeholder packages in the loading order (Figure 5.3).

These two nodes are always present in the graph. Ordering a package **`early`** is intuitively the same as ordering it '**`before {1}`**'. And ordering it **`late`** is the same as ordering it '**`after {2}`**'. All packages that are, after considering all rules, not (indirectly) ordered '**`before {1}`**' or '**`after {2}`**' are automatically ordered '**`after {1} before {2}`**'. A rule can have any number of order clauses, and all are taken into account when one of the conditions of the rule is satisfied.

Finally, a rule can be annotated with the *reason* it was created. This text should be semantically and grammatically correct when following the words "This rule was created because …". It can also be used for citing relevant sources.

▷ **5.14. Example:** The reason clause can be used as follows:

```
\Load {comicsans} always because {that font is awesome!} ⌟
```

This does not have any effect on the behavior of the rule. It is meant for human consumption, though should not be formatted in any way. It is used in certain `pkgloader` error messages (Section 5.3.5) and may eventually be used to generate documentation.

5.3.4 Rulesets

Rules can be placed directly inside the `pkgloader` area, but they can also be bundled in a file. By default, `pkgloader` loads a recommended set of rules, allowing the average user to get started without any hassle. But this behavior can be overwritten using package options:

```
\RequirePackage[recommended=false,
                my-better-rules=true]{pkgloader}
    ⋮
\LoadPackagesNow
```

This means: the recommended rules that are usually preloaded by default should *not* be loaded for this document. Instead, load the `my-better-rules` rule-set. Any user can create rules for their own documents, or distribute custom rulesets, e.g., through CTAN. But primarily, we expect two groups of people to author `pkgloader` rules:

The LaTeX community: The `recommended` ruleset would, ideally, be populated further through the efforts of anyone who diagnoses and solves package conflicts.

Package authors: `pkgloader` will eventually be directly usable for package authors just as for document authors, to include their own rules from right inside their packages. Rather than manually scanning for and fixing potential conflicts, they could leverage `pkgloader`.

5.3.5 Error messages

There are two types of error messages that may be generated by `pkgloader`.

The first type of error message happens when an **`error`** rule is triggered. It looks like this:

> A combination of packages fitting the
> following condition was requested:
> ⟨*condition*⟩
> This is an error because ⟨*reason*⟩.

The second type of error message is a bit more interesting. Since rules can effectively come from any source, the package loading order may not be an order at all; it may be an arbitrary transitive relation (Section 1.7.6).

▷ **5.15. Example:** A cycle can occur when contradictory ordering rules are specified:

```
\Load {pkgX} always before {pkgY}
             because {pkgX is better}
\Load {pkgY} always before {pkgX}
             because {pkgY is better}
```

In practice this could happen if the authors of `pkg1` and `pkg2` independently discover a conflict, and both try to solve it by patching their code and having their own package be loaded last. ⌟

A potential circular ordering is not necessarily a problem, so long as both rules are never applied in the same run. But taken literally, Example 5.15 generates the following error message:

```
There is a cycle in the requested
package loading order:
             pkgX
    --1-->   pkgY
    --2-->   pkgX
The circular reasoning is as follows:
(1) 'pkgX' is to be loaded before
    'pkgY' because pkgX is better.
(2) 'pkgY' is to be loaded before
    'pkgX' because pkgY is better.
```

Whenever this happens, the user may want to reconsider one of their included rulesets, or file a bug-report to the responsible party or parties — especially if the circularity comes from the `recommended` ruleset.

5.3.6 Obtaining these Packages

The two main TEX distributions, TeXlive and MikTeX, only come out with new versions periodically. The `delta-modules` and `pkgloader` packages can be downloaded from CTAN with full documentation. They are dependent on two other new packages: `withargs` and `lt3graph`.

5.4 Conclusion

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 to implement and demonstrate deltas for 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.

This chapter described the LaTeX packages implementing these ideas. The first part of the chapter introduced `delta-modules`, used to modularize the development of technical documents. The second part introduced `pkgloader`, which manages the LaTeX package loading process to resolve conflicts.

5.5 Related Work

The TeXbook by Knuth [115] is a fantastic resource for a grounding in the basics of TeX as a language. The standard book on LaTeX is Lamport's [118], though it does not go much beyond the basics.

As far as we have been able to discover, this is the first time any sort of product line principles have been applied to these languages. There are, however, some packages that attempt to make specific other packages work together. An example is `interfaces` [70], which also provides a consistent interface across the packages it supports. However, none attempt to provide a general solution.

The ability to selectively include certain chapters is offered by the core LaTeX command **`\includeonly`**. But this command is meant purely to save compilation time during development —cross-references to excluded chapters and page numbers are preserved— and **`\include`** operates on a very coarse-grained level — it forcibly starts a new page in the document and is meant for full chapters.

The `delta-modules` and `pkgloader` packages depend on two other packages I have written: `withargs` [87] and `lt3graph` [86]. The former provides a construct for anonymous functions, providing more convenient access to the LaTeX3 argument parcing facilities. The latter implements a graph datastructure for LaTeX3, supporting cycle detection, transitive and reflexive closure generation and vertex iteration in topological order. This is particularly important for the implementation of delta models.